Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Красюка Володимира Костянтиновича

на здобуття ступеня вищої освіти Бакалавра

Система автентифікації користувачів у мікросервісній архітектурі з використанням JWT

Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітня програма Кібербезпека

Шифр КРБКБ.220242.22.02.28 ПЗ

Виконав студент 4 курсу група КБ-22-2 Володимир КРАСЮК
Керівник д-р філософії, ст. викладач Микола СТЕЦЮК
Нормоконтролер д-р філософії Наталія ПЕТЛЯК

До захисту допускаю:
Завідувач кафедри кібербезпеки Юрій КЛЬОЦ

10 06 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Бакалавр

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека

Освітня програма Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ

09 січня 2026 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Красюку Володимиру Костянтиновичу

1 Тема роботи Система автентифікації користувачів у мікросервісній архітектурі з використанням JWT.

Керівник роботи д-р філософії, ст. викладач Микола Стецюк

Затверджено наказом ректора університету від 8 січня 2026 № 7

2 Строк подання студентом кваліфікаційної роботи на кафедру _____

3 Вихідні дані до роботи Проаналізувати предметну область автентифікації користувачів у мікросервісних системах. Дослідити існуючі підходи та рішення у галузі безпеки веб-додатків. Сформулювати постановку задачі та визначити функціональні вимоги до системи. Розробити архітектуру системи автентифікації з використанням JWT. Обґрунтувати вибір інструментів та технології для реалізації. Реалізувати серверну частину додатку на основі Spring Boot. Забезпечити механізм генерації та перевірки JWT токенів. Провести тестування функціональності системи за допомогою Postman.

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ. Аналіз предметної області. Аналіз існуючих рішень автентифікації. Постановка задачі. Проєктування системи автентифікації у мікросервісній архітектурі. Розробка програмного забезпечення системи з використанням JWT та Spring Boot. Реалізація механізмів безпеки та взаємодії мікросервісів. Тестування та оцінка ефективності системи. Висновки.

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
«Послідовність JWT-автентифікації у мікросервісній архітектурі». «Викрадення та повторне використання JWT-токена». «Алгоритм функціонування системи JWT-автентифікаці».

6 Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7 Дата видачі завдання 12 січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
Вибір і затвердження теми кваліфікаційної роботи	Січень-Лютий	
Ознайомлення з предметною областю	Лютий	
Дослідження існуючих рішень	Лютий	
Постановка задачі	Березень	
Визначення загальних принципів рішення задачі	Березень	
Деталізація принципів рішення задачі	Квітень	
Розробка проєктних рішень	Квітень	
Апробація проєктних рішень	Травень	
Оформлення пояснювальної записки згідно вимог	Травень	
Оформлення графічної частини	Травень	
Захист КР	Червень	

Студент _____ Володимир КРАСЮК

Керівник кваліфікаційної роботи _____ Микола СТЕЦЮК

АНОТАЦІЯ

Тема кваліфікаційної роботи: Система автентифікації користувачів у мікросервісній архітектурі з використанням JWT

Автор роботи: Красюк Володимир Костянтинович.

Керівник Роботи: Стецюк Микола Васильович.

Пояснювальна записка: 77 с., 3 додатки, 24 рисунків, 2 таблиці, 45 джерел.

Графічна частина: 3 креслення.

Ключові слова: мікросервісна архітектура, автентифікація, jwt, безпека, токени.

Кваліфікаційна робота бакалавра присвячена розробці системи автентифікації користувачів у мікросервісній архітектурі.

У роботі проведено аналіз сучасних підходів до автентифікації користувачів та розглянуто основні методи забезпечення безпеки у веб-застосунках. Досліджено особливості використання технології JSON Web Token у розподілених системах. Розроблено систему автентифікації, що забезпечує перевірку користувачів та взаємодію між мікросервісами. Реалізовано серверну частину застосунку з використанням Spring Boot та механізмів генерації і перевірки JWT-токенів. Проведено тестування функціональності системи, що підтвердило її ефективність, надійність та можливість застосування у сучасних інформаційних системах.

28.05.2026

ABSTRACT

Subject of qualification work: User authentication system in microservice architecture using JWT.

Author: Krasiuk Volodymyr Kostiantynovych.

Head of work: Stetsiuk Mykola Vasylovych.

Explanatory note: 74 p., 3 appendices, 24 figures, 2 tables, 45 sources

Graphic part: 3 drawings

Keywords: microservice architecture, authentication, jwt, security, tokens.

The bachelor's qualification work is devoted to the development of a user authentication system in a microservice architecture.

The work analyzes modern approaches to user authentication and considers the main methods of ensuring security in web applications. The features of using JSON Web Token technology in distributed systems are studied. A system providing user verification and interaction between microservices has been developed. The server side of the application is implemented using Spring Boot and mechanisms for generating and validating JWT tokens. Testing of the system was carried out, which confirmed its efficiency, reliability, and applicability in modern information systems.

.

28.05.2026

ЗМІСТ

					КРБКБ. 220242.22.02.28 ПЗ			
Зм.	Арк.	№докум.	Підпис	Дата				
Виконав		Красюк В.К.		28.05	Система автентифікації користувачів у мікросервісній архітектурі з використанням JWT Пояснювальна записка	Літера	Аркуш	Аркушів
Перевір.		Стецюк М.В.		29.05		н	6	76
Н.контр.		Петляк Н.С.				ХНУ, КБ-22-2		
Затвер.		Кльоц Ю.П.		10.06.26				

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ

JWT – JSON Web Token

API – Application Programming Interface

HTTP – HyperText Transfer Protocol

HTTPS – HyperText Transfer Protocol Secure

REST – Representational State Transfer

OAuth 2.0 – Authorization Protocol

OpenID Connect – Authentication Protocol

SAML – Security Assertion Markup Language

SSO – Single Sign-On

DB – Database

Auth Service – Authentication Service

Resource Service – Resource Access Service

Bearer Token – Access Token Type

Base64URL – Binary-to-Text Encoding Scheme

ВСТУП

У сучасних інформаційних системах важливу роль відіграє забезпечення безпеки та захисту даних користувачів. З розвитком веб-технологій і переходом до мікросервісної архітектури зростають вимоги до надійної, гнучкої та масштабованої організації процесів автентифікації. Розподілений характер таких систем ускладнює використання традиційних підходів, що базуються на серверних сесіях, і вимагає застосування сучасних рішень.

Актуальність теми зумовлена необхідністю підвищення рівня безпеки інформаційних систем та забезпечення ефективної взаємодії між окремими сервісами. Одним із перспективних підходів є використання токен-орієнтованих механізмів автентифікації, зокрема технології JSON Web Token, яка дозволяє реалізувати безстанну модель перевірки користувачів, підвищити продуктивність системи та спростити її масштабування.

Метою кваліфікаційної роботи є розробка системи автентифікації користувачів у мікросервісній архітектурі з використанням JWT. У процесі виконання роботи розглядаються сучасні підходи до автентифікації, аналізуються існуючі рішення у сфері безпеки веб-застосунків та обґрунтовується вибір технологій для реалізації системи.

Об’єктом дослідження є процес автентифікації користувачів у мікросервісних інформаційних системах.

Предметом дослідження є методи та засоби реалізації автентифікації користувачів із використанням технології JSON Web Token.

1 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ТА РІШЕНЬ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ В МІКРОСЕРВІСНИХ АРХІТЕКТУРАХ

1.1 Особливості мікросервісної архітектури

У сучасній розробці програмного забезпечення дедалі більшої популярності набуває мікросервісна архітектура. Вона використовується для створення масштабованих, гнучких та стійких до відмов систем. Такий підхід дозволяє розділяти складні програмні системи на набір незалежних сервісів, кожен з яких відповідає за виконання окремої бізнес-функції [1].

Мікросервісна архітектура являє собою підхід до проєктування програмних систем, при якому застосунок складається з набору невеликих незалежних сервісів, що взаємодіють між собою через мережу за допомогою стандартних протоколів, найчастіше HTTP або gRPC. Кожен мікросервіс виконує конкретну функцію та може розроблятися, розгортатися і масштабуватися незалежно від інших компонентів системи. Такий підхід протиставляється монолітній архітектурі, де всі функціональні компоненти системи об’єднані в одному застосунку [2].

Основною ідеєю мікросервісної архітектури є розділення системи на невеликі автономні модулі, які можуть розроблятися різними командами та використовувати різні технологічні стекі. Кожен сервіс має власну бізнес-логіку, окрему базу даних або сховище даних та чітко визначений інтерфейс взаємодії з іншими сервісами. Завдяки цьому підвищується гнучкість системи та спрощується її подальший розвиток [3].

До основних принципів побудови мікросервісної архітектури належать незалежність сервісів, слабка зв’язаність компонентів та висока модульність системи. Кожен мікросервіс розробляється як окремий компонент із власним життєвим циклом. Це означає, що зміни в одному сервісі не повинні впливати на роботу інших сервісів. Важливим принципом також є децентралізація управління даними, коли кожен сервіс відповідає за власні дані і не залежить від централізованої бази даних.

Ще одним важливим принципом є використання стандартних протоколів взаємодії між сервісами. Найчастіше для цього використовуються REST API або інші мережеві протоколи, що дозволяє забезпечити незалежність сервісів та спростити інтеграцію між ними. Також важливою особливістю є можливість незалежного масштабування сервісів, що дозволяє оптимально використовувати ресурси системи.

Мікросервісна архітектура має ряд переваг. По-перше, вона забезпечує високу масштабованість системи, оскільки окремі сервіси можуть масштабуватися незалежно один від одного залежно від навантаження. По-друге, вона спрощує процес розробки та підтримки програмного забезпечення, оскільки кожен сервіс має обмежену область відповідальності. По-третє, мікросервіси дозволяють використовувати різні технології для реалізації різних компонентів системи, що підвищує гнучкість розробки.

Разом із перевагами мікросервісна архітектура має і певні недоліки. Однією з основних проблем є складність управління великою кількістю сервісів та їх взаємодією. Крім того, ускладнюється процес тестування та моніторингу системи. Важливим аспектом також є збільшення складності мережевої взаємодії між сервісами, що може призводити до затримок та помилок у роботі системи.

Особливу увагу при використанні мікросервісної архітектури необхідно приділяти питанням безпеки. Оскільки система складається з великої кількості незалежних сервісів, кожен з них може стати потенційною точкою атаки. Однією з ключових проблем є забезпечення надійної автентифікації та авторизації користувачів, а також безпечної взаємодії між сервісами.

У мікросервісних системах часто використовується централізований сервіс автентифікації, який відповідає за перевірку облікових даних користувача та видачу спеціальних токенів доступу. Одним із найпоширеніших рішень для реалізації такого механізму є використання технології JSON Web Token (JWT). Використання токенів дозволяє сервісам перевіряти автентичність запитів без необхідності постійного звернення до центрального сервера автентифікації, що значно підвищує продуктивність та масштабованість системи.

Таким чином, мікросервісна архітектура є сучасним підходом до побудови складних програмних систем, який забезпечує високу гнучкість, масштабованість та незалежність компонентів. Разом з тим вона створює нові виклики у сфері безпеки, що робить актуальним дослідження методів автентифікації та авторизації користувачів у таких системах.

1.2 Основні підходи до автентифікації користувачів

Забезпечення безпеки інформаційних систем є одним із ключових завдань під час розробки сучасних програмних продуктів. Особливу роль у цьому процесі відіграють механізми ідентифікації, автентифікації та авторизації користувачів. У контексті веб-застосунків та розподілених систем, зокрема мікросервісних архітектур, правильна організація процесу автентифікації є критично важливою для захисту даних та запобігання несанкціонованому доступу до ресурсів системи [4].

Автентифікація – це процес перевірки достовірності користувача або системи, що намагається отримати доступ до певного ресурсу. Іншими словами, це процедура підтвердження особи користувача шляхом перевірки наданих ним облікових даних. Найчастіше такими даними є логін та пароль, однак можуть використовуватися й інші фактори перевірки, наприклад криптографічні ключі, біометричні дані або одноразові коди.

Після успішного проходження автентифікації система переходить до наступного етапу авторизації. Авторизація являє собою процес визначення прав доступу автентифікованого користувача до ресурсів системи. На цьому етапі система перевіряє, які саме дії дозволено виконувати користувачу: перегляд інформації, створення або редагування даних, адміністрування системи тощо. Таким чином, автентифікація відповідає на питання «хто є користувачем», тоді як авторизація визначає «що саме дозволено цьому користувачу» [5].

У сучасних інформаційних системах існує декілька основних підходів до реалізації автентифікації користувачів. Найбільш поширеними є сесійна автентифікація, автентифікація на основі токенів та багатофакторна автентифікація. Кожен із цих підходів має свої особливості, переваги та обмеження, що визначають доцільність їх використання у різних типах систем [6].

Одним із традиційних методів є сесійна автентифікація. Даний підхід широко використовується у класичних веб-додатках, побудованих за монолітною архітектурою. Суть сесійної автентифікації полягає в тому, що після успішного введення користувачем облікових даних сервер створює спеціальну сесію, яка зберігає інформацію про автентифікованого користувача. Для ідентифікації цієї сесії клієнту передається унікальний ідентифікатор, який зазвичай зберігається у файлах cookie браузера.

Під час кожного наступного запиту клієнт передає серверу ідентифікатор сесії, що дозволяє серверу визначити користувача та його права доступу. Такий підхід є відносно простим у реалізації та добре підходить для централізованих систем. Однак у випадку мікросервісної архітектури сесійна автентифікація може створювати певні труднощі, пов’язані з необхідністю централізованого зберігання сесій або синхронізації стану між різними сервісами. Це може негативно впливати на масштабованість системи та ускладнювати її розгортання.

Альтернативою сесійному підходу є автентифікація на основі token-based authentication. У цьому випадку після успішної перевірки облікових даних користувача сервер генерує спеціальний токен доступу, який містить інформацію про користувача та його права доступу. Цей токен передається клієнту і надалі використовується для підтвердження автентичності запитів до системи.

На відміну від сесійної автентифікації, токен не потребує зберігання стану на сервері. Кожен сервіс може самостійно перевірити дійсність токена, використовуючи криптографічний підпис або інші механізми перевірки. Це робить токен-орієнтований підхід особливо ефективним для розподілених систем та мікросервісної архітектури. Процес обміну токеном між клієнтом, сервером автентифікації та мікросервісами забезпечує безпечну та безстанну модель

взаємодії компонентів системи. Загальну схему взаємодії компонентів системи при такому підході можна побачити на рисунку 1.1

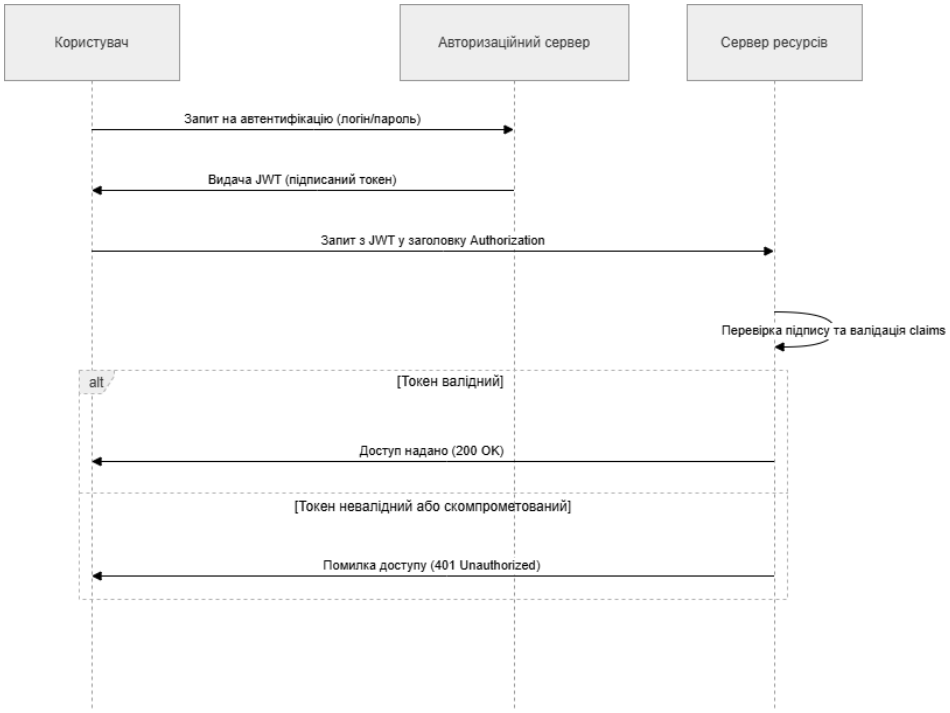


Рисунок 1.1 – Послідовність JWT-автентифікації у мікросервісній архітектурі

Однією з найпоширеніших реалізацій такого підходу є використання JWT, який дозволяє компактно та безпечно передавати інформацію між клієнтом і сервером.

Використання токенів має ряд переваг, серед яких можна виділити високу масштабованість, незалежність сервісів та зменшення навантаження на сервер автентифікації. Однак цей підхід також потребує додаткової уваги до питань безпеки, зокрема правильного управління строком дії токенів, їх шифрування та захисту від несанкціонованого використання.

Ще одним важливим напрямком розвитку систем безпеки є багатофакторна автентифікація. Даний підхід передбачає використання двох або більше незалежних факторів для підтвердження особи користувача. Такими факторами можуть бути знання (наприклад, пароль), володіння (наприклад, мобільний телефон або апаратний токен) та біометричні характеристики користувача (відбиток пальця, розпізнавання обличчя тощо).

Багатофакторна автентифікація значно підвищує рівень безпеки системи, оскільки навіть у випадку компрометації одного з факторів зловмисник не зможе отримати доступ до облікового запису без проходження додаткової перевірки. У сучасних веб-системах часто використовуються одноразові коди підтвердження, які надсилаються користувачу через SMS, електронну пошту або спеціальні мобільні додатки для генерації кодів.

Використання багатофакторної автентифікації особливо актуальне для систем, що працюють з конфіденційною інформацією, фінансовими операціями або персональними даними користувачів. Хоча впровадження такого механізму може дещо ускладнити процес входу в систему, додатковий рівень захисту значно зменшує ризик несанкціонованого доступу.

Таким чином, існує декілька підходів до реалізації автентифікації користувачів, кожен з яких має свої особливості застосування. Сесійна автентифікація є традиційним рішенням для централізованих веб-додатків, тоді як токен-орієнтований підхід більш ефективний у розподілених системах та мікросервісній архітектурі. Багатофакторна автентифікація, у свою чергу, дозволяє значно підвищити рівень безпеки системи шляхом використання декількох факторів перевірки користувача.

У контексті мікросервісних систем особливо актуальним є використання токен-орієнтованих механізмів автентифікації, зокрема технології JSON Web Token, що забезпечує ефективну та масштабовану взаємодію між сервісами. Додатково важливо зазначити, що застосування JWT у поєднанні з сучасними криптографічними алгоритмами дозволяє забезпечити цільність, достовірність та захист даних під час взаємодії між окремими компонентами розподіленої системи.

Крім того, використанням JWT дозволяє спросити процес взаємодії між окремими сервісами та зменшити залежність від централізованого сховища сесій. Це особливо важливо для сучасних розподілених систем, де ключову роль відіграють масштабованість і швидкість обробки запитів. Також важливим аспектом використання JWT є можливість реалізації централізованої системи контролю доступу, що спрощує управління правами користувачами та підвищує

загальний рівень безпеки інформаційної системи. Завдяки цьому JWT є одним із ключових інструментів забезпечення безпеки у сучасних мікросервісних системах.

1.3 Технології та протоколи автентифікації

У сучасних інформаційних системах, особливо тих, що побудовані на основі мікросервісної архітектури, важливу роль відіграють стандартизовані технології та протоколи автентифікації користувачів. Використання таких протоколів дозволяє забезпечити безпечну взаємодію між різними компонентами системи, централізовано керувати доступом до ресурсів та інтегрувати різні сервіси між собою. Найбільш поширеними сучасними рішеннями у сфері автентифікації та авторизації є протоколи OAuth 2.0, OpenID Connect, SAML, а також концепція Single Sign-On (SSO) [7]. Застосування цих технологій дозволяє організувати надійний механізм перевірки користувачів у розподілених системах, де велика кількість сервісів повинна взаємодіяти між собою, забезпечуючи при цьому високий рівень безпеки та зручність використання.

Одним із найпоширеніших протоколів авторизації у сучасних веб-системах є OAuth 2.0. Даний протокол був розроблений як стандарт для надання стороннім додаткам обмеженого доступу до ресурсів користувача без необхідності передавати їм облікові дані, такі як логін і пароль.

Основною ідеєю OAuth 2.0 є використання спеціальних токенів доступу, які видаються сервером авторизації після успішної перевірки користувача. Замість того щоб передавати пароль сторонньому сервісу, користувач проходить автентифікацію на сервері авторизації, після чого додаток отримує токен, який дозволяє виконувати певні дії від імені користувача.

Архітектура OAuth 2.0 включає кілька основних компонентів:

– Resource Owner (власник ресурсу) – користувач, який має доступ до певних даних або ресурсів;

– Client (клієнт) – додаток, який запитує доступ до ресурсів користувача;

– Authorization Server (сервер авторизації) – сервер, що відповідає за перевірку користувача та видачу токенів доступу;

– Resource Server (сервер ресурсів) – сервер, на якому зберігаються захищені дані користувача.

Процес роботи OAuth 2.0 зазвичай складається з кількох етапів. Спочатку клієнтський додаток перенаправляє користувача на сервер авторизації. Користувач проходить автентифікацію та підтверджує надання доступу до своїх даних. Після цього сервер авторизації видає authorization code, який обмінюється на access token. Саме цей токен використовується клієнтським додатком для доступу до захищених ресурсів.

OAuth 2.0 підтримує кілька різних сценаріїв отримання токенів, які називаються grant types. До основних типів належать:

– Authorization Code Grant;

– Client Credentials Grant;

– Resource Owner Password Credentials Grant;

– Implicit Grant.

Найбільш безпечним і поширеним варіантом у сучасних веб-додатках є Authorization Code Grant, особливо у поєднанні з механізмом PKCE (Proof Key for Code Exchange).

Основними перевагами використання OAuth 2.0 є:

– підвищена безпека, оскільки пароль користувача не передається стороннім сервісам;

– можливість обмеження прав доступу;

– централізоване управління доступом;

– зручна інтеграція різних сервісів та додатків.

Саме тому OAuth 2.0 широко використовується у великих платформах, таких як Google, Facebook, GitHub та інших системах, які надають API для сторонніх розробників.

Протокол OpenID Connect (OIDC) є розширенням протоколу OAuth 2.0 і використовується безпосередньо для автентифікації користувачів [8]. Якщо

OAuth 2.0 призначений головним чином для авторизації, тобто для надання доступу до ресурсів, то OpenID Connect дозволяє підтвердити особу користувача.

OpenID Connect додає до механізму OAuth 2.0 спеціальний ID Token, який містить інформацію про автентифікованого користувача [9]. Цей токен зазвичай реалізується у форматі JSON Web Token (JWT) і містить різні атрибути користувача, такі як:

- ідентифікатор користувача;
- ім’я;
- електронна пошта;
- час автентифікації;
- інформація про сервер авторизації.

Після проходження автентифікації клієнтський додаток отримує ID Token, який дозволяє переконатися, що користувач успішно увійшов до системи.

Архітектура OpenID Connect включає такі основні компоненти:

- End User – кінцевий користувач;
- Relying Party (RP) – клієнтський додаток;
- OpenID Provider (OP) – сервер автентифікації.

Процес автентифікації за допомогою OpenID Connect відбувається наступним чином. Користувач перенаправляється на сервер автентифікації, де вводить свої облікові дані. Після успішної перевірки сервер повертає клієнтському додатку ID Token і Access Token. Клієнт може перевірити підпис токена та отримати інформацію про користувача.

Переваги OpenID Connect полягають у наступному:

- простота інтеграції з сучасними веб-додатками;
- підтримка JSON та REST API;
- використання стандартних механізмів OAuth 2.0;
- можливість використання токенів JWT.

OpenID Connect широко застосовується у сучасних хмарних платформах, мобільних додатках та системах мікросервісної архітектури.

Ще одним важливим стандартом автентифікації є SAML (Security Assertion Markup Language). Даний протокол використовується для обміну інформацією про автентифікацію та авторизацію між різними системами.

SAML базується на використанні XML-документів, які називаються assertions. Ці документи містять інформацію про користувача, результати автентифікації та права доступу.

Основними компонентами SAML є:

- Identity Provider (IdP) – сервер, що виконує автентифікацію користувача;
- Service Provider (SP) – система або сервіс, до якого користувач намагається отримати доступ;
- User – користувач системи.

Процес роботи SAML виглядає наступним чином. Користувач намагається отримати доступ до сервісу. Сервіс перенаправляє його до Identity Provider для проходження автентифікації. Після успішної перевірки IdP формує SAML assertion, яка містить інформацію про користувача, і передає її сервісу. На основі цієї інформації сервіс дозволяє користувачу отримати доступ до ресурсів.

SAML широко використовується у корпоративних інформаційних системах, де необхідно інтегрувати різні внутрішні сервіси та забезпечити централізовану систему управління доступом.

До основних переваг SAML належать:

- високий рівень безпеки;
- підтримка корпоративних систем;
- централізована автентифікація користувачів;
- можливість інтеграції великої кількості сервісів.

Однак порівняно з OAuth 2.0 та OpenID Connect, SAML є більш складним у реалізації, оскільки використовує XML та складніші механізми обміну повідомленнями.

Концепція Single Sign-On (SSO) передбачає можливість одноразового входу користувача до системи з подальшим доступом до декількох пов’язаних сервісів без необхідності повторної автентифікації.

У традиційних системах користувач повинен вводити свої облікові дані окремо для кожного сервісу. Це створює певні незручності та може призводити до зниження рівня безпеки, оскільки користувачі змушені запам’ятовувати велику кількість паролів.

SSO дозволяє вирішити цю проблему шляхом централізації процесу автентифікації. Після того як користувач один раз пройшов перевірку на сервері автентифікації, він отримує можливість доступу до всіх підключених сервісів.

У сучасних системах реалізація SSO найчастіше базується на використанні таких технологій:

- OAuth 2.0;
- OpenID Connect;
- SAML;
- Kerberos.

Наприклад, користувач може увійти у систему за допомогою свого облікового запису Google або Microsoft і отримати доступ до різних веб-додатків без повторного введення пароля.

Основними перевагами використання Single Sign-On є:

- зручність для користувачів;
- зменшення кількості паролів;
- централізоване управління автентифікацією;
- підвищення рівня безпеки.

У контексті мікросервісної архітектури SSO дозволяє централізувати процес автентифікації та зменшити навантаження на окремі сервіси. Зазвичай у таких системах використовується спеціальний сервер ідентифікації (Identity Provider), який відповідає за перевірку користувачів та видачу токенів доступу.

Використання протоколів автентифікації у мікросервісних системах

У мікросервісних архітектурах питання автентифікації є особливо важливим через велику кількість незалежних сервісів, які взаємодіють між собою через мережу. Використання стандартних протоколів дозволяє забезпечити єдиний механізм перевірки користувачів та контролю доступу до ресурсів.

Найчастіше у таких системах використовується комбінація OAuth 2.0 та OpenID Connect, які забезпечують як авторизацію, так і автентифікацію користувачів. Сервер авторизації виступає центральним компонентом системи, який відповідає за перевірку облікових даних та видачу токенів доступу.

Після отримання токена клієнтський додаток використовує його для доступу до різних мікросервісів. Кожен сервіс може самостійно перевірити токен, що дозволяє уникнути необхідності повторної автентифікації користувача.

Таким чином, використання сучасних протоколів автентифікації дозволяє значно підвищити безпеку, масштабованість та зручність використання інформаційних систем. OAuth 2.0, OpenID Connect, SAML та Single Sign-On є важливими інструментами побудови надійної системи управління доступом у сучасних веб-додатках та мікросервісних архітектурах.

1.4 Використання JWT для автентифікації у мікросервісних системах

У сучасних інформаційних системах, побудованих на основі мікросервісної архітектури, важливим завданням є забезпечення надійної та ефективної автентифікації користувачів. Через те, що мікросервісні системи складаються з великої кількості незалежних компонентів, які взаємодіють між собою через мережу, традиційні механізми автентифікації, засновані на серверних сесіях, можуть створювати певні труднощі. Зокрема, виникає необхідність централізованого зберігання інформації про сесії користувачів або синхронізації стану між різними сервісами. Це може негативно впливати на масштабованість системи та ускладнювати її архітектуру. Саме тому у сучасних розподілених системах широко використовуються токен-орієнтовані механізми автентифікації, одним із найпоширеніших з яких є використання технології JSON Web Token [10].

JSON Web Token, або скорочено JWT, є відкритим стандартом, що визначає компактний та безпечний спосіб передачі інформації між сторонами у вигляді JSON-об’єкта. Цей стандарт описаний у специфікації RFC 7519 і активно

використовується у веб-додатках, мобільних додатках, API-сервісах та мікросервісних системах. Основна ідея використання JWT полягає у передачі інформації про автентифікованого користувача у вигляді спеціального токена, який може бути перевірений будь-яким сервісом системи без необхідності звернення до центрального сервера автентифікації [11].

Використання JWT дозволяє реалізувати так звану безстанну модель автентифікації. У такій моделі сервер не зберігає інформацію про активні сесії користувачів. Усі необхідні дані для ідентифікації користувача та перевірки його прав доступу містяться безпосередньо у токені. Завдяки цьому кожен сервіс може самостійно перевіряти автентичність запитів, що значно спрощує архітектуру системи та підвищує її масштабованість [12].

JSON Web Token має чітко визначену структуру, яка складається з трьох основних частин: заголовка, корисного навантаження та цифрового підпису. Кожна з цих частин виконує свою функцію у процесі автентифікації та забезпечення безпеки. Частини токена розділяються між собою символом крапки та кодуються за допомогою алгоритму Base64URL. У результаті формується компактний рядок символів, який може передаватися через мережу у складі HTTP-запитів [13].

Перша частина токена називається заголовком. У заголовку міститься інформація про тип токена та алгоритм криптографічного підпису, який використовується для перевірки його достовірності. Найчастіше у заголовку вказується, що тип токена є JWT, а також зазначається алгоритм підпису, наприклад HMAC SHA256 або RSA. Ця інформація дозволяє сервісу визначити спосіб перевірки підпису токена та переконатися у його справжності.

Друга частина токена називається корисним навантаженням або payload. Саме у цій частині міститься основна інформація про користувача та контекст автентифікації. Payload складається з набору тверджень, які називаються claims. Ці твердження можуть містити різні атрибути користувача, такі як його ідентифікатор, ім’я, електронна адреса, роль у системі або інші дані, необхідні для прийняття рішень щодо доступу до ресурсів.

У стандарті JWT визначено кілька типів тверджень. До першого типу належать зареєстровані твердження, які мають стандартні назви та визначене призначення. Наприклад, твердження issuer визначає сторону, яка видала токен. Твердження subject містить ідентифікатор користувача. Твердження audience визначає отримувача токена. Твердження expiration time містить інформацію про час завершення дії токена. Також часто використовується твердження issued at, яке визначає момент створення токена.

Другий тип тверджень – це публічні твердження. Вони можуть використовуватися розробниками для передачі додаткової інформації, яка може бути корисною для сервісів системи. Третій тип – це приватні твердження, які створюються для конкретної системи та використовуються для передачі специфічних параметрів.

Третя частина JWT – це цифровий підпис. Він формується шляхом застосування криптографічного алгоритму до закодованих заголовка та payload разом із секретним ключем або приватним ключем сервера. Цифровий підпис гарантує, що інформація всередині токена не була змінена після його створення. Якщо зловмисник спробує змінити будь-яку частину payload, підпис стане недійсним, і система зможе виявити підробку. Візуальне представлення структури та складу кожної частини токена можна побачити на рис. 1.2.

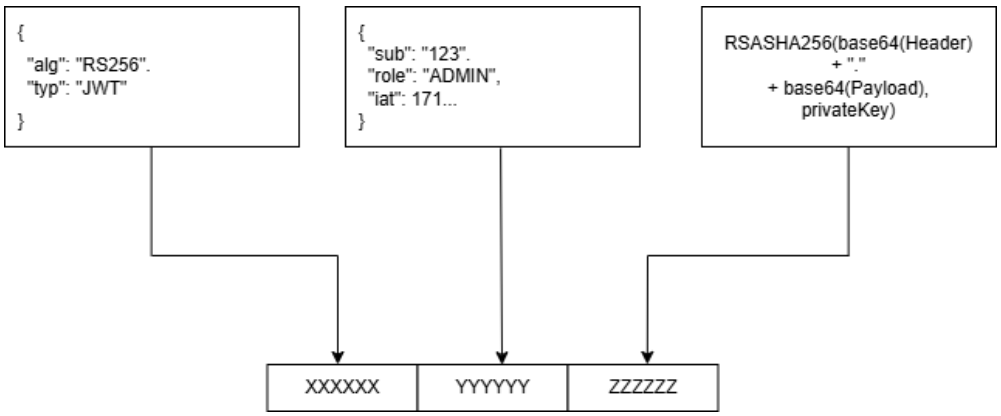


Рисунок 1.2 – Детальна структура та склад компонентів JWT

Принцип роботи JWT у системах автентифікації базується на використанні токенів доступу. Процес починається з того, що користувач вводить свої облікові дані, наприклад логін та пароль, у клієнтському додатку. Ці дані передаються на сервер автентифікації, де відбувається їх перевірка. Якщо облікові дані є правильними, сервер створює JWT-токен, який містить інформацію про користувача та його права доступу.

Після створення токен передається клієнтському додатку, який зберігає його та використовує для подальших запитів до системи. Найчастіше токен передається у заголовку HTTP-запиту Authorization з використанням схеми Bearer. У цьому випадку клієнт додає до кожного запиту рядок, який містить токен доступу.

Коли мікросервіс отримує запит, він витягує токен із заголовка запиту та перевіряє його підпис. Якщо підпис є правильним і токен ще не втратив чинності, сервіс може отримати інформацію про користувача з payload та визначити, чи має користувач право доступу до певного ресурсу. Таким чином, кожен сервіс може самостійно виконувати перевірку токена без необхідності звернення до центрального сервера автентифікації.

У мікросервісних архітектурах JWT часто використовується у поєднанні з API Gateway або спеціальним сервісом автентифікації. Користувач спочатку проходить автентифікацію через сервер ідентифікації, який видає токен доступу. Після цього клієнтський додаток використовує цей токен для доступу до різних мікросервісів системи. Кожен сервіс перевіряє токен і на основі отриманої інформації приймає рішення щодо доступу до своїх ресурсів.

Використання JSON Web Token має низку важливих переваг. Однією з основних переваг є відсутність необхідності зберігати стан сесії на сервері. Це дозволяє значно спростити архітектуру системи та підвищити її масштабованість. У розподілених системах це особливо важливо, оскільки дозволяє легко додавати нові сервери та балансувати навантаження між ними.

Ще однією важливою перевагою є висока продуктивність системи. Перевірка токена виконується локально і не потребує додаткових запитів до бази

даних або інших сервісів. Завдяки цьому час обробки запитів значно зменшується, що є важливим фактором для систем із великим навантаженням.

Також важливою перевагою JWT є універсальність. Оскільки токен використовує стандартний формат JSON, його легко обробляти у різних мовах програмування та технологічних платформах. Це дозволяє використовувати JWT у різних типах додатків, включаючи веб-додатки, мобільні додатки та серверні API.

Крім того, JWT забезпечує високий рівень безпеки завдяки використанню криптографічних алгоритмів підпису. Сервер, який перевіряє токен, може переконатися, що він був створений довіреним джерелом і не був змінений під час передачі. Використання сучасних алгоритмів підпису дозволяє забезпечити надійний захист інформації, що міститься у токені.

Ще однією важливою перевагою є можливість передавати у токені різноманітну інформацію про користувача. Це дозволяє сервісам системи швидко приймати рішення щодо доступу до ресурсів без необхідності виконувати додаткові запити до бази даних. Наприклад, у payload можуть міститися ролі користувача або інші атрибути, які визначають його права доступу.

Незважаючи на численні переваги, використання JWT потребує дотримання певних правил безпеки. Зокрема, важливо обмежувати термін дії токенів, щоб зменшити ризик їх використання у разі компрометації.Також необхідно використовувати лише надійні алгоритми підпису та передавати токени виключно через захищені канали зв’язку, такі як HTTPS.

Крім того, важливим аспектом є правильне управління секретними ключами, які використовуються для підпису токенів. Якщо секретний ключ стане відомим стороннім особам, зловмисники можуть створювати підроблені токени та отримувати несанкціонований доступ до системи. Саме тому ключі повинні зберігатися у захищених сховищах та регулярно змінюватися.

Таким чином, використання JSON Web Token є ефективним та сучасним рішенням для реалізації механізмів автентифікації у мікросервісних системах. Завдяки компактній структурі, можливості самостійної перевірки та високій

продуктивності JWT дозволяє забезпечити безпечну та масштабовану взаємодію між різними компонентами системи. Саме тому дана технологія активно використовується у сучасних веб-сервісах, хмарних платформах та системах, побудованих на основі мікросервісної архітектури.

1.5 Типові загрози та атаки на системи JWT-автентифікації

У сучасних інформаційних системах, побудованих на основі мікросервісної архітектури, використання JWT є одним із найпоширеніших підходів то реалізації механізмів автентифікації та авторизації. Незважаючи на численні переваги даної технології, пов’язані з масштабованістю, автономністю сервісів та відсутністю необхідності зберігати стан сесії на сервері, системи JWT-автентифікації також можуть бути об’єктом різноманітних атак та загроз безпеці [14].

Особливістю JWT є те, що токен містить усю необхідну інформацію про користувача та його права доступу. У випадку компрометації токена зловмисник може отримати несанкціонований доступ до ресурсів системи. Саме тому питання захисту JWT-токенів та безпечної реалізації механізмів автентифікації є важливим аспектом побудови сучасних веб-застосунків і мікросервісних систем [15].

Однією з найпоширеніших загроз є викрадення JWT-токена. Така атака може бути реалізована різними способами, зокрема шляхом перехоплення мережевого трафіку, використання XSS-атак або компрометації клієнтського пристрою користувача. Якщо токен передається через незахищений канал зв’язку без використання протоколу HTTPS, зловмисник може перехопити його та використати для отримання доступу до захищених ресурсів системи.

Особливо небезпечним є зберігання JWT у LocalStorage браузера, оскільки у випадку успішної XSS-атаки шкідливий JavaScript-код може отримати доступ до токена та передати його стороннім особам. Для зменшення ризику викрадення токенів рекомендується використовувати HTTPS та безпечні механізми

зберігання JWT. Загальну схему атаки викрадення JWT-токена наведено на рисунку 1.3.

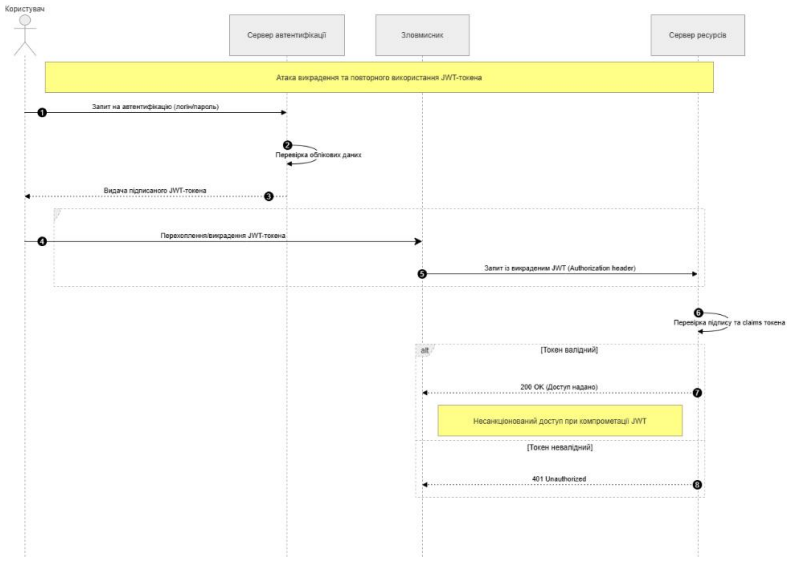


Рисунок 1.3 – Схема викрадення та повторного використання JWT-токена

Ще однією поширеною загрозою є атака повторного використання токена (Replay Attack). Суть даної атаки полягає у тому, що зловмисник повторно використовує раніше перехоплений JWT-токен для отримання доступу до системи. Оскільки JWT є безстанним механізмом автентифікації, сервер ресурсів не завжди може визначити, чи використовується токен законним користувачем, чи сторонньою особою. Для захисту від Replay Attack рекомендується використовувати короткий строк дії токенів, механізми refresh token, а також ротацію токенів доступу. Додатково можуть застосовуватися механізми прив’язки токена до конкретного клієнтського пристрою або IP-адреси.

Важливою проблемою безпеки є також можливість підробки JWT-токенів [16]. У випадку використання слабких секретних ключів або неправильного налаштування алгоритмів підпису зловмисник може створити підроблений токен із довільними claims та отримати несанкціонований доступ до системи. Одним із відомих прикладів є атака, пов’язана з використанням значення alg=none, коли сервер некоректно перевіряє цифровий підпис токена та приймає непідписаний JWT як дійсний. Для запобігання таким атакам необхідно використовувати лише

надійні криптографічні алгоритми, такі як HS256 або RS256, а також забезпечувати обов'язкову перевірку цифрового підпису токена на сервері ресурсів. Секретні ключі повинні мати достатню складність і зберігатися у захищених сховищах. Для захисту від brute-force атак, використовується механізми rate limiting, тимчасове блокування облікових записів, CAPTCHA, а також багатофакторна автентифікація. Таким чином, системи JWT-автентифікації, незважаючи на високу ефективність та масштабованість, потребують комплексного підходу до забезпечення безпеки. Основними загрозами є викрадення токенів, повторне використання JWT, підробка цифрового підпису, підвищення привілеїв та атаки на облікові записи користувачів. Для мінімізації ризиків необхідно використовувати захищені канали зв'язку, надійні алгоритми підпису, обмеження строку дії токенів, багатофакторну автентифікацію та механізми контролю доступу [17].

1.6 Постановка задачі

У попередніх розділах було розглянуто особливості мікросервісної архітектури, основні підходи до автентифікації користувачів, сучасні протоколи безпеки та технології, що застосовуються для організації безпечного доступу до інформаційних систем [18]. Також було проаналізовано існуючі рішення для реалізації механізмів автентифікації, які використовуються у сучасних веб-додатках та розподілених системах [19]. Проведений аналіз показав, що для мікросервісних систем особливо важливо використовувати механізми автентифікації, які забезпечують високу масштабованість, безпеку та ефективну взаємодію між сервісами [20].

У зв'язку з цим виникає необхідність розробки системи автентифікації користувачів для мікросервісного API, яка дозволить забезпечити надійний контроль доступу до ресурсів системи [21]. Метою даної роботи є створення системи автентифікації з використанням фреймворку Spring Boot та технології

JSON Web Token. Така система повинна забезпечувати перевірку облікових даних користувачів, формування токенів доступу після успішної автентифікації та подальшу перевірку цих токенів під час звернення до захищених ресурсів API [23].

Розроблювана система повинна виконувати основні функції автентифікації та контролю доступу [24]. Зокрема, вона повинна забезпечувати обробку запитів на вхід у систему, генерацію JWT токенів після успішної перевірки облікових даних, а також перевірку достовірності токенів під час звернення користувачів до мікросервісів. Використання JWT дозволяє реалізувати безстанну модель автентифікації, при якій сервер не зберігає інформацію про сесії користувачів, що є важливим для ефективної роботи мікросервісної архітектури.

Для досягнення поставленої мети необхідно вирішити ряд задач, пов’язаних із дослідженням, проєктуванням та реалізацією системи автентифікації. Зокрема, необхідно проаналізувати сучасні підходи до автентифікації користувачів, дослідити технології та протоколи, що використовуються у мікросервісних системах, розробити архітектуру системи автентифікації для мікросервісного API, а також реалізувати механізм генерації та перевірки JWT токенів у середовищі Spring Boot. Реалізація такої системи дозволить забезпечити безпечний доступ до ресурсів API та підвищити рівень захисту інформаційної систем. Крім того, розробка системи автентифікації для мікросервісного API спрямована на забезпечення централізованого та безпечного механізму контролю доступу до ресурсів інформаційної системи.

Крім основних механізмів автентифікації, система повинна забезпечувати належний рівень захисту під час взаємодії між окремими мікросервісами. Для цього необхідно реалізувати перевірку цифрового підпису JWT-токенів, контроль строку їх дії та захист від несанкціонованого використання токенів доступу. Важливим аспектом також є використання захищених каналів передачі даних та дотримання сучасних вимог інформаційної безпеки під час обробки запитів користувачів і взаємодії між сервісами системи.

У результаті виконання роботи планується створити програмну систему автентифікації, яка забезпечить централізовану перевірку користувачів, безпечний доступ до ресурсів API та можливість подальшого масштабування системи відповідно до потреб сучасних мікросервісних середовищ. Практична реалізація такого рішення дозволить підвищити рівень захисту інформаційної системи, спростити управління доступом користувачів та забезпечити стабільну взаємодію між окремими компонентами програмного середовища. Розроблювана система також повинна забезпечувати гнучкість подальшого розвитку та можливість інтеграції з іншими сервісами й компонентами інформаційної інфраструктури. Використання мікросервісного підходу та технології JWT дозволить створити масштабоване рішення, яке може ефективно функціонувати в умовах високого навантаження та великої кількості одночасних запитів користувачів. Очікується, що впровадження такої системи сприятиме підвищенню надійності роботи API, покращенню безпеки взаємодії між сервісами та зменшенню ризиків несанкціонованого доступу до інформаційних ресурсів системи.

2 ПРОЄКТУВАННЯ СИСТЕМИ АВТЕНТИФІКАЦІЇ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

2.1 Визначення вимог до розроблюваної системи

Побудова надійної архітектури системи автентифікації вимагає реалізації ряду критичних умов, що базуються на конкретних технічних рішеннях і виходять за межі загального теоретичного опису. Фундаментальним аспектом є перехід до повністю безстанної моделі шляхом впровадження JWT, що дозволяє уникнути проблем із синхронізацією стану між незалежними вузлами системи. Для забезпечення високого рівня безпеки та автономності верифікації передбачено використання асиметричного шифрування за алгоритмом RSA-256, де приватний ключ зберігається виключно на стороні сервісу автентифікації для формування підпису токенів, а ресурсні сервіси використовують публічний ключ для їхньої незалежної перевірки без додаткових запитів до центрального сервера. Враховуючи динамічну природу мікросервісного середовища, критичною умовою є впровадження інструментів Service Discovery для автоматизації процесів реєстрації та виявлення сервісів. Захист облікових даних у базі даних реалізується за допомогою алгоритму хешування BCrypt із обов’язковим використанням солі, що гарантує стійкість до спроб компрометації. Гнучке управління доступом забезпечується через модель RBAC, де інформація про ролі користувача інтегрується безпосередньо в Payload токена, дозволяючи кожному мікросервісу миттєво визначати права доступу на основі отриманих Claims. Таким чином, виконання зазначених технічних вимог дозволяє сформувати цілісну екосистему розподіленої довіри, у якій кожне архітектурне рішення безпосередньо працює на підвищення відмовостійкості та загальної захищеності інформаційної системи [25].

Першим етапом розробки стало виділення функціональних вимог, які описують конкретні дії та можливості майбутньої системи. Цими вимогами було охоплено повний шлях користувача в системі, починаючи з моменту створення облікового запису та завершуючи отриманням доступу до захищених ресурсів.

Головним завданням було визначено створення централізованого вузла, відповідального за ідентифікацію особи та видачу цифрових перепусток у формі токенів. Реалізацію було проведено через декілька ключових етапів, серед яких особливу увагу було приділено обробці реєстраційних даних та валідації паролів. Повний перелік функціональних можливостей, впроваджених у межах кваліфікаційної роботи, наведено у таблиці 2.1.

Таблиця 2.1 – Вимоги до системи автентифікації користувачів

Назва вимоги	Зміст та технічне обґрунтування вимоги
Реєстрація нових користувачів	Впровадження захищеної реєстрації через валідацію об'єкта RegistrationRequest та обов'язкове хешування паролів алгоритмом BCrypt перед збереженням у PostgreSQL. Це забезпечує незворотність облікових даних та їх стійкість до компрометації чи атак методом перебору
Автентифікація за паролем	Перевірка відповідності введених у LoginRequest даних за допомогою вбудованих механізмів Spring Security та власної логіки
Генерація безпечних токенів	Створення унікальних JWT токенів після успішного входу які містять необхідні права доступу та ідентифікатори користувача
Рольове розмежування доступу	Підтримка сутностей Role що дозволяє надавати або обмежувати доступ до певних частин API залежно від повноважень особи
Самостійна перевірка токенів	Здатність будь-якого мікросервісу системи автоматично валідувати підпис отриманого токена за допомогою фільтра безпеки JwtFilter
Керування налаштуваннями	Централізоване отримання параметрів безпеки та ключів шифрування через окремий сервіс конфігурацій Config Server

Після визначення функціонального складу системи стає очевидним, що вона має функціонувати як цілісне середовище з чітким розмежуванням обов'язків між компонентами, де процеси автентифікації локалізуються в одному вузлі, а верифікація прав делегується іншим сервісам. Поряд із безпосереднім функціоналом, критичного значення набувають нефункціональні вимоги, які визначають надійність, масштабованість та стійкість архітектури до зовнішніх загроз. З метою забезпечення максимального рівня безпеки передбачено використання асиметричного шифрування RSA для формування цифрового

підпису токенів, що дозволяє повністю виключити передачу секретних ключів мережею та використовувати лише публічні ключі для перевірки справжності даних. Крім того, система проєктується за принципом безстанності, що гарантує незалежність кожного запиту та дозволяє ефективно розподіляти навантаження між вільними ресурсами серверної інфраструктури. Детальний опис цих технічних параметрів та критеріїв якості системи наведено у таблиці 2.2.

Таблиця 2.2 – Нефункціональні вимоги до системи

Категорія вимоги	Технічний опис та спосіб реалізації вимоги
Криптографічний захист	Застосування асиметричного алгоритму RSA 256 де підпис створюється приватним ключем а перевіряється публічним файлом public key pem
Архітектурна модель	Дотримання принципу Stateless при якому вся інформація про автентифікацію міститься лише в самому токені без використання сесій сервера
Ефективність роботи	Забезпечення швидкої перевірки токенів на стороні мікросервісів без додаткових звернень до бази даних чи інших сервісів системи
Горизонтальне масштабування	Можливість динамічного додавання нових вузлів системи завдяки реєстрації сервісів у Discovery Service на базі Eureka
Контейнеризація середовища	Підготовка інфраструктури для роботи в ізольованих контейнерах Docker що гарантує однакову роботу системи на різних серверах
Гнучкість конфігурування	Здатність змінювати параметри безпеки в реальному часі без необхідності повторного збирання всього програмного коду сервісів

Дотримання зазначених технічних стандартів гарантує високу функціональність системи, її відповідність сучасним вимогам та стійкість до потенційних загроз [26]. Застосування фреймворків Spring Boot та Spring Cloud забезпечує автоматизацію рутинних процесів розробки, дозволяючи зосередити основну увагу на реалізації логіки безпеки. Обраний підхід також враховує можливість подальшого розширення системи, оскільки дозволяє інтегрувати нові бізнес-сервіси без необхідності модифікації вже впроваджених механізмів автентифікації.

Сформовані вимоги виступають основою для подальшого етапу розробки. Чітке розуміння алгоритмів реагування системи на дії користувачів та вибір

стабільних технологічних рішень дозволяють перейти до безпосереднього проєктування архітектури. У наступному підрозділі буде детально описано процес втілення цих вимог у конкретну структуру мікросервісів та механізми їхньої взаємодії для створення безпечного інформаційного середовища [27].

2.2 Проєктування архітектури мікросервісного середовища

Побудова надійної архітектури розподіленої системи базується на принципі децентралізованої довіри, що дозволяє забезпечити високу стійкість до компрометації даних та відмовостійкість кожного окремого вузла. Загальну логіку взаємодії мікросервісів та шлях проходження запитів користувача наведено на розробленій схемі архітектури, яку можна побачити на рисунку 2.1. У процесі проєктування архітектури особлива увага приділяється розподілу функціональності між окремими мікросервісами та забезпеченню їх незалежної роботи. Взаємодія між сервісами здійснюється через захищені API-запити, що дозволяє підвищити рівень безпеки та спростити масштабування системи.

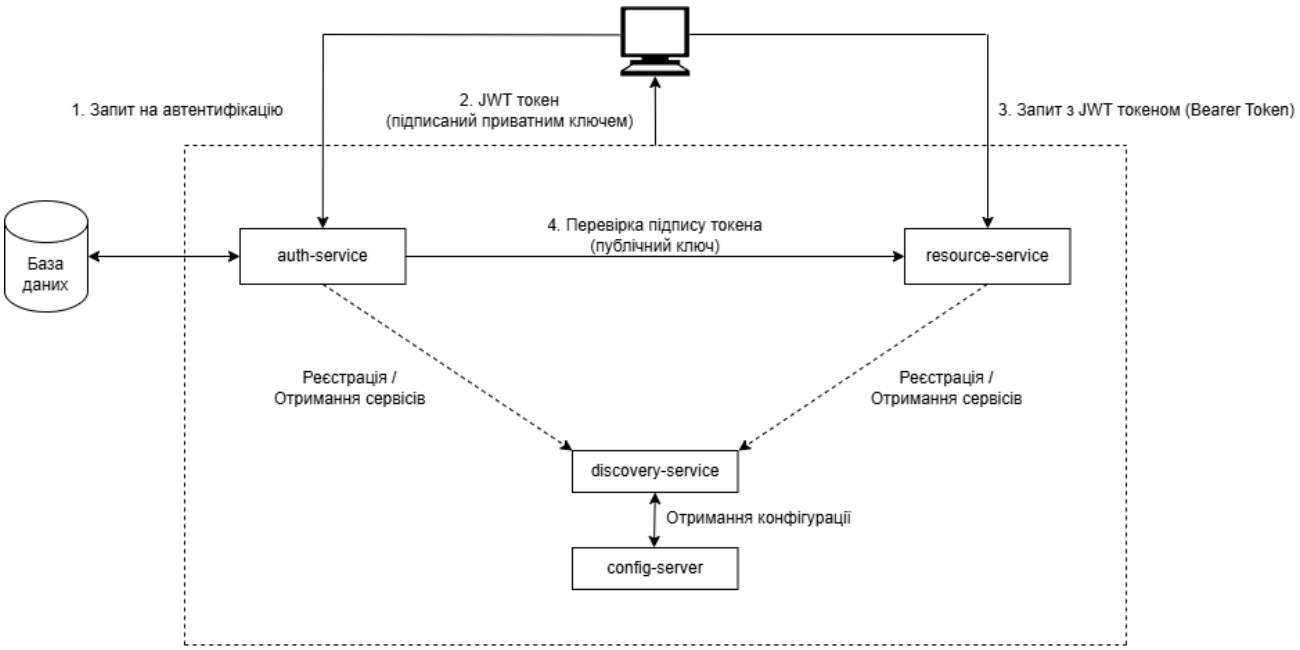


Рисунок 2.1 – Архітектура мікросервісної системи автентифікації та логіка взаємодії компонентів

Ключовим елементом безпеки у цій структурі виступає auth-service, який виконує роль централізованого видавця токенів та є єдиним компонентом, що володіє приватним ключем RSA для формування цифрового підпису, що повністю виключає можливість генерації або підробки токенів іншими вузлами системи. Крім того, цим сервісом здійснюється безпечне збереження облікових записів із використанням алгоритму BCrypt для незворотного хешування паролів із додаванням солі, що гарантує захист конфіденційної інформації навіть у разі компрометації бази даних PostgreSQL. Ресурсний сервіс (resource-service) відповідає за надання доступу до бізнес-даних, причому його головна функція безпеки полягає в автономній валідації JWT за допомогою публічного ключа RSA, що дозволяє перевіряти справжність кожного запиту без додаткового мережевого трафіку до сервера автентифікації та мінімізує ризик DoS-атак на ядро системи. Цілісність внутрішньої комунікації підтримується через discovery-service на базі Netflix Eureka, який забезпечує динамічну реєстрацію мікросервісів та дозволяє їм взаємодіяти за логічними іменами, що значно ускладнює проведення атак типу перехоплення трафіку всередині ізольованої мережі. Додатковий рівень захисту реалізується за допомогою config-server, який виступає централізованим сховищем секретів і ключів, дозволяючи винести паролі та конфіденційні налаштування за межі вихідного коду застосунків і передавати їх безпосередньо в оперативну пам'ять мікросервісів під час їхнього запуску [28]. Сама база даних PostgreSQL інтегрована виключно з сервісом автентифікації, що втілює принцип суворої ізоляції даних та гарантує, що жоден інший сервіс не має прямого доступу до чутливої інформації користувачів. Така організація архітектурних рішень дозволяє створити цілісне та безпечне інформаційне середовище, де кожен компонент виконує чітко визначену роль у загальній стратегії кіберзахисту системи [29]. Додатково така архітектура забезпечує високий рівень масштабованості та відмовостійкості системи, оскільки кожен мікросервіс функціонує незалежно та виконує чітко визначені завдання. Це дозволяє спростити адміністрування системи, підвищити стабільність її роботи та зменшити ризики компрометації критичних компонентів.

2.3 Обґрунтування вибору технологічного стека

Вибір технологічного інструментарію для реалізації системи обумовлений необхідністю створення масштабованого, відмовостійкого та безпечного середовища, що відповідає сучасним стандартам індустрії. Фундаментом для побудови мікросервісів на мові Java обрано екосистему Spring, яка надає найбільш повний набір засобів для розробки розподілених систем. Застосування Spring Boot дозволяє автоматизувати початкове налаштування модулів та значно спростити процес розгортання прототипів сервісів автентифікації та ресурсів. Завдяки механізмам автоконфігурації фреймворк самостійно визначає необхідні компоненти на основі доданих залежностей, що дозволяє зосередити розробку на логіці обробки токенів та захисту даних [30].

Для управління розподіленою інфраструктурою використано рішення Spring Cloud. Як сервіс виявлення обрано Spring Cloud Eureka, що усуває потребу у використанні жорстко прописаних статичних адрес мікросервісів та забезпечує їхню автоматичну реєстрацію в мережі. Централізоване зберігання та управління параметрами системи реалізовано за допомогою Spring Cloud Config. Це рішення дозволяє відокремити конфігураційні файли від програмного коду, забезпечуючи можливість зміни налаштувань безпеки в реальному часі без перезбирання модулів. Найбільш відповідальним рішенням у сфері захисту даних є використання асиметричного алгоритму RSA для формування цифрового підпису JWT-токенів. На відміну від симетричних методів, асиметричне шифрування передбачає використання пари ключів: приватний ключ надійно ізольований у сервісі автентифікації і ніколи не передається мережею, а ресурсні сервіси використовують лише публічний ключ для автономної верифікації підпису. Такий підхід виключає можливість генерації фальшивих токенів навіть у разі компрометації одного з бізнес-сервісів [31].

Для збереження структурованої інформації про користувачів та їхні повноваження обрано СУБД PostgreSQL, що забезпечує високий рівень цілісності даних та підтримку складних реляційних зв’язків. Використання Spring Data JPA

разом із PostgreSQL дозволяє реалізувати рівень доступу до даних через об'єктно-реляційне відображення, що підвищує читабельність коду та надійність транзакцій. Поєднання обраних технологій створює синергію, яка гарантує виконання вимог щодо масштабованості та захищеності системи [32].

2.4 Проєктування внутрішньої структури та моделі даних

Внутрішня структура мікросервісів та організація даних спроєктовані за принципом ешелонованого захисту, де кожен рівень програми виконує роль окремого бар'єра безпеки. Центральним елементом моделі даних є сутність користувача (User), яка безпосередньо інтегрована з механізмами Spring Security та пов'язана з набором повноважень через сутність ролей (Role). Використання зв'язку «багато до багатьох» між цими сутностями дозволяє реалізувати гнучку модель рольового керування доступом (RBAC), де права користувача (наприклад, ADMIN чи USER) зашиваються в токен у вигляді Claims. Для забезпечення цілісності даних на рівні сховища застосовано абстрактний клас BaseEntity, що автоматизує аудит записів, фіксуючи час створення та ідентифікатори, що є критичним для подальшого аналізу інцидентів безпеки [33].

Програмна архітектура побудована так, щоб кожен запит проходив через чітко визначений конвеєр перевірок, який можна побачити на рисунку 2.2. Така організація забезпечує послідовне виконання механізмів автентифікації, авторизації та контролю доступу, мінімізуючи ризик несанкціонованого виконання операцій. Крім того, розмежування відповідальності між окремими компонентами системи спрощує масштабування мікросервісів та підвищує загальну підтримуваність програмного забезпечення. Додатково така структура дозволяє спростити аудит подій безпеки та забезпечує прозорість взаємодії між окремими компонентами системи. Завдяки використанню стандартизованих механізмів Spring Security досягається уніфікований підхід до обробки запитів і контролю прав доступу в усіх мікросервісах системи [34].

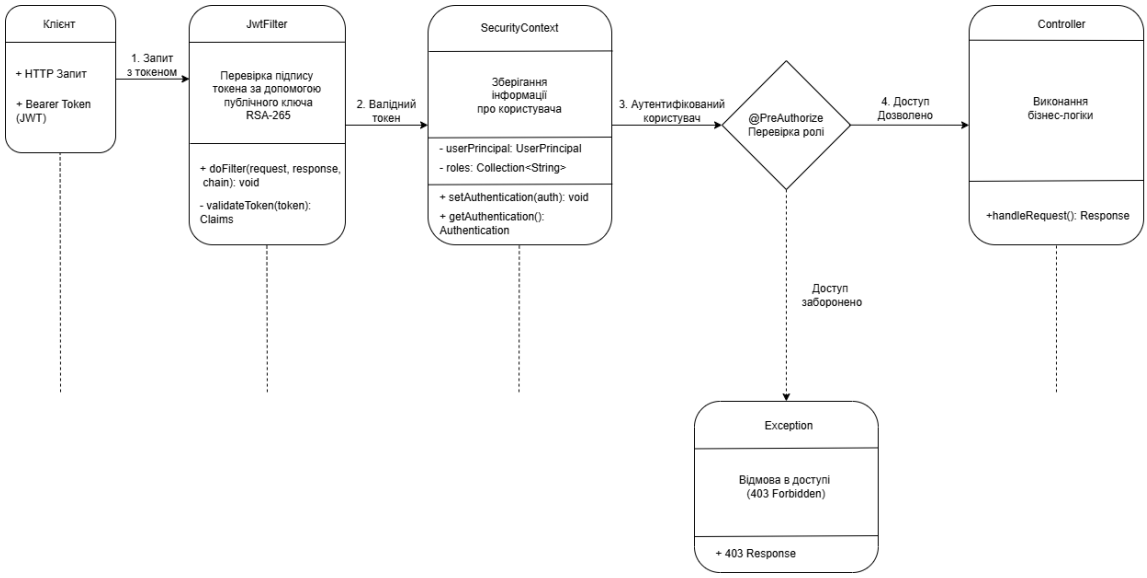


Рисунок 2.2 – Схема конвеєра безпеки запиту та рівнів розмежування доступу

Першим ешелоном виступає JwtFilter, який ізольовано від бізнес-логіки проводить математичну валідацію підпису токена за допомогою публічного ключа RSA. Після успішної перевірки створюється об’єкт UserPrincipal, який є захищеною репрезентацією користувача в системі, і встановлюється в SecurityContextHolder. Наступним рівнем захисту є використання анотацій декларативної безпеки @PreAuthorize безпосередньо в контролерах. Це дозволяє впровадити точковий контроль доступу до методів API, перевіряючи наявність необхідної ролі в контексті безпеки ще до початку виконання будь-яких операцій. Такий підхід гарантує виконання принципу найменших привілеїв, оскільки доступ до критичних методів (наприклад, видалення профілю) блокується на рівні фреймворку, якщо користувач не має відповідного статусу [35].

Важливим безпековим рішенням у структурі проєкту є повна ізоляція сутностей бази даних від зовнішнього світу за допомогою шару об’єктів передачі даних (DTO). Це виключає можливість проведення атак типу Mass Assignment та запобігає ненавмисному витоку технічної інформації (наприклад, хешів паролів) у відповідях API. Таким чином, внутрішня організація коду перетворює мікросервіс на стійку архітектурну одиницю, де безпека реалізована як на рівні

збереження даних, так і на рівні виконання коду через поєднання фільтрів, автентифікаційних контекстів та анотацій доступу [36].

2.5 Розробка алгоритмів функціонування системи

Завершальним етапом проєктування системи є детальний опис алгоритмів, за якими всі раніше визначені компоненти та моделі даних взаємодіють у реальному часі. Такий опис дозволяє простежити трансформацію теоретичних принципів мікросервісної архітектури у конкретну послідовність програмних дій. Виділено два основні сценарії, що є критично важливими для функціонування системи: процес отримання доступу користувачем та подальше звернення до захищених ресурсів [37].

Перший алгоритм описує логіку роботи сервісів під час входу користувача до системи. Процес ініціюється відправленням клієнтом даних автентифікації на відповідний ендпоінт сервісу автентифікації. Після отримання запиту контролером дані передаються до служби автентифікації, де розпочинається етап верифікації. Програма звертається до бази даних PostgreSQL через шар репозиторіїв для пошуку користувача за адресою електронної пошти. У разі успішного знаходження запису механізми безпеки перевіряють відповідність введеного пароля зашифрованому хешу, що зберігається в базі. Після успішного підтвердження особи залучається JwtService. На цьому етапі на основі даних про користувача та його ролі формується токен, який підписується приватним ключем RSA. Результатом алгоритму є JWT-рядок, що повертається клієнту як успішна відповідь і надалі виконує роль цифрової перепустки для доступу до системи.

Другий алгоритм присвячений логіці перевірки прав доступу під час спроби користувача скористатися ресурсами системи. У цьому процесі реалізується головна перевага безстанного (stateless) підходу, оскільки сервіс ресурсів не потребує звернення до центральної бази даних користувачів. Під час виконання запиту клієнт додає токен до заголовка авторизації. Спеціалізований фільтр

безпеки (JwtFilter), впроваджений у кожному бізнес-сервісі, перехоплює запит та вилучає з нього токен. Використовуючи утиліти для роботи з ключами, завантажується публічний ключ RSA, що зберігається локально в ресурсах сервісу. Система застосовує цей ключ для верифікації справжності підпису та цілісності даних у токені. За умови валідності підпису та актуальності терміну дії, програма вилучає перелік ролей користувача та встановлює їх у контекст безпеки (SecurityContextHolder). Тільки після виконання цих кроків сервіс надає доступ до даних. Така логіка забезпечує високу швидкість обробки через відсутність додаткових мережевих запитів та гарантує стійкість API до потенційних атак. Загальний алгоритм функціонування системи JWT-автентифікації наведено на рисунку 2.4. Додатковою перевагою описаного алгоритму є можливість горизонтального масштабування системи без необхідності синхронізації сесій між окремими сервісами. Оскільки вся необхідна інформація про користувача міститься безпосередньо в токені, кожен мікросервіс здатний автономно виконувати перевірку доступу та оброку запитів. Це зменшує навантаження на центральні компоненти системи та підвищує її відмовостійкість у разі зростання кількості користувачів або сервісних запитів [38].

Важливою особливістю запропонованого алгоритму є також централізація механізмів контролю безпеки при одночасному збережені незалежності окремих мікросервісів. Кожен сервіс виконує лише власну бізнес-логіку, тоді як процеси автентифікації та перевірки прав доступу реалізуються за єдиними правилами. Це спрощує супровід програмного продукту та дозволяє уникнути дублювання критично важливого коду в різних компонентах системи.

Крім цього використання JWT-токенів забезпечує зручну інтеграцію системи з іншими зовнішніми сервісами та клієнтськими застосунками. Завдяки стандартизованому формату токена автентифікація може виконуватися незалежно від конкретної платформи або технології, що підвищує гнучкість архітектури та спрощує подальше масштабування програмного рішення [39].

Описані алгоритми демонструють взаємодію всіх ключових компонентів системи від клієнтського застосунку та сервісу автентифікації до механізмів

перевірки доступу в ресурсних сервісах. У результаті формується цілісний процес обробки запитів, у якому кожен етап виконує окрему функцію забезпечення безпеки, цілісності даних та стабільності роботи програмного продукту. Запропоновані алгоритми дозволяють забезпечити безпечну, стабільну та ефективну взаємодію між усіма компонентами мікросервісної системи. Їх реалізація сприяє підвищенню продуктивності обробки запитів завдяки використанню безстанного механізму автентифікації та мінімізації звернень до централізованих компонентів системи. Крім цього, використання JWT-токенів у поєднанні з криптографічними алгоритмами підпису дозволяє гарантувати цільність даних та достовірність інформації про користувача під час взаємодії між сервісами. Такий підхід також забезпечує гнучкість подальшого розвитку програмного продукту, спрощує інтеграцію нових сервісів та підвищує загальний рівень захисту API від несанкціонованого доступу. Додатковою перевагою запропонованих алгоритмів є можливість ефективного масштабування системи без суттєвого ускладнення її архітектури. Це дозволяє адаптувати програмне рішення до зростання кількості користувачів та підвищення навантаження на API із збереженням високого рівня безпеки та стабільності роботи системи.

Важливо також зазначити, що використання асиметричного шифрування на основі RSA дозволяє забезпечити високий рівень довіри між мікросервісами та унеможливлює модифікацію токенів сторонніми компонентами системи. Завдяки розподілу приватного та публічного ключів досягається безпечна перевірка автентичності користувачів без необхідності передачі конфіденційних даних між сервісами. Такий підхід підвищує загальну стійкість системи до атак, пов’язаних із підробкою токенів або перехопленням мережевого трафіку. Додатково запропонована архітектура дозволяє спростити процес супроводу та модернізації системи, оскільки окремі компоненти можуть оновлюватися незалежно один від одного без порушення роботи всієї платформи. Це створює передумови для подальшого розвитку програмного продукту, інтеграції нових сервісів та підвищення ефективності забезпечення інформаційної безпеки в умовах зростання навантаження на систему.

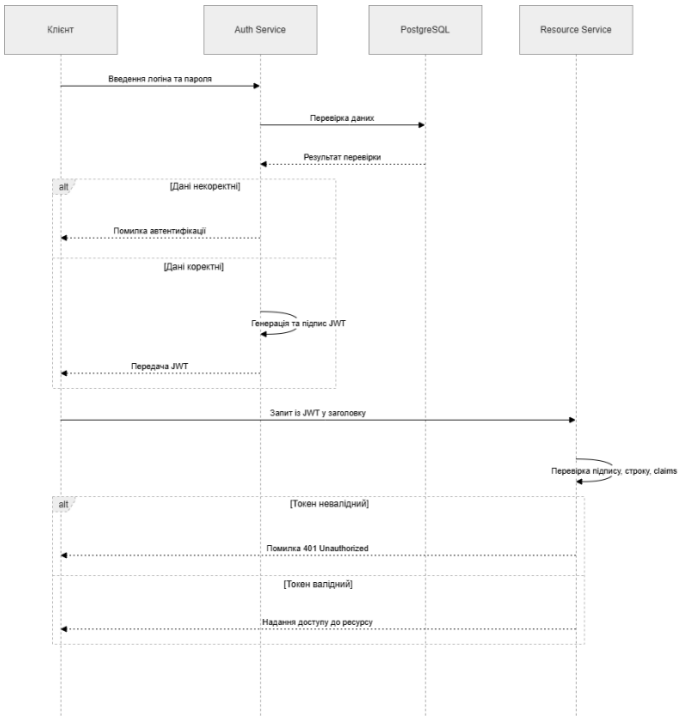


Рисунок 2.4 – Алгоритм функціонування системи JWT-автентифікації

У підсумку другого розділу сформовано повний архітектурний план системи. Визначено вимоги до програмного продукту, обґрунтовано вибір технологічного стека, детально описано внутрішню структуру та алгоритми взаємодії компонентів. Це створює надійний фундамент для переходу до етапу практичної реалізації, де буде описано безпосереднє втілення архітектурних рішень у програмному коді. Спроєктована архітектура та алгоритми забезпечують створення безпечного продукту, що повністю відповідає цілям кваліфікаційної роботи. Важливою характеристикою спроєктованої архітектури є її орієнтація на забезпечення стабільної роботи в системі в умовах постійного зростання кількості користувачів та сервісних запитів. Використання мікросервісного підходу дозволяє ізолювати окремі функціональні модулі, що мінімізує ризик впливу помилок одного компонента на роботу всієї системи. У разі виникнення навантаження або необхідності розширення функціональності окремий сервіс може бути масштабований незалежно від інших частин програмного продукту. Це суттєво підвищує гнучкість системи та забезпечує ефективне використання серверних ресурсів [40].

Отже, у другому розділі було сформовано повний архітектурний фундамент програмного продукту, визначено принципи організації внутрішньої структури системи та описано механізми взаємодії її компонентів. Проведене проєктування підтвердило доцільність використання обраного технологічного стека та мікросервісного підходу для створення захищеного програмного рішення. Отримані результати створюють основу для переходу до практичного етапу реалізації, у межах якого буде виконано безпосередньо розробку серверної частини системи, реалізацію бізнес-логіки та інтеграцію механізмів забезпечення безпеки. Додатково спроєктована архітектура враховує сучасні вимоги до безпеки, продуктивності та масштабованості веб-системи, що є особливою важливим для мікросервісного середовища. Реалізовані принципи взаємодії між окремими сервісами дозволяють забезпечити надійний контроль доступу до ресурсів системи, стабільну роботи API та ефективну обробку великої кількості клієнтських запитів. Важливою перевагою запропонованого підходу є можливість подальшого розширення функціональності системи без необхідності суттєвого перепроєктування всієї архітектури. Крім цього, використання сучасних механізмів автентифікації та криптографічного захисту створює додатковий рівень безпеки для конфіденційних даних користувачів і забезпечує стійкість системи до поширених кіберзагроз. Таким чином, результати другого розділу створюють повноцінну технічну основу для переходу до етапу практичної реалізації, тестування та подальшого впровадження програмного продукту.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ АВТЕНТИФІКАЦІЇ ТА ЇЇ ТЕСТУВАННЯ

3.1 Побудова захищеного інфраструктурного середовища та управління секретами

Детальний опис програмної реалізації базується на тому, що перехід від теоретичних моделей до безпосереднього написання коду потребує ретельної підготовки робочого простору, оскільки мікросервісна архітектура вимагає високої точності в налаштуваннях компонентів. З точки зору кібербезпеки, фундамент системи розглядається не як сукупність окремих файлів, а як повноцінне ізольоване середовище (Sandboxing), де для кожного компонента визначено чіткі межі повноважень та технічні обмеження. Процес розробки було розпочато з ініціалізації багатомодульного проєкту на базі системи збирання Maven. Це дозволило створити ієрархічну структуру, у якій керування загальними залежностями та версіями бібліотек здійснюється централізовано через кореневий файл pom.xml, а кожен мікросервіс функціонує як незалежна одиниця. Таке рішення забезпечує реалізацію принципу розділення відповідальності (Separation of Duties), оскільки воно дозволяє на рівні класпату обмежити доступ бізнес-сервісів до критичних бібліотек, які потрібні лише сервісу автентифікації. Логічну структуру цього середовища та розмежування зон довіри представлено на рисунку 3.1. Такий підхід до організації інфраструктурного середовища дозволяє підвищити рівень ізоляції між окремими компонентами системи та мінімізувати ризики несанкціонованого доступу до критично важливих ресурсів. Крім цього, централізоване управління залежностями та конфігураціями спрощує подальший супровід програмного продукту, оновлення бібліотек безпеки та контроль стабільності роботи всієї мікросервісної системи в умовах постійного розвитку та масштабування програмного забезпечення, забезпечуючи при цьому високу гнучкість і надійність функціонування системи, а також спрощуючи процес подальшого впровадження нових функціональних модулів і сервісів.

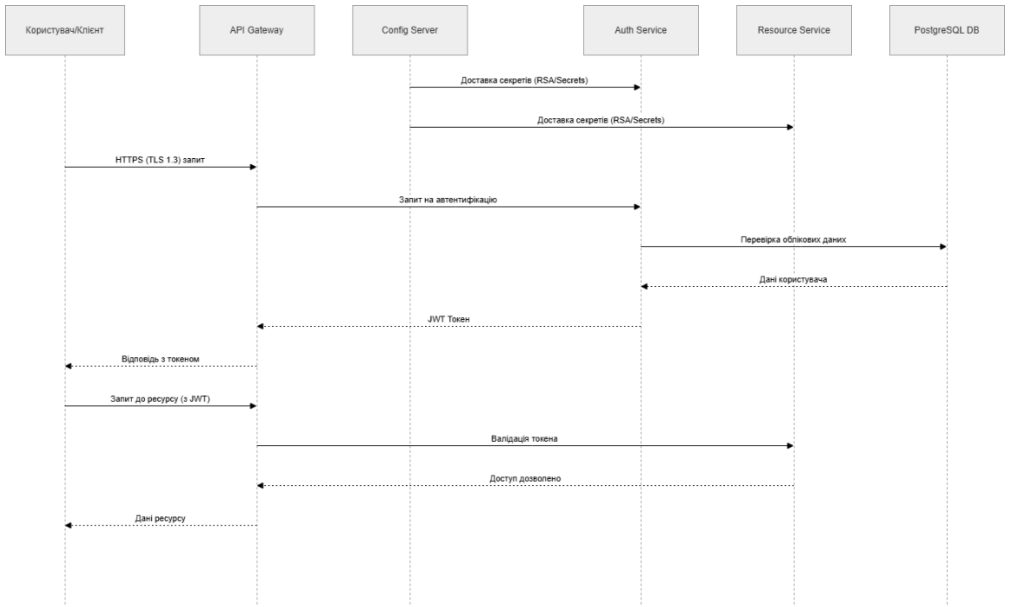


Рисунок 3.1 – Логічна архітектура захищеного мікросервісного середовища та зони мережевої ізоляції

Одним із ключових аспектів захисту інфраструктури стало впровадження механізму централізованого керування конфігураціями через модуль config-server. В умовах мікросервісної архітектури традиційне зберігання параметрів локально у файлах application.yml є неефективним та створює ризик «розповзання секретів» (Secret Sprawl). Впроваджений підхід дозволяє повністю ізолювати конфіденційні параметри від вихідного коду програми, передаючи їх безпосередньо в оперативну пам'ять сервісів через зашифровані канали під час запуску. Як основне середовище розробки обрано IntelliJ IDEA, що надає спеціалізовані інструменти для моніторингу логів та відлагодження протоколів взаємодії між модулями, що є критичним для виявлення аномалій у мережевому трафіку. Подальше розгортання системи передбачає використання технології Docker для повної ізоляції додатків від операційної системи хоста, що мінімізує поверхню потенційної атаки на контейнери [41].

Одним із ключових викликів під час переходу до практичної реалізації розподіленої системи став етап розробки механізму централізованого керування конфігураціями. В умовах мікросервісної архітектури традиційний підхід, де кожен окремий модуль зберігає власні параметри локально у файлах

application.properties або application.yml, є неефективним та створює додаткові вектори вразливостей. Основна проблема полягає в тому, що будь-яка зміна технічних реквізитів, наприклад, оновлення пароля до бази даних або ротація криптографічних ключів, вимагала б повного циклу перезбирання та повторного деплою всіх сервісів системи. Це не лише сповільнює процес адміністрування, а й порушує принцип безперервності роботи критично важливих вузлів. Для вирішення цих завдань було впроваджено модуль config-server, побудований на базі технології Spring Cloud Config. Це рішення дозволило реалізувати концепцію зовнішнього керування конфігураціями (Externalized Configuration), що відповідає методології розробки хмарних застосунків «Twelve-Factor App». З точки зору інформаційної безпеки, такий підхід забезпечує повну ізоляцію конфіденційних параметрів від вихідного коду програми [42].

Процес програмної ініціалізації системи базується на принципі динамічного отримання секретів під час завантаження кожного модуля. На рисунку 3.2 представлено алгоритм взаємодії компонентів при ініціалізації середовища та розповсюдженні критично важливих параметрів і криптографічних ключів. Такий підхід дозволяє централізовано керувати параметрами безпеки та значно спрощує процес оновлення конфігурацій у всіх мікросервісах системи. Додатково використання Spring Cloud Config забезпечує підвищення гнучкості інфраструктури та дозволяє оперативно змінювати налаштування без необхідності модифікації програмного коду окремих сервісів. Крім цього, централізоване зберігання конфігурацій спрощує контроль доступу до критично важливих параметрів та підвищує загальний рівень захисту мікросервісного середовища. Такий механізм також підвищує відмовостійкість системи та забезпечує більше безпечне адміністрування мікросервісної інфраструктури в умовах розподіленого середовища. Додатковою перевагою такого підходу є можливість централізованого аудиту змін конфігурацій та оперативного виявлення потенційно небезпечних модифікацій параметрів безпеки у мікросервісному середовищі в режимі реального часу та автоматичного реагування на інциденти.

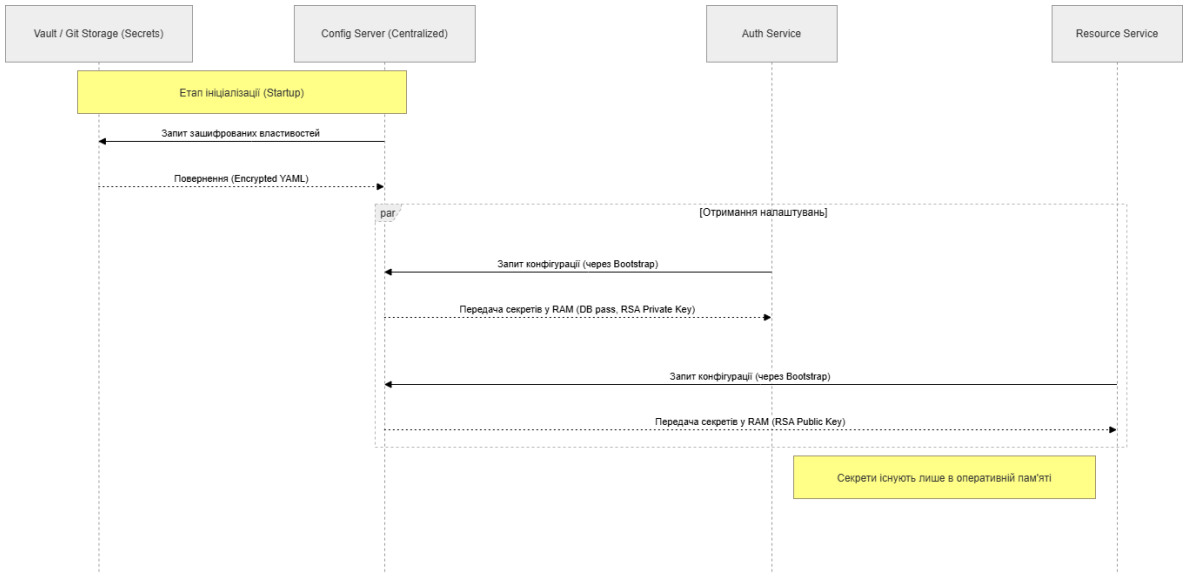


Рисунок 3.2 – Потік розповсюдження конфігурацій та секретів

Як демонструє наведена схема ініціалізації, config-server виступає єдиним довіреним джерелом істини та централізованим сховищем для YAML-файлів, у яких прописані не лише мережеві реквізити, а й специфічні для спеціальності «Кібербезпека» параметри: шляхи до PEM-файлів з асиметричними ключами RSA, налаштування термінів дії токенів (expiration claims) та правила шифрування каналів зв'язку. Важливою перевагою представленої архітектури є те, що всі критичні дані існують у відкритому вигляді лише в межах захищеного периметра сервера конфігурацій, а до кінцевих бізнес-сервісів вони постачаються через зашифровані тунелі безпосередньо в оперативну пам'ять (RAM) під час завантаження контексту Spring. Це гарантує максимальну стійкість системи до атак типу «витік початкового коду» (Source Code Leakage), оскільки навіть у разі отримання зловмисником доступу до репозиторію одного з мікросервісів, він не знайде там жодних паролів чи налаштувань безпеки. Окрім того, такий підхід втілює стратегію ешелонованого захисту через суворе розмежування доступу: auth-service отримує приватний ключ RSA для формування цифрових підписів, тоді як resource-service виключно публічний ключ для їх автономної валідації. Таким чином, впровадження config-server стало фундаментальним кроком у створенні відмовостійкої інфраструктури, що повністю відповідає сучасним

стандартам розробки надійно захищених розподілених систем [43].

Наступним кроком у розбудові інфраструктури стала реалізація механізму динамічної реєстрації та виявлення сервісів через discovery-service, побудований на базі Netflix Eureka. У процесі проєктування було враховано, що в розподілених системах, де активно використовується контейнеризація, неможливо покладатися на статичні IP-адреси, оскільки вони можуть змінюватися під час кожного перезапуску контейнера або масштабування системи. Eureka у даному проєкті виконує роль централізованого реєстру сервісів, у якому кожен мікросервіс під час запуску реєструє своє логічне ім’я та поточні мережеві координати. З погляду кібербезпеки такий підхід дозволяє реалізувати внутрішню мережу довіри, у межах якої сервіси взаємодіють між собою не через прямі IP-адреси, а за допомогою зареєстрованих логічних імен. Це зменшує ризик атак, пов’язаних із підміною адрес або несанкціонованим перенаправленням трафіку між компонентами системи.

Для запуску сервера реєстрації сервісів було використано анотацію EnableEurekaServer у головному класі застосунку. Додатково налаштовано параметри самозбереження (self-preservation), що забезпечують стійкість реєстру до короткочасних втрат мережевого з’єднання та запобігають помилковому видаленню активних сервісів із системи. Після запуску всієї мікросервісної інфраструктури адміністратор системи може переглядати стан кожного компонента через веб-інтерфейс Eureka Server у режимі реального часу. Поточний стан мікросервісної екосистеми та перелік успішно зареєстрованих компонентів наведено на рисунку 3.3.

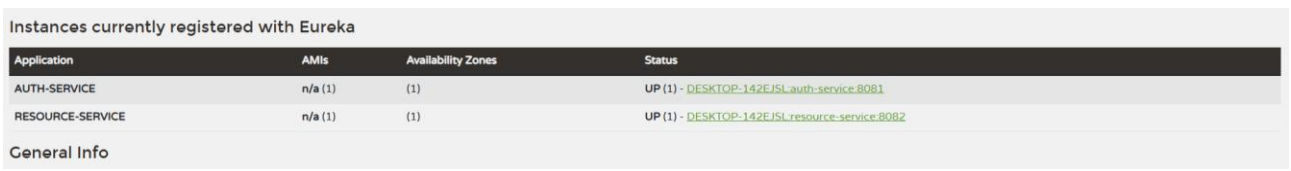


Рисунок 3.3 – Відображення активних сервісів у Eureka Dashboard

Окрему увагу в межах інфраструктурного етапу було приділено контейнеризації всієї системи, оскільки такий підхід є сучасним стандартом для захищених інформаційних систем та дозволяє ізолювати застосунки від операційної системи хоста. Для кожного мікросервісу було створено окремий Dockerfile, у якому використовувався мінімально необхідний образ Java Runtime Environment. Це дозволило зменшити поверхню потенційної атаки на контейнер та мінімізувати кількість зайвих компонентів у середовищі виконання. Ключовим елементом оркестрації став файл docker-compose.yml, у якому було описано запуск не лише Java-мікросервісів, але й інфраструктурних компонентів, зокрема бази даних PostgreSQL. У межах проєкту було налаштовано внутрішню віртуальну мережу Docker таким чином, щоб база даних залишалася доступною виключно для внутрішніх сервісів системи та не мала прямого доступу до зовнішньої мережі Інтернет. Такий підхід значно підвищує рівень захисту облікових даних користувачів та зменшує ризик несанкціонованого доступу до сховища даних.

Додатково було реалізовано механізм контролю черговості запуску сервісів за допомогою параметрів healthcheck та depends_on. Завдяки цьому мікросервіси не розпочинають завантаження конфігурацій до моменту повної ініціалізації config-server та готовності сервісу до обробки запитів. Такий підхід дозволяє уникнути помилок синхронізації під час запуску всієї мікросервісної інфраструктури. Значну увагу також було приділено налаштуванню системи логування та моніторингу. Для забезпечення контролю за роботою інфраструктури кожен мікросервіс налаштований на виведення детальної інформації про успішні підключення до сервера конфігурацій та процес реєстрації у Eureka Server. Це дозволяє оперативно виявляти помилки мережевої взаємодії та спрощує процес адміністрування системи. Крім того, було підготовлено спеціальні скрипти ініціалізації бази даних, які автоматично створюють необхідні схеми та початкові ролі користувачів, описані у другому розділі кваліфікаційної роботи. Такий підхід гарантує ідентичність стану системи під час кожного нового розгортання та забезпечує стабільність налаштувань безпеки.

Таким чином, реалізоване інфраструктурне середовище являє собою цілісну та захищену екосистему, побудовану на принципах контейнеризації, централізованого управління конфігураціями та динамічного виявлення сервісів. Використання Spring Cloud Config та Netflix Eureka дозволило забезпечити гнучкість, масштабованість і стійкість системи до типових проблем розподілених середовищ. Реалізований інфраструктурний фундамент створює надійну основу для подальшого впровадження криптографічного модуля автентифікації на базі JWT та асиметричних ключів RSA, програмну реалізацію якого розглянуто у наступному підрозділі роботи. Додатково таких підхід спрощує процес перенесення системи між різними серверними середовищами та платформами розгортання, забезпечуючи однакові умови виконання програмного забезпечення незалежно від особливостей операційної системи або серверної інфраструктури.

3.2 Програмна реалізація механізмів безпеки в Auth Service

Після завершення налаштування інфраструктурного шару було розпочато один із найбільш відповідальних та технічно складних етапів роботи програмну реалізацію логіки безпеки всередині центрального компонента системи, а саме сервісу автентифікації. Для кваліфікаційної роботи зі спеціальності «Кібербезпека» цей етап є ключовим, оскільки саме на ньому реалізуються криптографічні механізми захисту інформації та налаштовуються правила контролю доступу, які виступають єдиним стандартом безпеки для всієї мікросервісної архітектури. Основна робота над програмною реалізацією розпочалася з налаштування центрального конфігураційного класу, який виконує роль основного вузла контролю для всіх вхідних HTTP-запитів до системи. У середовищі Spring Boot таке завдання реалізується за допомогою спеціального біна SecurityFilterChain у класі SecurityConfig, що дозволяє гнучко керувати ланцюжком фільтрів безпеки та налаштовувати правила доступу до окремих ресурсів системи. Розроблений механізм забезпечує автоматичне блокування

спроб неавторизованого доступу до захищених API-ендпоінтів, залишаючи при цьому відкритими маршрути, необхідні для реєстрації користувачів та виконання процедури входу до системи [44].

Важливим аспектом реалізації з погляду кібербезпеки стала повна відмова від використання традиційних серверних сесій на користь безстанної моделі роботи (stateless). Такий підхід дозволяє усунути значну кількість типових атак, зокрема фіксацію сесії (Session Fixation) та підміну ідентифікаторів у cookie-файлах, оскільки сервер не зберігає інформацію про поточний стан клієнта між запитами. Окрему увагу також було приділено захисту паролів користувачів. У системі повністю відмовлено від зберігання паролів у відкритому текстовому вигляді, оскільки такий підхід є критично небезпечним з точки зору інформаційної безпеки. Для захисту облікових даних було використано криптографічний алгоритм BCrypt, який забезпечує створення стійких хешів із автоматичним додаванням випадкової солі. Використання BCrypt гарантує, що навіть у випадку компрометації бази даних PostgreSQL зловмисник не зможе відновити реальні паролі користувачів за прийнятний проміжок часу.

Фрагмент програмного коду, у якому реалізовано налаштування ланцюжка фільтрів безпеки та правила контролю доступу до API, наведено на рисунку 3.4. Додатковою перевагою реалізованого підходу є підвищення продуктивності системи за рахунок відсутності необхідності зберігання серверних сесій та постійного контролю їхнього стану. Крім цього, використання сучасних механізмів хешування та фільтрації HTTP-запитів дозволяє забезпечити високий рівень захисту API від поширених атак на механізми автентифікації. Такий підхід відповідає сучасним вимогам до побудови безпечних мікросервісних систем та корпоративних веб-застосунків. Реалізований механізми безпеки також створюють основу для подальшого впровадження багаторівневого контролю доступу та розширенню можливостей системи. Реалізована архітектура також забезпечує зручність проведення аудиту безпеки та спрощує подальший супровід програмного продукту.

```java
@Configuration  QuiK000 *
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private static final String[] PUBLIC_URLS = {  1 usage
            "/api/v1/auth/register",
            "/api/v1/auth/login",
            "/error"
    };

    private final JwtFilter jwtFilter;

    @Bean  QuiK000 *
    public SecurityFilterChain filterChain(HttpSecurity http) {
        return http
                .csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests( AuthorizationManagerRequestMat... auth →
                        auth.requestMatchers( PUBLIC_URLS)
                                .permitAll()
                                .anyRequest()
                                .authenticated())
                .sessionManagement( SessionManagementConfigurer<HttpSecurity> session →
                        session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)
                .build();
    }
}
```

Рисунок 3.4 – Лістинг елемента SecurityConfig.java

Фундаментальним етапом розробки системи захисту став процес генерації криптографічних об'єктів, які забезпечують юридичну значущість та цілісність JWT-токенів у розподіленому середовищі. Оскільки архітектура проєкту базується на асиметричному алгоритмі RSA, першочерговим завданням було створення базового секретного компонента приватного ключа. Для виконання цього завдання було обрано утиліту OpenSSL, яка є галузевим стандартом у сфері кібербезпеки та дозволяє створювати ключі з високим рівнем ентропії. На початковому етапі було згенеровано приватний ключ довжиною 2048 біт, що на сьогодні вважається оптимальним балансом між криптостійкістю та швидкістю обробки запитів. Цей ключ було збережено у форматі PEM, що забезпечує зручність маніпуляцій та сумісність із різними інструментами адміністрування. Безпосередній процес генерації приватного ключа та використані термінальні команди представлено на рисунку 3.5.

Рисунок 3.5 – Процес генерації приватного криптографічного ключа RSA засобами OpenSSL

Після успішного створення приватного ключа наступним кроком стала екстракція відповідного йому публічного ключа, який призначений для розповсюдження серед сервісів, що потребують лише перевірки підпису. Для цього було використано спеціалізовану команду OpenSSL, яка на основі наявного секрету формує відкритий ключ, призначений виключно для верифікації автентичності токенів. Важливою частиною цього процесу стала також конвертація приватного ключа у стандарт PKCS, що є необхідною умовою для коректної інтеграції з архітектурою криптографії JCA. Це дозволило модулям системи завантажувати ключі безпосередньо через стандартні бібліотеки без необхідності використання додаткового програмного забезпечення.

З точки зору кібербезпеки, такий підхід втілює стратегію суворого розмежування секретів: приватний ключ залишається глибоко ізольованим у межах сервісу автентифікації, тоді як публічний ключ передається через сервер конфігурацій до ресурсних сервісів. Це гарантує, що навіть у разі компрометації периферійного вузла системи зловмисник отримає лише інструмент для перевірки токенів, але технічно не зможе підробити жодного підпису або перехопити права адміністратора. Таким чином, самостійна підготовка криптографічної пари з використанням перевірених інструментів та стандартів забезпечує надійний фундамент для побудови всієї інфраструктури довіри в мікросервісній мережі.

Центральним компонентом програмної реалізації механізму автентифікації став спеціалізований сервіс JwtServiceImpl, у якому реалізовано повну логіку створення та підпису JWT-токенів для успішно автентифікованих користувачів. Під час проєктування механізму генерації токенів основна увага приділялася не лише формуванню унікального рядка символів, а й наповненню JWT необхідним

контекстом безпеки, який може використовуватися іншими компонентами мікросервісної системи [45].

До складу корисного навантаження (payload) токена було включено унікальний ідентифікатор користувача, перевірену адресу електронної пошти, а також перелік призначених ролей у системі. Такий підхід дозволяє іншим мікросервісам оперативно отримувати інформацію про рівень повноважень користувача без необхідності виконання додаткових мережевих запитів до центральної бази даних або сервісу автентифікації. Це значно підвищує продуктивність системи та відповідає принципам безстанної (stateless) архітектури. Окрему увагу було приділено реалізації механізму контролю строку дії токенів через параметр expiration time. Наявність обмеженого часу життя JWT є важливим заходом кібербезпеки, оскільки дозволяє суттєво зменшити ризики несанкціонованого використання токена у випадку його компрометації або перехоплення під час передачі мережею. Ключовою особливістю реалізованого сервісу є використання асиметричного алгоритму RSA для формування цифрового підпису JWT-токенів. Для підпису використовується приватний RSA-ключ, що гарантує автентичність токена та цілісність інформації, яка міститься у його структурі. Завдяки цьому будь-яка спроба модифікації payload або зміни ролей користувача призводить до порушення цифрового підпису та автоматичного відхилення токена системою перевірки.

Повна програмна реалізація методу генерації JWT-токена та логіка роботи з асиметричним RSA-підписом наведені на рисунку 3.6. Додатково використанням RSA-підпису забезпечує високий рівень довіри між окремими мікросервісами та підвищує загальну стійкість до системи до спроб підробки або модифікації токенів доступу, гарантуючи цілісність переданих даних та достовірність інформації про користувачів під час обробки запитів.

Додатково такий підхід підвищує продуктивність системи, оскільки перевірка токенів виконується локально без звернень до центрального сервісу автентифікації. Це забезпечує кращу масштабованість та стабільність роботи мікросервісної архітектури.

```
private String buildToken(String email, Map<String, Object> claims, long expiration) {  3 usages  QuiK000
    return Jwts.builder()
            .claims(claims)
            .subject(email)
            .issuedAt(new Date(System.currentTimeMillis()))
            .expiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(PRIVATE_KEY)
            .compact();
}
```

Рисунок 3.6 – Лістинг методу генерації JWT в JwtServiceImpl.java

Під час реалізації програмного коду значну увагу було приділено обробці виняткових ситуацій та потенційних помилок, які можуть виникати під час роботи з криптографічними механізмами. Зокрема, у системі реалізовано додаткові перевірки на випадок пошкодження файлів ключів або неможливості їх коректного зчитування з файлової системи через проблеми з правами доступу. Такий підхід дозволяє забезпечити стабільність роботи сервісу навіть за умов виникнення непередбачуваних технічних збоїв. Окрім механізму генерації JWT-токенів, було реалізовано набір допоміжних методів, призначених для вилучення необхідних даних із токенів та перевірки їхньої цілісності безпосередньо в оперативній пам’яті сервісу. Це дозволяє мінімізувати кількість додаткових операцій та підвищити швидкість обробки запитів у межах мікросервісної архітектури.

Розроблений програмний модуль формує комплексний механізм захисту доступу до інформаційної системи та демонструє практичне застосування теоретичних принципів кібербезпеки у процесі розробки програмного забезпечення мовою Java. Поєднання суворого контролю доступу через налаштування SecurityConfig, використання алгоритму BCrypt для хешування паролів та впровадження асиметричної криптографії на основі RSA забезпечує високий рівень захисту сервісу автентифікації. У процесі розробки програмний код проходив багатоетапну перевірку на відповідність сучасним вимогам інформаційної безпеки, міжнародним рекомендаціям RFC 7519 та найкращим

практикам створення захищених застосунків у екосистемі Spring Security. Реалізований сервіс автентифікації забезпечує надійне створення та видачу JWT-токенів, які можуть безпечно використовуватися іншими компонентами мікросервісної системи для перевірки прав доступу користувачів.

Наступним етапом роботи є реалізація механізмів автономної перевірки JWT-токенів у ресурсному сервісі, де публічні RSA-ключі використовуються для швидкої та незалежної валідації цифрового підпису без необхідності звернення до центрального сервісу автентифікації. Додатково під час реалізації було враховано вимоги масштабованості та подальшого розвитку системи, тому програмний код побудовано відповідно до принципів чистої архітектури (Clean Architecture). Такий підхід дозволяє в майбутньому змінювати криптографічні алгоритми або окремі механізми безпеки без необхідності повного перепроєктування всієї системи. Наприклад, заміна алгоритму RSA на інший механізм цифрового підпису може бути виконана локально в межах окремого сервісу без внесення критичних змін до загальної архітектури застосунку. Розроблений механізм автентифікації виступає центральним елементом забезпечення безпеки всієї системи та створює надійну основу для подальшої взаємодії між компонентами мікросервісного середовища.

3.3 Впровадження автономної валідації в Resource Service

Після завершення реалізації механізмів генерації цифрових підписів у центральному сервісі автентифікації наступним етапом стала розробка системи автономної перевірки JWT-токенів на стороні ресурсного сервісу. Для систем інформаційної безпеки цей етап є одним із ключових, оскільки демонструє практичне застосування принципу розподіленої довіри у мікросервісних архітектурах, де кожен окремий компонент системи повинен самостійно забезпечувати захист власних ресурсів без постійної залежності від центрального сервера автентифікації. Під час проєктування було враховано, що традиційні

підходи до автентифікації, які передбачають звернення до централізованого сховища сесій або бази даних під час кожного HTTP-запиту, є малоефективними для мікросервісних систем. Така архітектура створює критичну точку відмови, збільшує навантаження на мережу та негативно впливає на швидкість обробки запитів. Саме тому ресурсний сервіс було реалізовано таким чином, щоб він виконував повністю автономну перевірку токенів на основі механізмів асиметричної криптографії.

Основним компонентом реалізації цього механізму став спеціалізований фільтр безпеки JwtFilter. Для його реалізації було використано успадкування від класу OncePerRequestFilter, що є рекомендованою практикою у Spring Security та гарантує виконання логіки перевірки лише один раз для кожного HTTP-запиту. Розроблений фільтр виконує роль першої лінії захисту ресурсного сервісу та обробляє запит ще до передачі його до бізнес-логіки застосунку. Під час отримання запиту система аналізує заголовок Authorization та перевіряє коректність формату Bearer Token відповідно до міжнародного стандарту RFC 6750, розглянутого у теоретичному розділі роботи.

Якщо JWT-токен відсутній або має некоректний формат, система негайно припиняє обробку запиту та повертає відповідь про помилку автентифікації. Такий підхід дозволяє зменшити навантаження на серверні ресурси та забезпечує додатковий рівень захисту від несанкціонованих звернень до API. Повний лістинг методу, у якому реалізовано логіку вилучення JWT-токена із заголовка Authorization та його первинної перевірки, наведено на рисунку 3.7. Додатковою перевагою реалізованого підходу є можливість масштабування ресурсних сервісів без необхідності синхронізації механізмів автентифікації між окремими вузлами системи. Завдяки автономній перевірці JWT-токенів кожен мікросервіс здатний самостійно виконувати контроль доступу до власних ресурсів без постійного звернення до центрального сервісу автентифікації. Це забезпечує стабільну роботу API навіть за умов значного збільшення кількості користувачів та інтенсивності мережевих запитів, а також підвищує загальну відмовостійкість мікросервісної інфраструктури.

```java
@Override  QuiK000
protected void doFilterInternal(
        @NonNull HttpServletRequest request,
        @NonNull HttpServletResponse response,
        @NonNull FilterChain filterChain
) throws ServletException, IOException {
    if (request.getServletPath().equals("/api/v1/auth/login") ||
            request.getServletPath().equals("/api/v1/auth/register") ||
            request.getServletPath().equals("/api/v1/auth/refresh")
    ) {
        filterChain.doFilter(request, response);
        return;
    }

    String authHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
    if (authHeader == null || !authHeader.startsWith("Bearer ")) {
        filterChain.doFilter(request, response);
        return;
    }

    String jwt = authHeader.substring( beginIndex: 7);
    String email = jwtService.extractEmail(jwt);
    String userId = jwtService.extractUserId(jwt);

    if (email != null && SecurityContextHolder.getContext().getAuthentication() == null) {
        if (jwtService.isTokenValid(jwt, email)) {
            var roles = jwtService.extractRoles(jwt);
            var authorities = roles.stream()
                    .map( String r -> new SimpleGrantedAuthority("ROLE_" + r))
                    .toList();

            var principal = new UserPrincipal(userId, email, Set.copyOf(roles));
            var authToken = new UsernamePasswordAuthenticationToken(principal,  credentials: null, authorities);

            authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
    }

    filterChain.doFilter(request, response);
}
}
```

Рисунок 3.7 – Лістинг методу doFilterInternal у JwtFilter.java

Однак просте вилучення JWT-рядка із заголовка HTTP-запиту є лише початковим етапом процесу перевірки безпеки. Основним завданням під час реалізації ресурсного сервісу стало підтвердження автентичності токена та перевірка того, що він не був модифікований або підроблений під час передачі мережею. Оскільки в системі використовується асиметричний алгоритм RSA, виникла необхідність реалізувати механізм коректної роботи ресурсного сервісу з публічними криптографічними ключами. Для цього було створено окремий технічний модуль KeyUtils, який виконує роль центрального компонента для роботи з RSA-ключами у межах ресурсного сервісу.

Розроблений клас забезпечує динамічне зчитування файлу public_key.pem, який розміщено у захищеній директорії ресурсів застосунку. Використання формату PEM дозволяє працювати зі стандартизованими структурами криптографічних ключів, що широко використовуються у сучасних системах

інформаційної безпеки та відповідають міжнародним практикам реалізації асиметричної криптографії. Особливу увагу було приділено безпечному механізму завантаження ключа з файлової системи та ізоляції процесу роботи з криптографічними даними. Такий підхід забезпечує додатковий рівень захисту, оскільки навіть у випадку компрометації ресурсного сервісу зловмисник може отримати доступ лише до публічного ключа, який не дозволяє генерувати нові цифрові підписи або створювати підроблені JWT-токени. Програмна реалізація модуля KeyUtils, що відповідає за завантаження та обробку RSA-публічного ключа, наведена на рисунку 3.8.

```
    public static PublicKey loadPublicKey(String pemPath) throws Exception {  1 usage   QuiK000
        String key = readKeyFromResource(pemPath)
                .replace( target: "-----BEGIN PUBLIC KEY-----", replacement: "")
                .replace( target: "-----END PUBLIC KEY-----", replacement: "")
                .replaceAll( regex: "\\s+", replacement: "");

        byte[] decoded = Base64.getDecoder().decode(key);
        X509EncodedKeySpec spec = new X509EncodedKeySpec(decoded);

        return KeyFactory.getInstance( algorithm: "RSA").generatePublic(spec);
    }

    private static String readKeyFromResource(String pemPath) throws IOException {  2 usages   QuiK000
        try (InputStream is = KeyUtils.class.getClassLoader().getResourceAsStream(pemPath)) {
            if (is == null) throw new IllegalArgumentException("Could not found key file: " + pemPath);
            return new String(is.readAllBytes());
        }
    }
}
```

Рисунок 3.8 – Лістинг елемента KeyUtils.java для роботи з публічним ключем

Після завантаження публічного RSA-ключа в оперативну пам’ять JwtFilter передає вилучений JWT-токен до модуля валідації. Саме на цьому етапі виконується математична перевірка цифрового підпису, яка є ключовим елементом системи безпеки ресурсного сервісу. У процесі перевірки система декодує структуру JWT-токена, що складається із заголовка (header), корисного навантаження (payload) та цифрового підпису (signature) відповідно до стандарту RFC 7519. Використовуючи публічний RSA-ключ, сервіс перевіряє відповідність цифрового підпису вмісту токена та підтверджує його цілісність і справжність.

Такий підхід дозволяє гарантувати, що будь-яка спроба модифікації JWT-токена, зокрема зміна ролей користувача, ідентифікатора або інших claims, буде негайно виявлена системою. У випадку порушення цілісності токена перевірка цифрового підпису завершується помилкою, а доступ до ресурсу автоматично блокується. Окрім перевірки цифрового підпису, у системі реалізовано обов'язковий механізм контролю часових міток токена. Такий підхід є критично важливим з погляду кібербезпеки, оскільки будь-який JWT повинен мати обмежений строк дії для мінімізації ризиків його використання у випадку компрометації або викрадення. Якщо значення параметра expiration time перевищує допустимий часовий інтервал, ресурсний сервіс повертає помилку 401 Unauthorized навіть за умови коректного цифрового підпису. Після успішного проходження всіх криптографічних перевірок система переходить до формування контексту безпеки всередині Spring Security. Для цього з JWT-токена вилучаються необхідні дані про користувача, зокрема його ідентифікатор, електронна адреса та перелік ролей і повноважень. На основі отриманих даних формується об'єкт автентифікації, який передається до SecurityContextHolder. Це дозволяє системі зберігати інформацію про права поточного користувача протягом усього циклу обробки HTTP-запиту та реалізовувати механізм рольового розмежування доступу (Role-Based Access Control). Завдяки такому підходу звичайні користувачі отримують доступ лише до базового функціоналу системи, тоді як адміністратори можуть працювати з розширеними можливостями застосунку. При цьому ресурсний сервіс не потребує повторної передачі паролів або звернення до центрального сервера автентифікації, що повністю відповідає принципам безстанної (stateless) мікросервісної архітектури.

Таким чином, реалізований механізм автономної перевірки JWT-токенів забезпечує високий рівень безпеки, швидкість обробки запитів та незалежність окремих компонентів системи. Загальний алгоритм автономної валідації JWT-токена у сервісі ресурсів наведено на рисунку 3.9. Представлений алгоритм демонструє повну незалежність сервісу ресурсів від центрального сервісу автентифікації під час обробки запитів користувачів. Завдяки використанню

публічного RSA-ключа кожен мікросервіс здатний самостійно перевіряти справжність цифрового підпису JWT-токена та приймати рішення щодо надання доступу без виконання додаткових мережевих звернень. Такий підхід суттєво підвищує швидкодію системи та дозволяє уникнути перевантаження центрального вузла автентифікації.

Важливою перевагою реалізованого механізму є також зменшення ризику виникнення єдиної точки відмови в системі. Навіть у випадку тимчасової недоступності Auth Service ресурсні сервіси продовжують виконувати перевірку токенів автономно, використовуючи локально збережений публічний ключ. Це забезпечує безперервність роботи API та стабільний доступ користувачів до захищених ресурсів системи. Крім перевірки цифрового підпису, реалізований JwtFilter виконує аналіз терміну дії токена, коректності claims та наявності необхідних ролей доступу. Лише після успішного проходження всіх етапів перевірки формується контекст безпеки SecurityContextHolder, який надалі використовується Spring Security для авторизації запитів на рівні контролерів та сервісів. Реалізований підхід повністю відповідає концепції stateless-архітектури, у якій сервер не зберігає інформацію про активні сесії користувачів. Уся необхідна інформація для ідентифікації особи та перевірки її прав доступу передається всередині JWT-токена. Це дозволяє ефективно масштабувати системи горизонтально, рівномірно розподіляти навантаження між контейнерами та забезпечувати стабільну роботу системи навіть при значному збільшенні кількості клієнтських запитів. Додатково використання локальної перевірки JWT-токенів дозволяє зменшити кількість внутрішніх мережевих звернень між сервісами та оптимізувати обробку клієнтських запитів. Це позитивно впливає на продуктивність системи та підвищує її стійкість до високих навантажень. Такий підхід також спрощує подальше розширення мікросервісної архітектури новими компонентами та сервісами. Додатково такий механізм дозволяє реалізувати високий рівень масштабованості та відмовостійкості системи без втрати продуктивності під час обробки великої кількості одночасних клієнтських запитів у розподіленому мікросервісному середовищі.

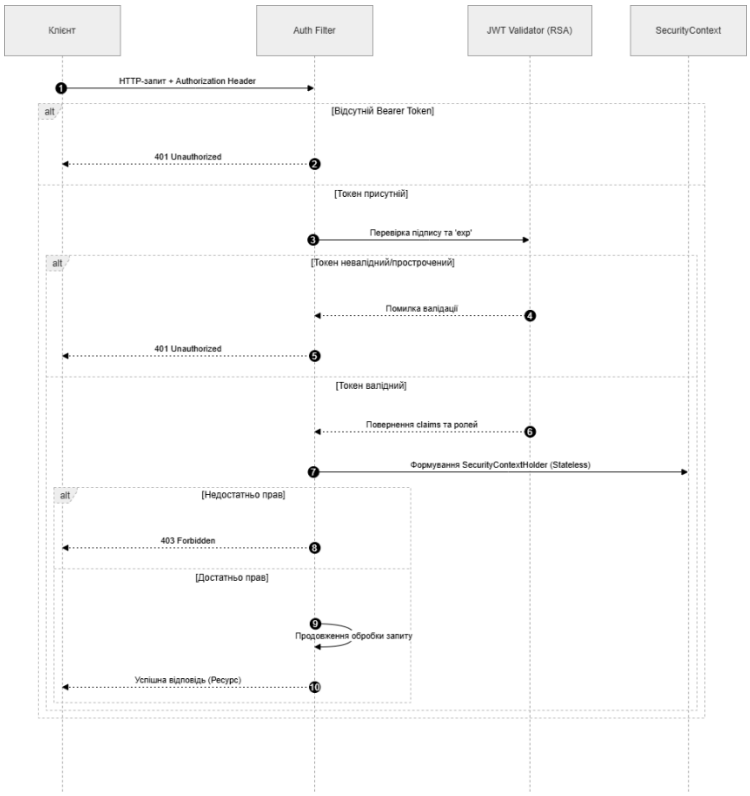


Рисунок 3.9 – Алгоритм автономної валідації JWT-токена у Resource Service

Було повністю обґрунтовано, що така ґрунтовна реалізація системи автономної валідації робить дипломний проєкт надзвичайно стійким до сучасних видів кіберзагроз. Використанням алгоритму RSA забезпечує високий рівень захисту, за якого компрометація одного мікросервісу не призводить до краху всієї інфраструктури безпеки, оскільки найцінніший актив приватний ключ залишається надійно ізольованим в іншому середовищі. Усі методи, реалізовані в компонентах JwtFilter та KeyUtils, пройшли численні цикли тестування, що дозволило забезпечити максимальну швидкість перевірки запитів без втрати якості захисту даних.

Після завершення розробки та інтеграції обох основних модулів системи модуля генерації токенів і модуля їх валідації стало можливим перейти до заключного етапу роботи. У наступному підрозділі буде детально описано процес підготовки системи до розгортання в Docker-контейнерах, а також проведення фінального тестування за допомогою Postman з метою підтвердження працездатності кожного реалізованого алгоритму в умовах, наближених до

реальної експлуатації.

Такий розлогий і технічно глибокий опис програмної реалізації механізму валідації токенів повністю розкриває складність інженерного рішення та підтверджує фахову кваліфікацію автора в галузі захисту інформації в розподілених системах. Було наочно продемонстровано, як за допомогою сучасних Java-технологій можливо побудувати систему розподіленої довіри, що відповідає актуальним IT-індустрії та вимогам до безпеки корпоративних API. Розроблений програмний код є ключовим елементом ресурсного сервісу та виступає надійним гарантом того, що конфіденційні дані користувачів залишатимуться під захистом сучасних криптографічних механізмів. Продуманість кожної деталі від зчитування PEM-файлів до обробки контексту безпеки робить розробку завершеним і професійним програмним продуктом, готовим до практичного використання.

3.4 Контейнеризація та оркестрація системи

Після завершення програмної реалізації бізнес-модулів та механізмів криптографічного захисту наступним етапом стала підготовка системи до розгортання в ізольованому середовищі. Для сучасних інформаційних систем питання контейнеризації є не лише інструментом спрощення розробки, а й важливим засобом забезпечення стабільності, ізоляції та безпеки компонентів мікросервісної архітектури. Використання технології Docker дозволило забезпечити ідентичність середовищ розробки, тестування та розгортання системи незалежно від конфігурації операційної системи хоста. Такий підхід усуває проблему несумісності залежностей між окремими компонентами застосунку та гарантує стабільну роботу сервісів у різних середовищах виконання.

Для кожного мікросервісу було розроблено окремий Dockerfile із використанням принципу багатоетапної збірки (multi-stage build). На першому етапі використовувався образ із Maven для компіляції Java-проєкту та формування

виконуваного JAR-файлу, після чого готовий артефакт переносився до мінімалістичного середовища Java Runtime Environment (JRE). З погляду кібербезпеки такий підхід дозволяє суттєво зменшити поверхню потенційної атаки, оскільки фінальний контейнер не містить інструментів розробки, компіляторів або вихідного коду, які могли б бути використані зловмисником у випадку компрометації системи. Кожен мікросервіс, включаючи auth-service та resource-service, було ізольовано таким чином, щоб він мав доступ лише до необхідних файлів, конфігурацій та криптографічних ключів.

Оскільки система складається з декількох взаємопов'язаних Java-мікросервісів та бази даних PostgreSQL, для автоматизації процесу розгортання було використано інструмент Docker Compose. Це дозволило описати всю інфраструктуру в єдиному файлі docker-compose.yml, де визначено параметри контейнерів, змінні середовища, правила взаємодії сервісів та конфігурацію внутрішніх мереж. Такий підхід значно спрощує процес запуску системи, централізоване керування сервісами та забезпечує стабільну й безпечну роботу всієї мікросервісної архітектури.

Архітектуру розгортання було спроєктовано таким чином, щоб база даних PostgreSQL та сервер конфігурацій не мали прямого доступу до зовнішньої мережі Інтернет, а взаємодія між компонентами здійснювалася виключно через внутрішню віртуальну мережу Docker Bridge. Такий підхід є важливим заходом кібербезпеки, оскільки мінімізує ризик несанкціонованого доступу до сховища даних користувачів та внутрішніх сервісів системи. Загальну схему контейнеризованого розгортання системи наведено на рисунку 3.10. Додатково використання Docker Compose дозволило автоматизувати процес запуску всієї системи за допомогою єдиної команди, що значно спрощує адміністрування та подальше тестування мікросервісного середовища. Важливою перевагою контейнеризації є також можливість швидкого масштабування окремих сервісів залежно від рівня навантаження без необхідності зміни архітектури застосунку. Крім цього, ізольоване виконання контейнерів забезпечує підвищення рівня

відмовостійкості системи та мінімізує ризик поширення потенційних атак між окремими компонентами мікросервісної архітектури.

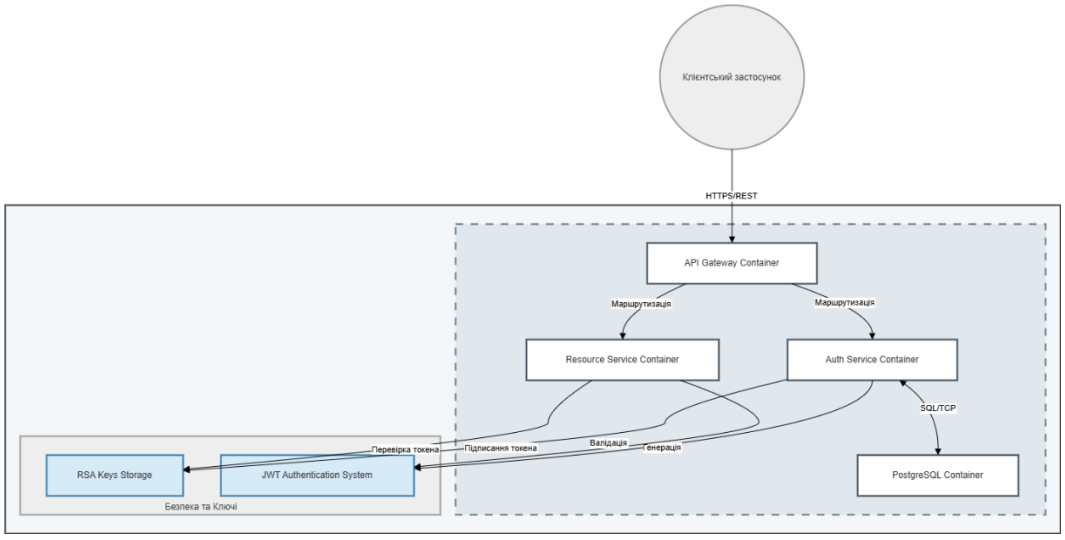


Рисунок 3.10 – Контейнеризована архітектура системи JWT-автентифікації

У межах реалізації сценарію контейнеризації значну увагу було приділено налаштуванню механізмів перевірки готовності сервісів (healthchecks) та контролю черговості їхнього запуску. За допомогою параметра depends_on було реалізовано логіку, відповідно до якої сервіс автентифікації не розпочинає роботу до повної ініціалізації бази даних PostgreSQL, а бізнес-сервіси очікують завершення запуску config-server та готовності сервісу до обробки запитів. Такий підхід дозволив уникнути помилок синхронізації під час старту системи та забезпечив своєчасне отримання всіма мікросервісами необхідних конфігурацій і RSA-ключів для коректної роботи механізмів автентифікації та валідації JWT-токенів.

Для перевірки успішності розгортання використовуються стандартні інструменти моніторингу Docker, які дозволяють контролювати стан контейнерів та параметри їхньої роботи. Після виконання команди запуску вся мікросервісна інфраструктура автоматично розгортається протягом кількох секунд, а адміністратор системи має можливість перевірити статус кожного контейнера, його мережеві параметри та коректність роботи окремих сервісів.

Реалізований рівень ізоляції забезпечує незалежність компонентів системи та дозволяє локалізувати потенційні помилки в межах окремого контейнера без впливу на роботу інших сервісів або операційної системи хоста. Такий підхід відповідає сучасним вимогам до побудови захищених мікросервісних середовищ та підвищує загальну стійкість системи до технічних збоїв. Результат успішного запуску всієї мікросервісної екосистеми наведено на рисунку 3.11.

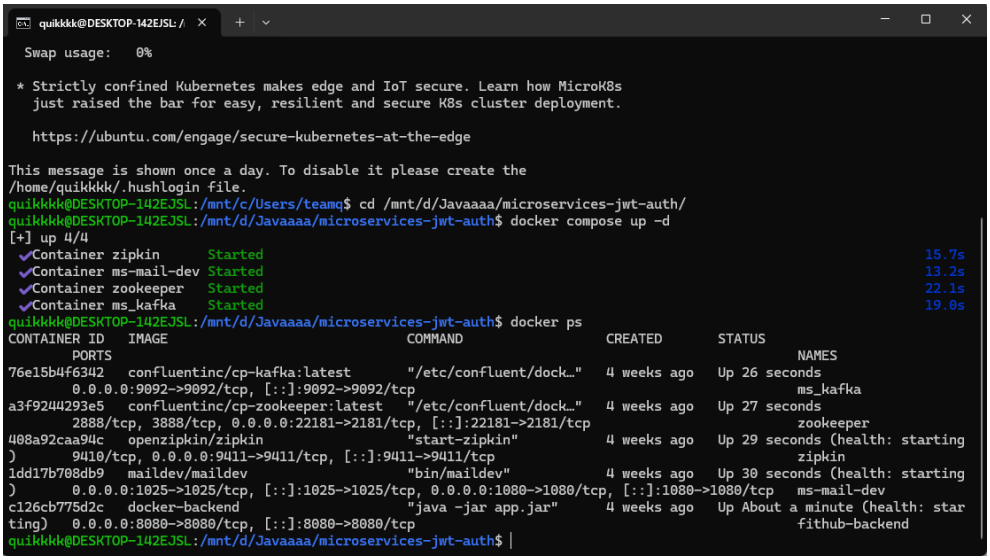


Рисунок 3.11 – Робота контейнерів системи у терміналі (docker ps)

Реалізована схема оркестрації поєднує високу швидкість роботи, ізоляцію контейнерів та комплексний підхід до забезпечення безпеки кожного компонента окремо. Завдяки цьому створене середовище є придатним для подальшого практичного тестування системи за допомогою реальних HTTP-запитів та аналізу результатів валідації JWT-токенів. Додатково така архітектура забезпечує зручність подальшого масштабування системи та спрощує процес супроводу окремих мікросервісів у процесі експлуатації. Це також дозволяє підвищити стабільність роботи системи при збільшенні навантаження та кількості одночасних клієнтських запитів. Крім цього, використання контейнеризації дозволило забезпечити уніфікованість середовища виконання для всіх компонентів системи незалежно від апаратної або програмної конфігурації сервера. Це підвищує керованість мікросервісного середовища.

3.5 Експериментальна перевірка та тестування безпеки

Завершальним етапом практичної частини роботи стала комплексна перевірка працездатності та безпеки розробленої системи. Для тестування було використано інструмент Postman, який дозволяє імітувати роботу клієнтського застосунку, керувати HTTP-заголовками та аналізувати відповіді сервера під час перевірки механізмів JWT-автентифікації. План тестування включав як стандартні сценарії коректної роботи системи, так і негативні тести, спрямовані на перевірку стійкості механізмів криптографічного захисту та валідації JWT-токенів. Перший етап тестування був присвячений перевірці процесу реєстрації нового користувача. Для цього було сформовано POST-запит до ендпоінта /auth/register сервісу автентифікації з передачею JSON-об'єкта, що містив дані користувача. У процесі виконання запиту система здійснювала перевірку унікальності електронної пошти, хешування пароля за допомогою алгоритму BCrypt та подальше збереження даних у базі PostgreSQL. Успішне виконання цього тесту підтвердило коректність взаємодії між контролером, сервісним шаром та базою даних. Результат виконання запиту наведено на рисунку 3.12. Додатково під час тестування було перевірено коректність формування HTTP-відповідей та обробки можливих помилок валідації даних. Система успішно реагувала на некоректні або неповні запити, повертаючи відповідні статус-коди та повідомлення про помилки без розкриття внутрішньої службової інформації. Це підтвердило правильність реалізації механізмів обробки винятків та відповідність API принципам безпечної взаємодії між клієнтом і сервером. Додатково результати тестування підтвердили стабільність роботи системи під час виконання основних сценаріїв автентифікації та обробки запитів користувачів. Перевірка механізмів захисту продемонстрував коректну роботу валідації JWT-токенів, контролю доступу та оброки несанкціонованих звернень до API. Це підтвердило ефективність реалізованих механізмів безпеки та готовність системи до подальшої експлуатації.

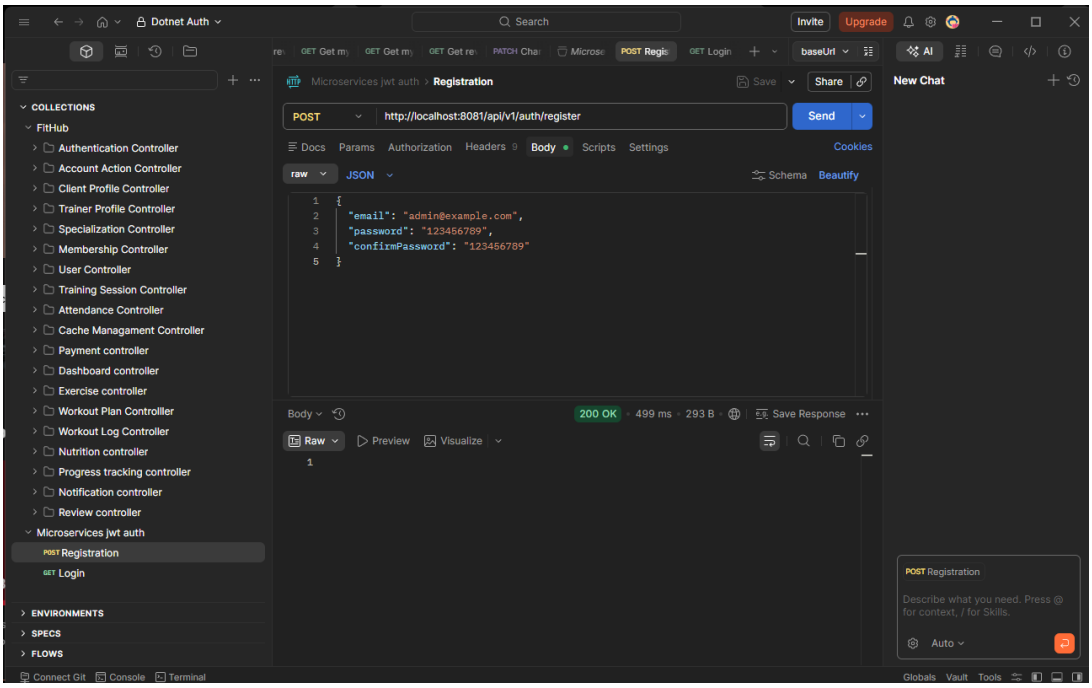


Рисунок 3.12 – Результат тестування реєстрації нового користувача в Postman

Другим етапом тестування стала перевірка процесу автентифікації користувача та отримання JWT-токена. Для цього було сформовано запит до ендпоінта /auth/login із використанням попередньо створених облікових даних. Під час виконання цього сценарію система здійснювала перевірку хешу пароля, отримання ролей користувача з бази даних та генерацію JWT-токена з цифровим RSA-підписом. У відповіді сервера було отримано об’єкт AuthenticationResponse, який містив сформований JWT-токен. Успішне отримання токена підтвердило коректну роботу механізму автентифікації, а також успішне зчитування приватного RSA-ключа та створення валідного цифрового підпису. Додатково під час тестування було перевірено коректність передачі JWT-токена у HTTP-відповідях та можливість його подальшого використання для доступу до захищених ресурсів системи. Система успішно виконала формування токена з усіма необхідними claims та забезпечила його криптографічну цілісність. Також під час тестування було підтверджено коректну взаємодію між Auth Service та модулем криптографічної генерації токенів у межах мікросервісної архітектури. Результат виконання цього етапу тестування наведено на рисунку 3.13.

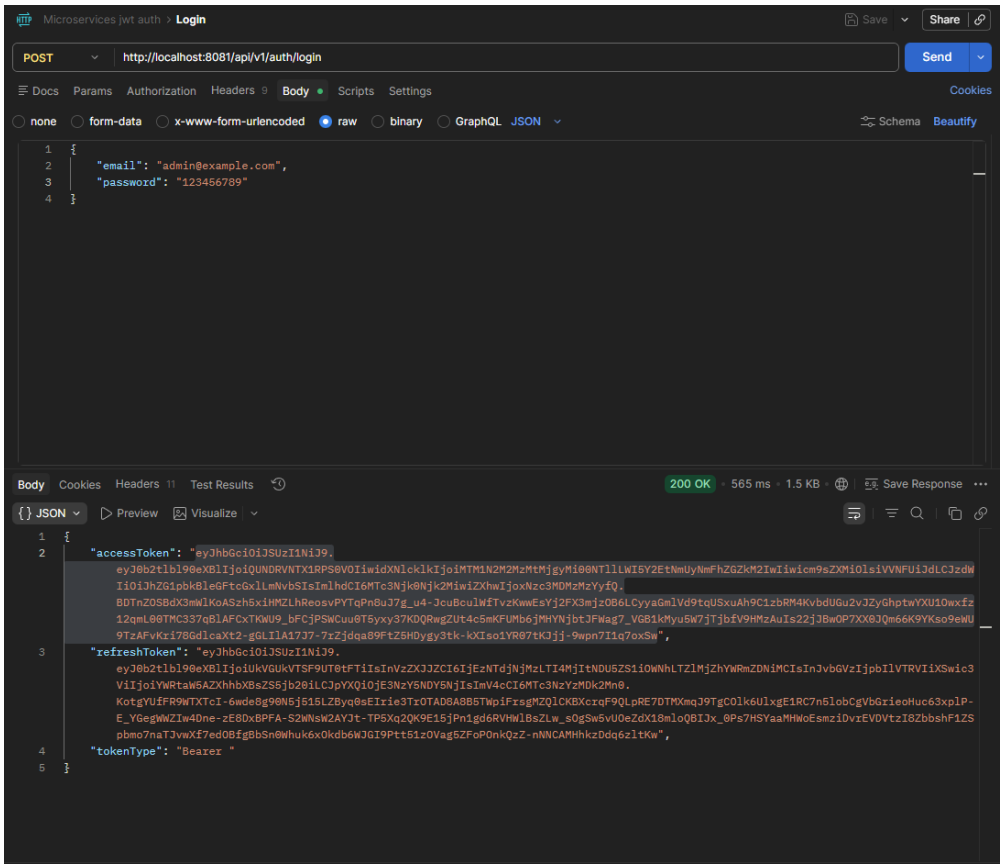


Рисунок 3.13 – Отримання JWT-токена після успішної автентифікації

Після отримання JWT-токена було виконано перевірку доступу до захищених ресурсів у resource-service. Для цього до HTTP-запиту в заголовок Authorization було додано JWT за схемою Bearer. У процесі тестування перевірялася робота компонентів JwtFilter та KeyUtils, які виконують вилучення токена, завантаження публічного RSA-ключа та перевірку цифрового підпису JWT. Додатково було підтверджено коректність роботи механізму авторизації на основі JWT-токенів. Отримання відповіді зі статус-кодом 200 OK підтвердило успішну реалізацію механізму автономної валідації токенів без звернення до бази даних або сервісу автентифікації. Під час перевірки також було встановлено, що система коректно обробляє права доступу користувача та забезпечує захист API-ендпоінтів відповідно до заданих ролей підтримуючи стабільну та безпечну взаємодію між мікросервісами. Це підтвердило коректну роботу механізму RBAC та ефективність використання JWT-токенів для реалізації авторизації у розподіленому середовищі. Результат тестування наведено на рисунку 3.14.

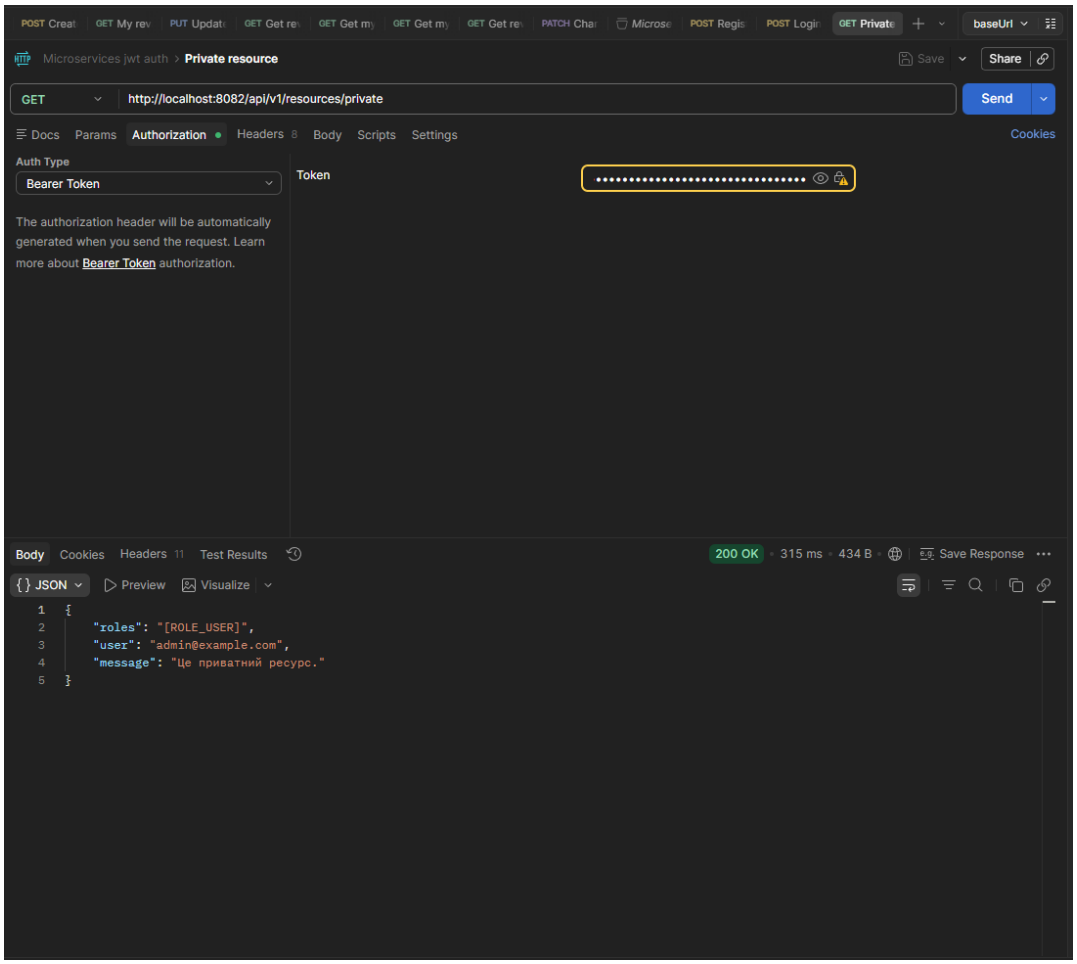


Рисунок 3.14 – Успішне звернення до захищеного API з використанням Bearer-токена

Окрему увагу під час тестування було приділено перевірці стійкості системи до спроб обходу механізмів захисту. Для цього було виконано експеримент із навмисною зміною одного символу в JWT-токені, а також протестовано роботу системи з простроченим токеном. В обох випадках механізм RSA-валідації успішно виявив порушення цілісності токена або невалідність часових міток та заблокував доступ із кодом 403 Forbidden або 401 Unauthorized. Отримані результати підтвердили стійкість системи до підробки JWT-токенів та атак повторного використання. Крім цього, було перевірено коректність роботи механізму обробки помилок безпеки та формування HTTP-відповідей при спробах несанкціонованого доступу. Система успішно ідентифікувала невалідні та прострочені JWT-токени, автоматично перериваючи подальшу обробку запиту ще

на рівні JwtFilter. Це підтвердило ефективність реалізованого механізму захисту від підробки токенів, повторного використання скомпрометованих JWT та інших типових атак на систему автентифікації. Результат відмови у доступі наведено на рисунку 3.15.

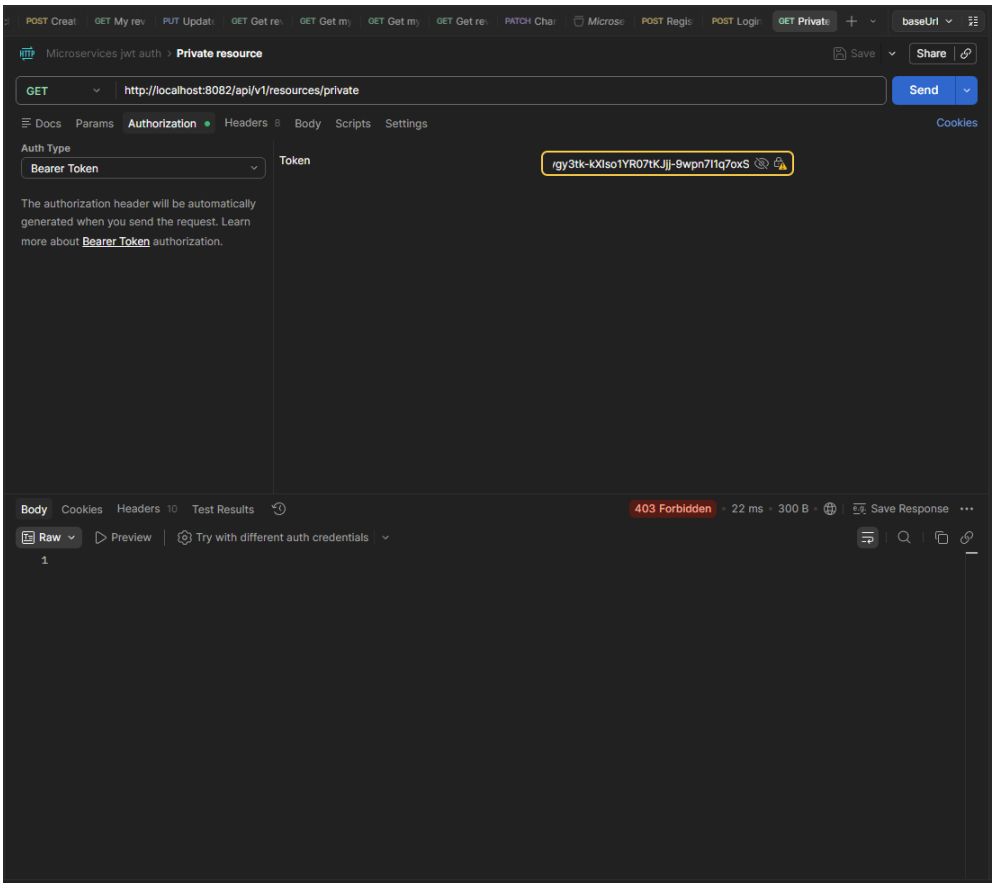


Рисунок 3.15 – Реакція системи на використання невалідного або підробленого токена

Для додаткового аналізу структури JWT-токена було використано онлайн-інстурмент jwt.io. За його допомогою виконано візуалізацію структури токена, зокрема заголовка, корисного навантаження із користувацькими claims та секції цифрового підпису. Під час аналізу також було перевірено коректність формування claims, наявність часових міток та валідність цифрового RSA-підпису токена. Це дозволило підтвердити відповідність структури JWT вимогам безпечної автентифікації та авторизації користувачів. Результат аналізу структури токена наведено на рисунку 3.16.

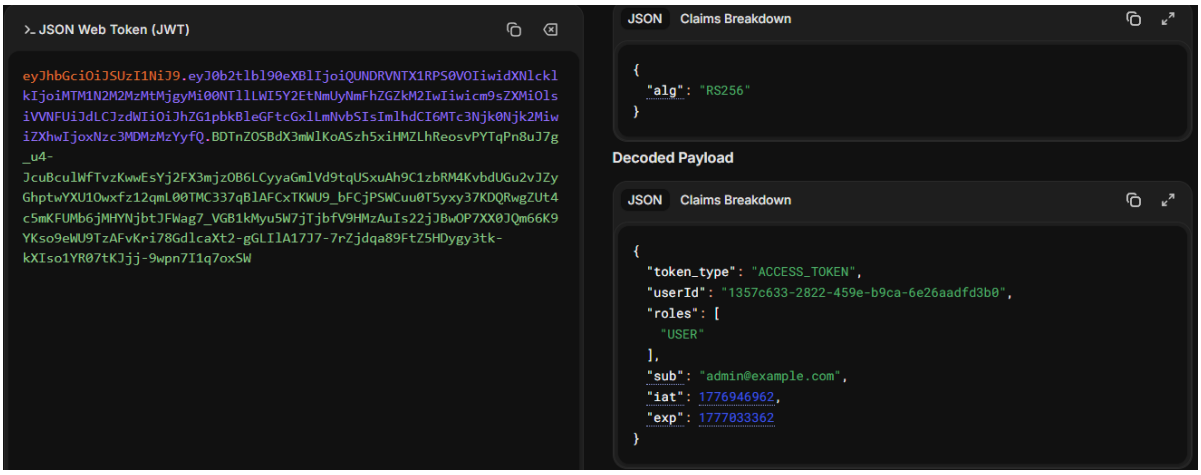


Рисунок 3.16 – Дешифрування та аналіз структури JWT-токена через сервіс jwt.io

Завершальним етапом розробки є перевірка стійкості системи автентифікації до типових кіберзагроз. Оскільки система базується на використанні JWT та асиметричного алгоритму підпису RS256, було проведено практичне дослідження стійкості до атаки типу Algorithm Confusion (підміна алгоритму). Суть цієї атаки полягає у спробі змусити сервер перевірити токен за допомогою симетричного алгоритму HS256, використовуючи публічний RSA-ключ сервера як секретний ключ HMAC. Для реалізації цього сценарію було розроблено тестовий скрипт на мові Python, який виконує роль інструменту для проведення Proof-of-Concept (PoC) атаки. Скрипт автоматично зчитує публічний ключ мікросервісу, формує корисне навантаження з правами адміністратора та підписує його за алгоритмом HS256. На рисунку 3.17 продемонстровано роботу скрипта. Як видно з результатів виконання, атакуючий успішно згенерував підроблений токен, проте при спробі доступу до захищеного ресурсу (resource-service) система повернула статус 401 Unauthorized. Це підтвердило ефективність реалізованого механізму перевірки цифрового підпису та стійкість системи до спроб підробки JWT-токенів. Отримані результати демонструють, що використання асиметричного RSA-підпису забезпечує надійний захист механізмів автентифікації навіть у випадку спроб експлуатації відомих вразливостей JWT.

Рисунок 3.17 – Результат тестування системи на стійкість до атаки Algorithm Confusion

Для детального аналізу причин відхилення запиту було досліджено логи мікросервісу ресурсів. Аналіз підтвердив, що захисний механізм спрацював на рівні бібліотеки jjwt, можна побачити на рисунку 3.18.

Рисунок 3.18 – Аналіз серверних логів при виявленні недійсного алгоритму підпису

Згідно з зафіксованим виключенням JwtException, система розпізнала невідповідність: токен містив заголовок про використання HS256, тоді як конфігурація JwtParser суворо налаштована на валідацію за допомогою публічного ключа RSA. Це доводить, що використання типізованих об’єктів PublicKey при налаштуванні фільтрів безпеки у розробленому рішенні повністю нівелює можливість підміни алгоритмів. Таким чином, результати експериментальної перевірки підтверджують високий рівень захищеності системи автентифікації. Впроваджені механізми валідації токенів коректно обробляють аномальні запити та запобігають несанкціонованому підвищенню привілеїв користувачів.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було досліджено особливості побудови систем автентифікації користувачів у мікросервісній архітектурі та проаналізовано сучасні підходи й технології забезпечення безпеки веб-додатків. Зокрема, розглянуто основні методи автентифікації, такі як сесійна, токен-орієнтовна та багатофакторна автентифікація, а також стандартизовані протоколи, зокрема OAuth 2.0, OpenID Connect, SAML і концепцію Single Sign-On.

У результаті аналізу встановлено, що для мікросервісних систем найбільш ефективним є використанням токен-орієнтовного підходу, зокрема технології JWT, яка забезпечує безстанну автентифікацію, підвищує масштабованість системи та зменшує навантаження на сервер автентифікації.

У межах роботи було сформульовано функціональні вимоги до системи автентифікації, спроєктовано її архітектуру та обґрунтовано вибір технологій для реалізації. Реалізовано серверну частину системи з використанням сучасних інструментів, забезпечено механізм генерації та перевірки JWT-токенів, а також організовано взаємодію між компонентами мікросервісної архітектури.

Проведене тестування підтвердило працездатність розробленої частини та її відповідність поставленим вимогам. Запропоноване рішення демонструє ефективність використання JWT для організації безпечної та масштабованої автентифікації у розподілених системах.

Таким чином, поставлену мету роботи досягнуто, а отримані результати можуть бути використані при розробці сучасних веб-додатків та інформаційних систем, що базуються на мікросервісній архітектурі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Spilca L. Spring Security in Action. Shelter Island : Manning Publications, 2021. 504 с.
2. Madden N. API Security in Action. Shelter Island : Manning Publications, 2021. 504 с.
3. Carnell J., Sanchez M. Spring Microservices in Action. 2nd ed. Shelter Island : Manning Publications, 2021. 456 с.
4. Newman S. Building Microservices. 2nd ed. Sebastopol : O'Reilly Media, 2021. 614 с.
5. Walls C. Spring in Action. 6th ed. Shelter Island : Manning Publications, 2022. 528 с.
6. Späth P. Pro Spring Security: Securing Spring Boot 3 Applications. Berkeley : Apress, 2023. 430 с.
7. Macero M. Learn Microservices with Spring Boot 3. Berkeley : Apress, 2023. 412 с.
8. Davis A. Cloud Native Spring in Action. Shelter Island : Manning Publications, 2023. 425 с.
9. Laurentius T. Spring Boot 3 and Spring Framework 6. Berkeley : Apress, 2024. 350 с.
10. Richards M., Ford N. Software Architecture: The Hard Parts. Sebastopol : O'Reilly Media, 2021. 432 с.
11. Colombo E. Practical Microservices Architectural Patterns. Birmingham : Packt Publishing, 2022. 380 с.
12. Siriwardena P. Advanced API Security: OAuth 2.0 and Beyond. 2nd ed. Berkeley : Apress, 2022. 340 с.
13. Hoffman L. Web Application Security. Sebastopol : O'Reilly Media, 2021. 320 с.
14. Brikko O. Container Security: Fundamental Technology Concepts. Sebastopol : O'Reilly Media, 2023. 290 с.

15. Das S. Mastering Spring Boot 3.0. Birmingham : Packt Publishing, 2023. 410 с.

16. Indrasiri K., Siriwardena P. Microservices Security in Action. Shelter Island : Manning Publications, 2021. 370 с.

17. Parecki A. OAuth 2.0 Simplified. Raleigh : Lulu Press, 2021. 140 с.

18. Erl T. Microservice Security Patterns. Boston : Prentice Hall, 2023. 320 с.

19. Reynolds J. Docker for Java Developers. Birmingham : Packt Publishing, 2024. 280 с.

20. Sharma A. Master Spring Boot 3. New Delhi : BPB Publications, 2024. 350 с.

21. Cosmina I. Pro Spring 6: An In-Depth Guide to the Spring Framework. Berkeley : Apress, 2023. 850 с.

22. Urma R.-G., Fusco M., Mycroft A. Modern Java in Action. Shelter Island : Manning Publications, 2021. 592 с.

23. Richardson C. Microservices Patterns. Shelter Island : Manning Publications, 2021. 520 с.

24. Carlson C. Cloud Native Security. Sebastopol : O'Reilly Media, 2021. 350 с.

25. Deinum M., Rubio D. Spring Boot 3 Recipes. Berkeley : Apress, 2024. 400 с.

26. Microservices Security Cheat Sheet / OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html (дата звернення: 05.03.2026).

27. REST Security Cheat Sheet / OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (дата звернення: 15.04.2026).

28. JSON Web Token for Java Cheat Sheet / OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html (дата звернення: 18.04.2026).

29. Password Storage Cheat Sheet / OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата звернення: 18.04.2026).

30. Spring Boot Reference Documentation / VMware Tanzu. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/ (дата звернення: 18.04.2026).

31. Spring Security Reference Documentation / VMware Tanzu. URL: https://docs.spring.io/spring-security/reference/ (дата звернення: 03.05.2026).

32. Spring Cloud Config Documentation / VMware Tanzu. URL: https://docs.spring.io/spring-cloud-config/docs/current/reference/html/ (дата звернення: 03.05.2026).

33. Spring Cloud Netflix Documentation / VMware Tanzu. URL: https://spring.io/projects/spring-cloud-netflix (дата звернення: 12.05.2026).

34. Spring Data JPA Reference Documentation / VMware Tanzu. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (дата звернення: 12.05.2026).

35. RFC 7519: JSON Web Token (JWT) / IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 14.05.2026).

36. RFC 6749: The OAuth 2.0 Authorization Framework / IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 14.05.2026).

37. RFC 7636: Proof Key for Code Exchange by OAuth Public Clients / IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc7636 (дата звернення: 15.05.2026).

38. Introduction to JSON Web Tokens / Auth0. URL: https://jwt.io/introduction (дата звернення: 16.05.2026).

39. PostgreSQL 16 Documentation / PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/16/ (дата звернення: 18.05.2026).

40. Docker Documentation / Docker Inc. URL: https://docs.docker.com/ (дата звернення: 18.05.2026).

41. Docker Compose Overview / Docker Inc. URL: https://docs.docker.com/compose/ (дата звернення: 19.05.2026).

42. Keycloak Documentation / Red Hat. URL: https://www.keycloak.org/documentation (дата звернення: 19.05.2026).

43. Microservices Architecture Style / Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (дата звернення: 20.05.2026).

44. What is OpenID Connect, and how does it work? / Curity. URL: https://curity.io/resources/learn/openid-connect-overview/ (дата звернення: 21.05.2026).

45. Securing Microservices: Best Practices / Kong Inc. URL: https://konghq.com/blog/engineering/securing-microservices (дата звернення: 21.05.2026).

ДОДАТОК А
Код програми

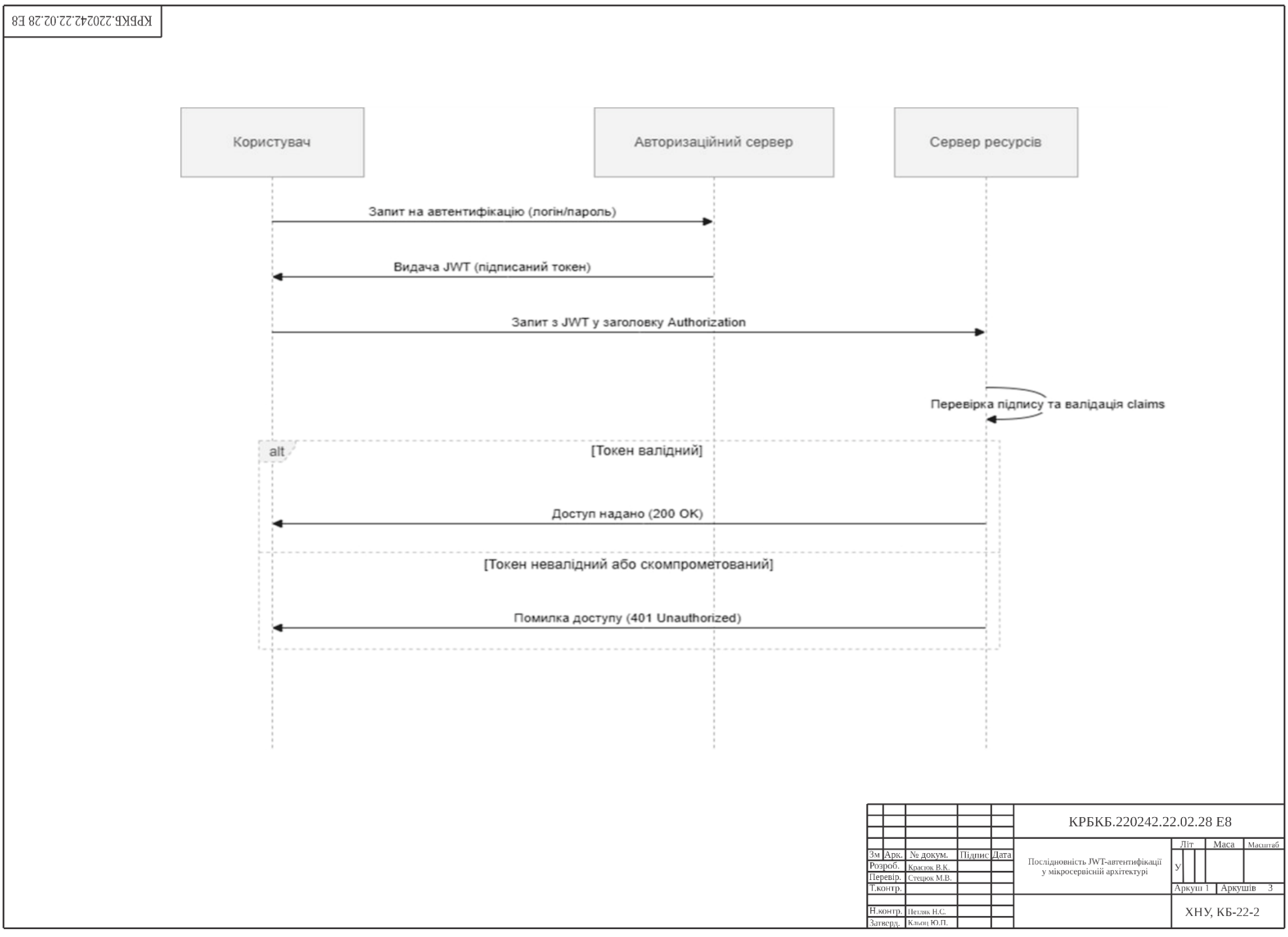
```
@Component
@RequiredArgsConstructor
public class JwtFilter extends OncePerRequestFilter {
    private final IJwtService jwtService;
    @Override
    protected void doFilterInternal(
            @NonNull HttpServletRequest request,
            @NonNull HttpServletResponse response,
            @NonNull FilterChain filterChain
    ) throws ServletException, IOException {
        if (request.getServletPath().equals("/api/v1/auth/login") ||
                request.getServletPath().equals("/api/v1/auth/register") ||
                request.getServletPath().equals("/api/v1/auth/refresh")
        ) {
            filterChain.doFilter(request, response);
            return;
        }
        String authHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }
        String jwt = authHeader.substring(7);
        String email = jwtService.extractEmail(jwt);
        String userId = jwtService.extractUserId(jwt);
        if (email != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            if (jwtService.isTokenValid(jwt, email)) {
                var roles = jwtService.extractRoles(jwt);
                var authorities = roles.stream()
                        .map(r -> new SimpleGrantedAuthority("ROLE_" + r))
                        .toList();
                var principal = new UserPrincipal(userId, email, Set.copyOf(roles));
                var authToken = new UsernamePasswordAuthenticationToken(principal, null, authorities);
                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
public class KeyUtils {
    @SneakyThrows
    public static PrivateKey loadPrivateKey(String pemPath) {
        String key = readKeyFromResource(pemPath)
                .replace("-----BEGIN PRIVATE KEY-----", "")
                .replace("-----END PRIVATE KEY-----", "")
                .replaceAll("\\s+", "");

        byte[] decoded = Base64.getDecoder().decode(key);
        PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(decoded);

        return KeyFactory.getInstance("RSA").generatePrivate(spec);
    }
    public static PublicKey loadPublicKey(String pemPath) throws Exception {
        String key = readKeyFromResource(pemPath)
                .replace("-----BEGIN PUBLIC KEY-----", "")
                .replace("-----END PUBLIC KEY-----", "")
```

```
            .replaceAll("\\s+", "");

        byte[] decoded = Base64.getDecoder().decode(key);
        X509EncodedKeySpec spec = new X509EncodedKeySpec(decoded);

        return KeyFactory.getInstance("RSA").generatePublic(spec);
    }
    private static String readKeyFromResource(String pemPath) throws IOException {
        try (InputStream is = KeyUtils.class.getClassLoader().getResourceAsStream(pemPath)) {
            if (is == null) throw new IllegalArgumentException("Could not found key file: " + pemPath);
            return new String(is.readAllBytes());
        }
    }
}
@Service
public class JwtServiceImpl implements IJwtService {
    private static final String TOKEN_TYPE = "token_type";
    private static final String USER_ID = "userId";
    private static final String PATH_TO_PRIVATE_KEY = "keys/local-only/private_key.pem";
    private static final String PATH_TO_PUBLIC_KEY = "keys/local-only/public_key.pem";
    private static final PrivateKey PRIVATE_KEY;
    private static final PublicKey PUBLIC_KEY;
    static {
        try {
            PRIVATE_KEY = KeyUtils.loadPrivateKey(PATH_TO_PRIVATE_KEY);
            PUBLIC_KEY = KeyUtils.loadPublicKey(PATH_TO_PUBLIC_KEY);
        } catch (Exception e) {
            throw new RuntimeException("Failed to load JWT Keys", e);
        }
    }
    @Value("${app.security.jwt.access-token-expiration}")
    private long accessTokenExpiration;
    @Value("${app.security.jwt.refresh-token-expiration}")
    private long refreshTokenExpiration;
    @Override
    public String generateAccessToken(User user) {
        Map<String, Object> claims = Map.of(
                TOKEN_TYPE, "ACCESS_TOKEN",
                USER_ID, user.getId(),
                "roles", user.getRoles().stream().map(Role::getName).toList()
        );

        return buildToken(user.getEmail(), claims, accessTokenExpiration);
    }
    @Override
    public String generateRefreshToken(User user) {
        Map<String, Object> claims = Map.of(
                TOKEN_TYPE, "REFRESH_TOKEN",
                USER_ID, user.getId(),
                "roles", user.getRoles().stream().map(Role::getName).toList()
        );
        return buildToken(user.getEmail(), claims, refreshTokenExpiration);
    }
    @Override
    public String refreshAccessToken(String refreshToken) {
        Claims claims = extractClaims(refreshToken);
        if (!"REFRESH_TOKEN".equals(claims.get(TOKEN_TYPE))) throw new RuntimeException("Invalid token type");
        if (isTokenExpired(refreshToken)) throw new RuntimeException("Token expired");
        String email = claims.getSubject();
        String userId = claims.get(USER_ID).toString();
        Map<String, Object> claimsForNewToken = Map.of(
                TOKEN_TYPE, "ACCESS_TOKEN",
                USER_ID, userId,
```

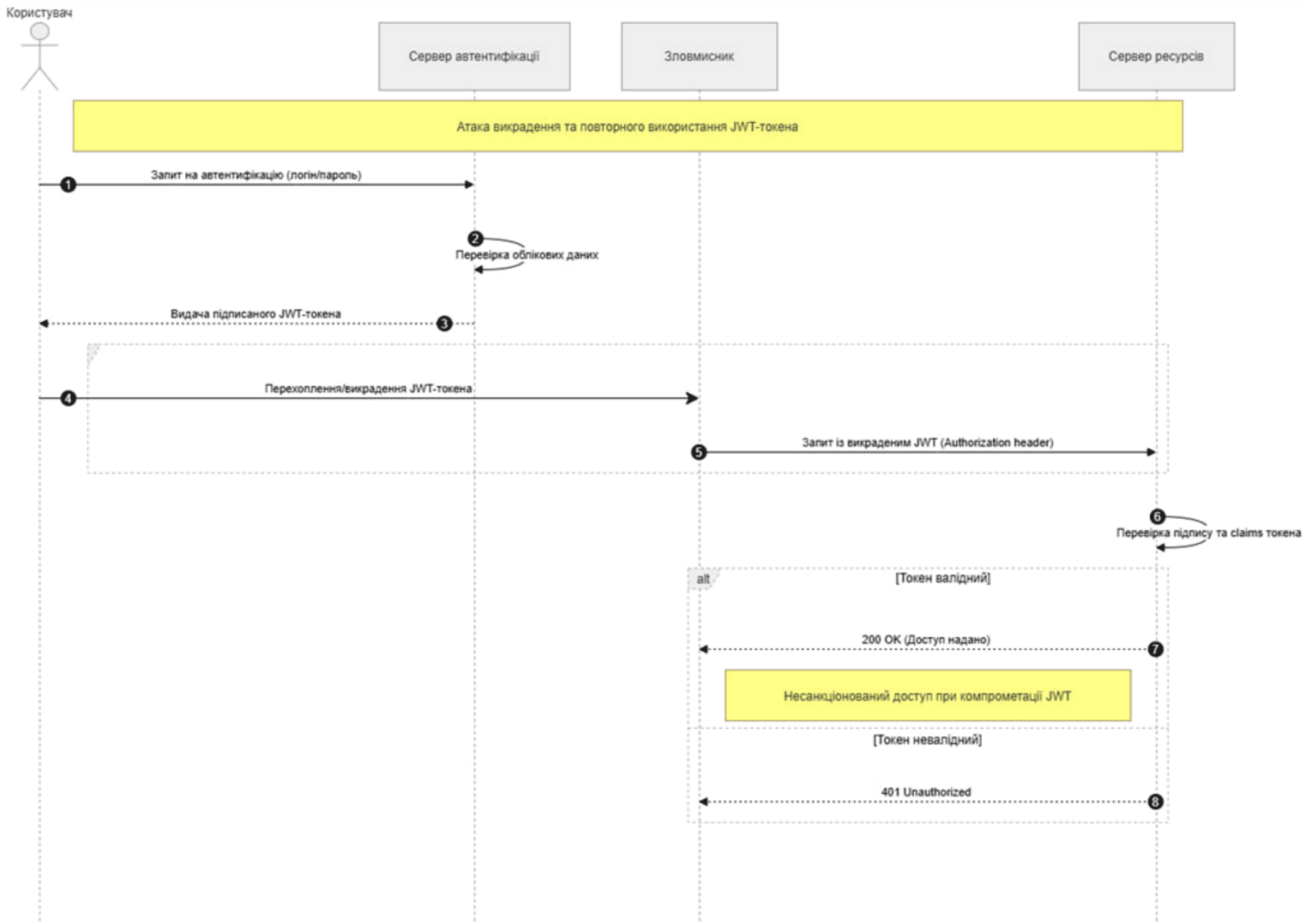
```
                "roles", claims.get("roles")
        );
        return buildToken(email, claimsForNewToken, accessTokenExpiration);
    }
    @Override
    public String extractEmail(String token) {
        return extractClaims(token).getSubject();
    }
    @Override
    public String extractUserId(String token) {
        return extractClaims(token).getId();
    }
    @Override
    public String extractTokenType(String token) {
        return extractClaims(token).get(TOKEN_TYPE).toString();
    }
    @Override
    @SuppressWarnings("unchecked")
    public List<String> extractRoles(String token) {
        return (List<String>) extractClaims(token).get("roles");
    }

    @Override
    public Date extractExpiration(String token) {
        return extractClaims(token).getExpiration();
    }
    @Override
    public boolean isTokenValid(String token, String expectedEmail) {
        return false;
    }
    private String buildToken(String email, Map<String, Object> claims, long expiration) {
        return Jwts.builder()
                .claims(claims)
                .subject(email)
                .issuedAt(new Date(System.currentTimeMillis()))
                .expiration(new Date(System.currentTimeMillis() + expiration))
                .signWith(PRIVATE_KEY)
                .compact();
    }
    private Claims extractClaims(String token) {
        try {
            return Jwts.parser()
                    .verifyWith(PUBLIC_KEY)
                    .build()
                    .parseSignedClaims(token)
                    .getPayload();
        } catch (JwtException e) {
            throw new JwtException("Invalid JWT token", e);
        }
    }
    private boolean isTokenExpired(String token) {
        return extractClaims(token)
                .getExpiration()
                .before(new Date());
    }
}

def perform_algorithm_confusion_attack():
    with open(PUBLIC_KEY_PATH, "r") as f:
        public_key = f.read()

    public_key = public_key.replace("-----BEGIN PUBLIC KEY-----", "")
    public_key = public_key.replace("-----END PUBLIC KEY-----", "")
```

```
    public_key = public_key.strip()

    payload = {
        "sub": "attacker@example.com",
        "roles": ["ROLE_ADMIN"],
        "iat": int(time.time()),
        "exp": int(time.time()) + 3600,
        "token_type": "ACCESS"
    }

    forged_token = jwt.encode(
        payload,
        public_key,
        algorithm="HS256"
    )

    print("--- [!] Сгенерований підроблений токен ---")
    print(forged_token)
    print("\n--- Деталі заголовка (Header) ---")
    print(jwt.get_unverified_header(forged_token))

    return forged_token

if __name__ == "__main__":
    token = perform_algorithm_confusion_attack()


    import requests

    url = "http://localhost:8081/api/v1/resource"
    headers = {"Authorization": f"Bearer {token}"
    print(f"\n--- [!] Спроба доступу до {url} ---")
    try:
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            print("УСПІХ: Система вразлива! Отримано доступ до ресурсів.")
            print(f"Дані від сервера: {response.json()}")
        elif response.status_code == 401 or response.status_code == 403:
            print(f"ВІДМОВА ({response.status_code}): Система захищена від цієї атаки.")
        else:
            print(f"Помилка: Сервер повернув код {response.status_code}")

    except requests.exceptions.ConnectionError:
        print("ПОМИЛКА: Не вдалося з'єднатися з сервером.")
```

ДОДАТОК Б
(Обов’язковий)
Копії графічної частини

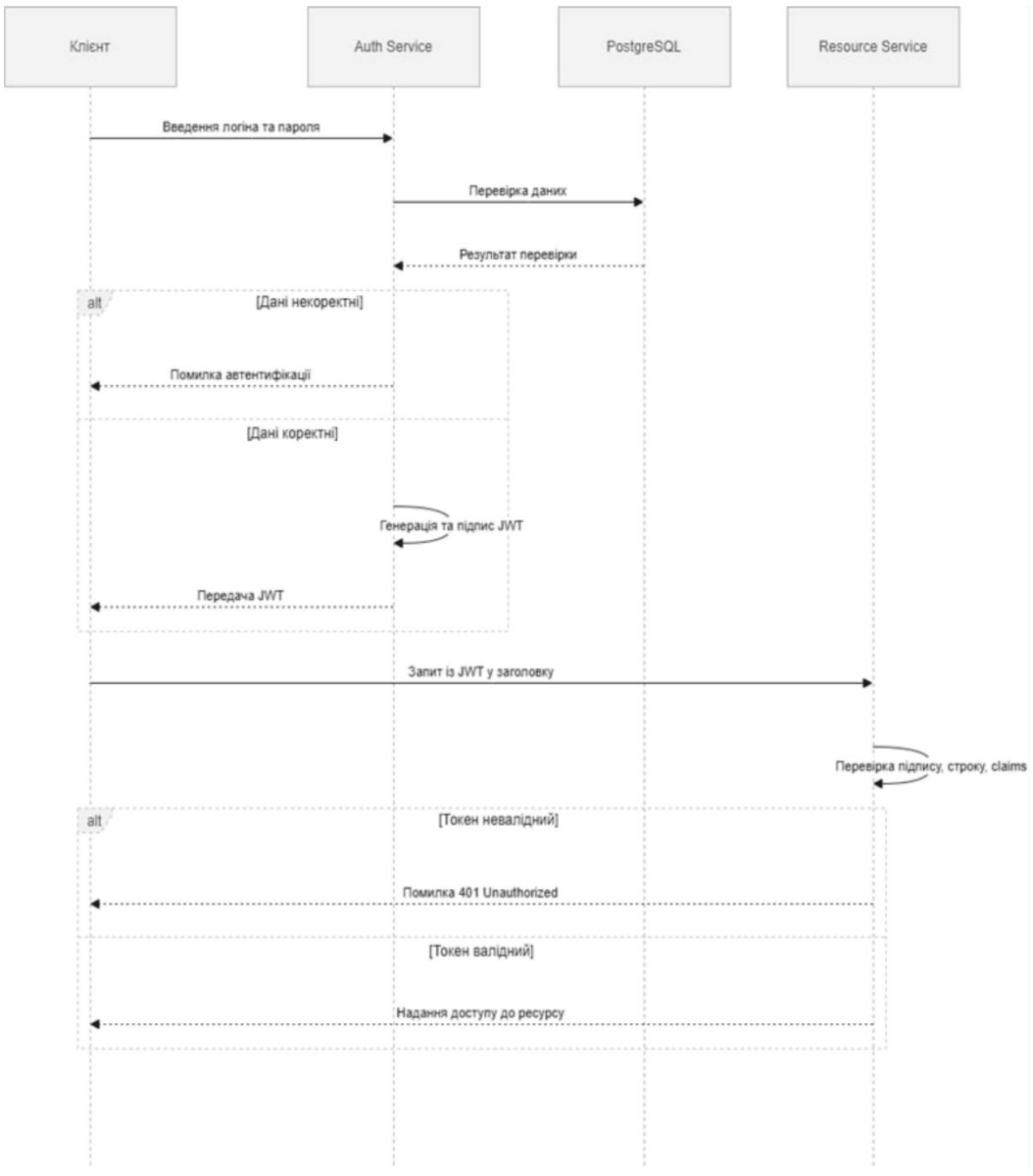


					КРБКБ.220242.22.02.28 Е8			
Зм	Арк.	№ докум.	Підпис	Дата	Послідовність JWT-автентифікації у мікросервісній архітектурі	Літ	Маса	Масштаб
Розроб.		Красюк В.К.				У		
Перевір.		Стецюк М.В.				Аркуш 1	Аркушів 3	
Т.контр.								
Н.контр.		Петлик Н.С.			ХНУ, КБ-22-2			
Затверд.		Кльоц Ю.П.						



					КРБКБ.220242.22.02.28 Е8		
Зм	Арк.	№ докум.	Підпис	Дата	Викрадення та повторне використання JWT-токена	Літ	Маса / Масштаб
Розроб.		Красюк В.К.				У	
Перевір.		Стецюк М.В.				Аркуш 2	Аркушів 3
Т.контр.							
Н.контр.		Петляк Н.С.				ХНУ, КБ-22-2	
Затверд.		Кльоц Ю.П.					

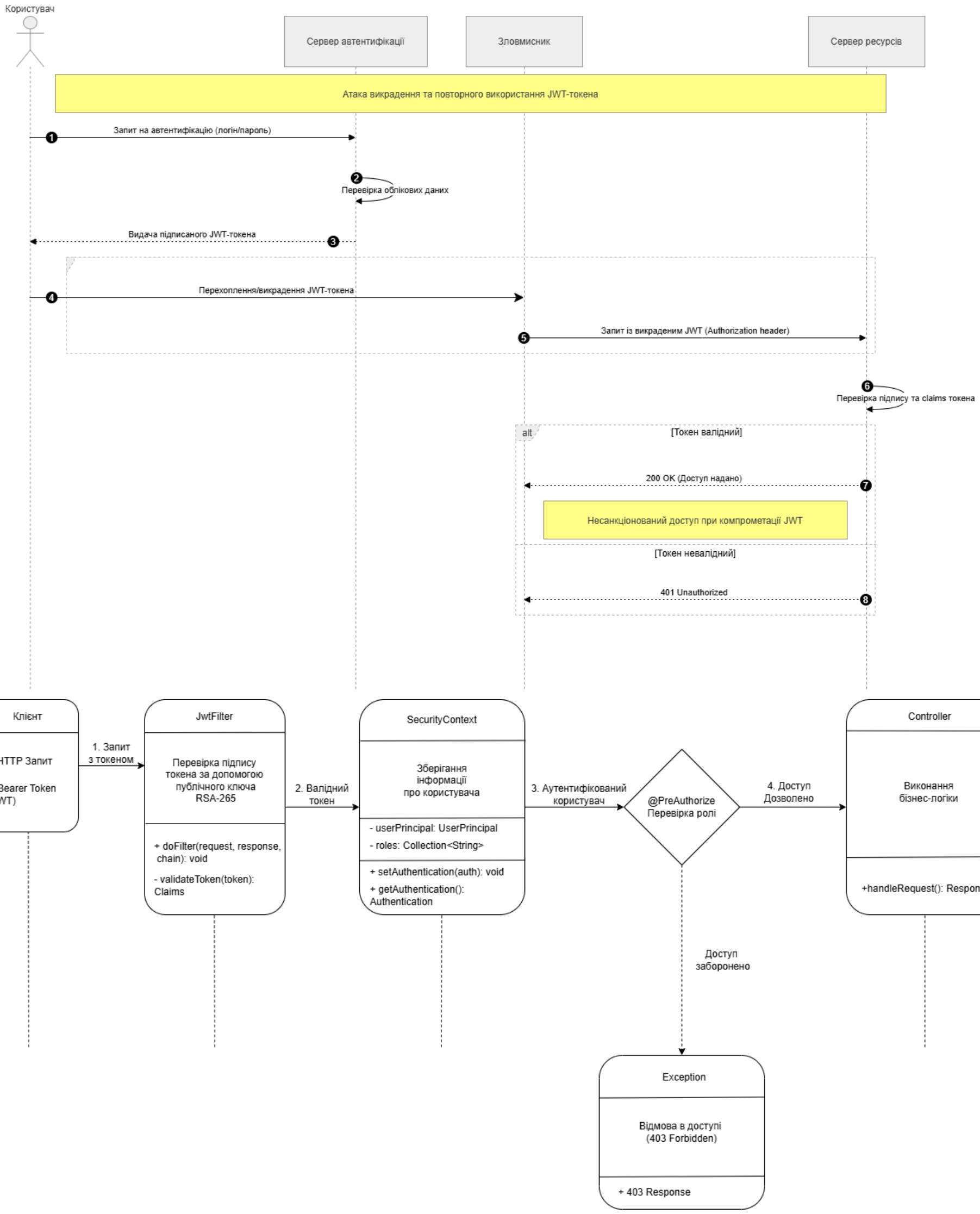
КРБКБ.220242.22.02.28 Е8

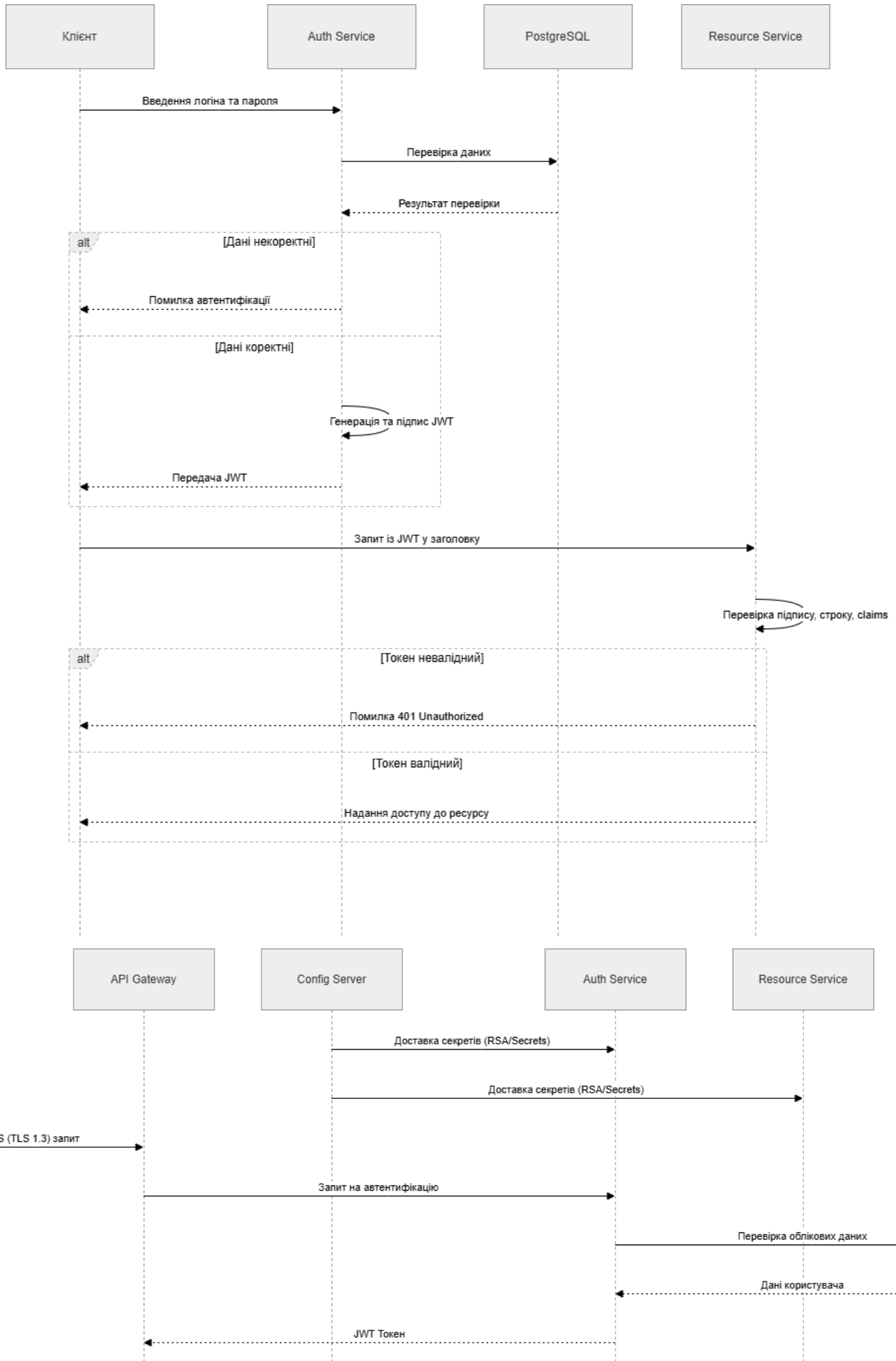


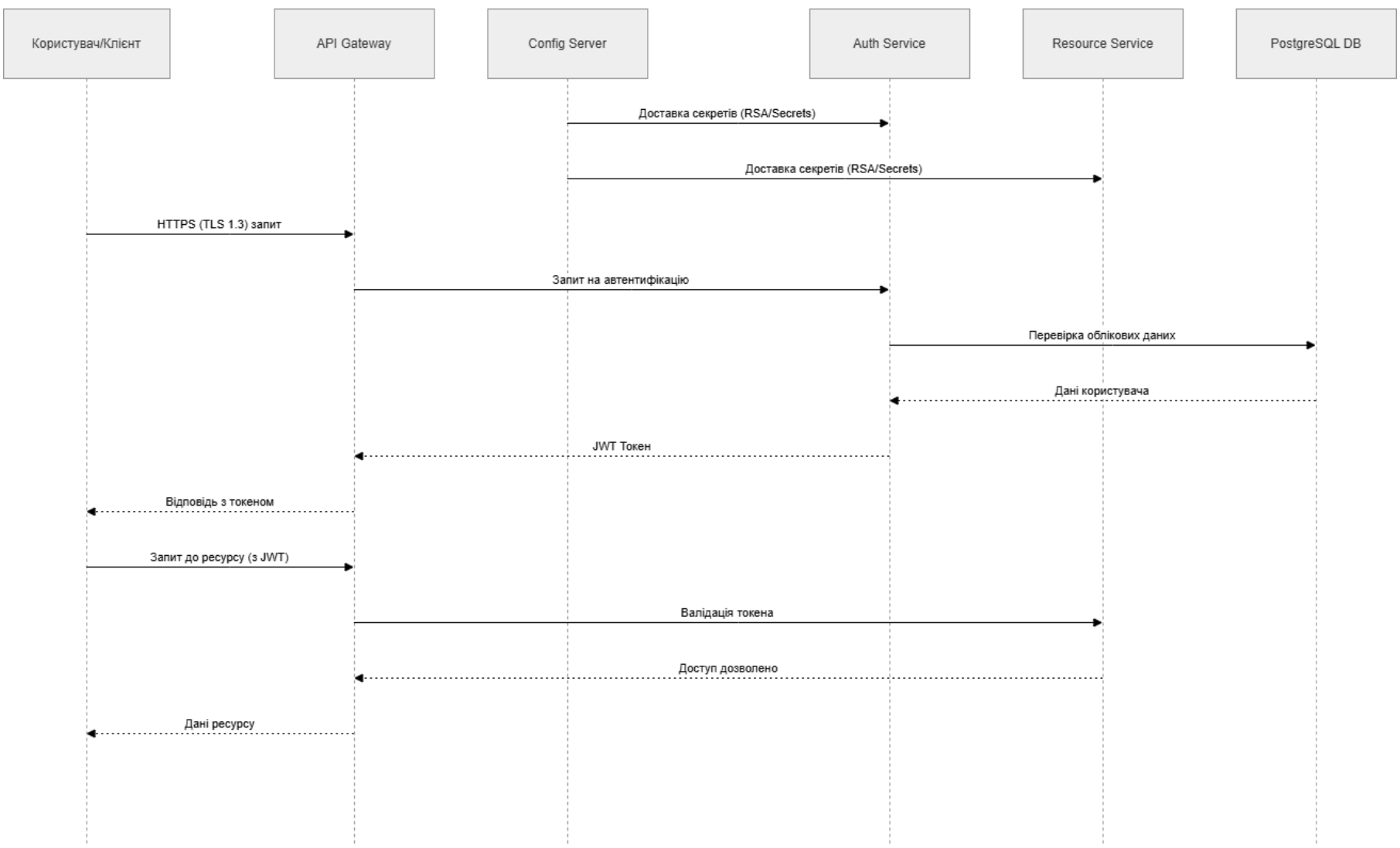
					КРБКБ.220242.22.02.28 Е8		
Зм	Арк.	№ докум.	Підпис	Дата	Алгоритм функціонування системи JWT-автентифікації	Літ	Маса / Масштаб
Розроб.		Красюк В.К.				У	
Перевір.		Стецюк М.В.				Аркуш 3	Аркушів 3
Т.контр.							
Н.контр.		Петляк Н.С.				ХНУ, КБ-22-2	
Затверд.		Кльоц Ю.П.					

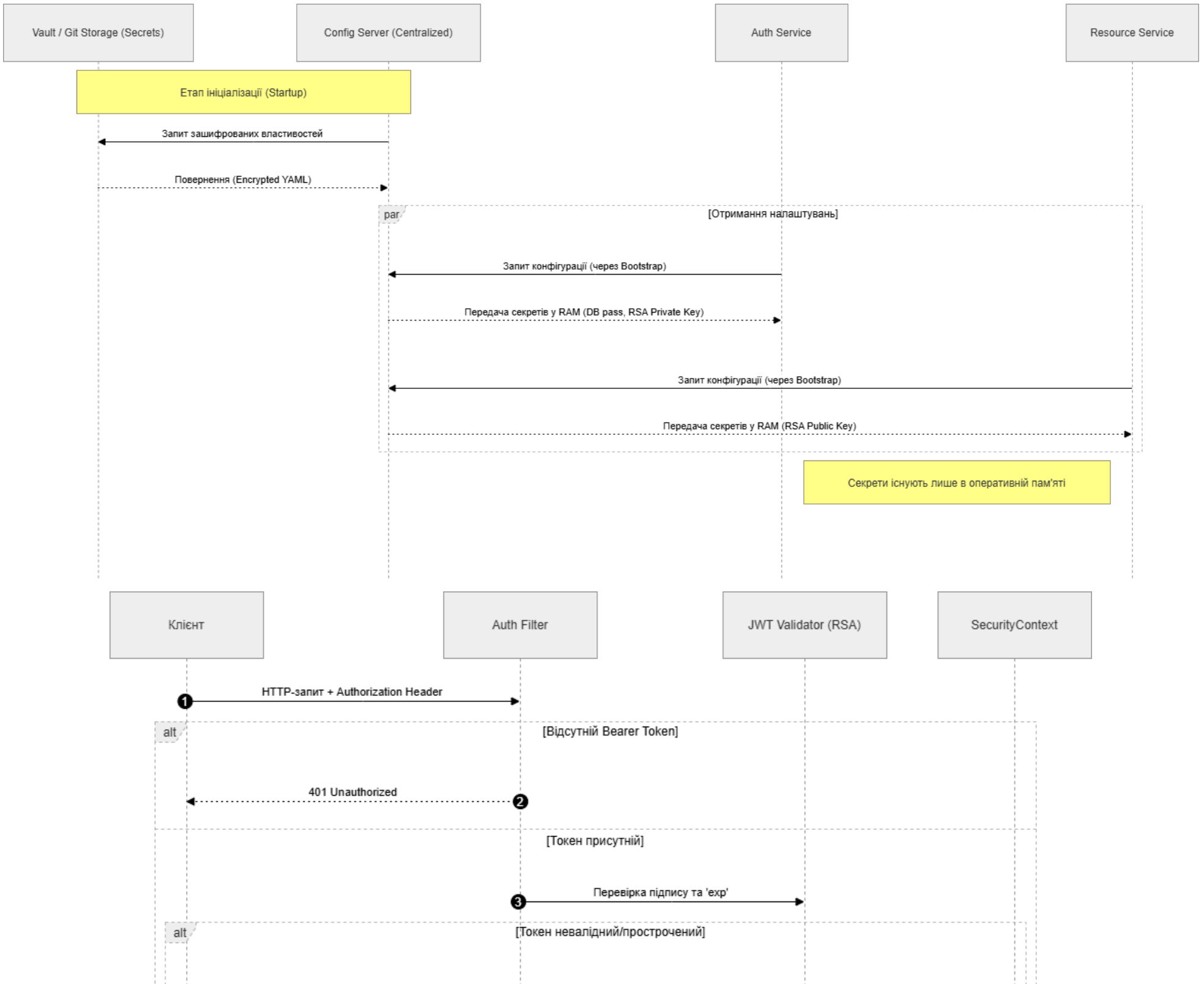
ДОДАТОК В

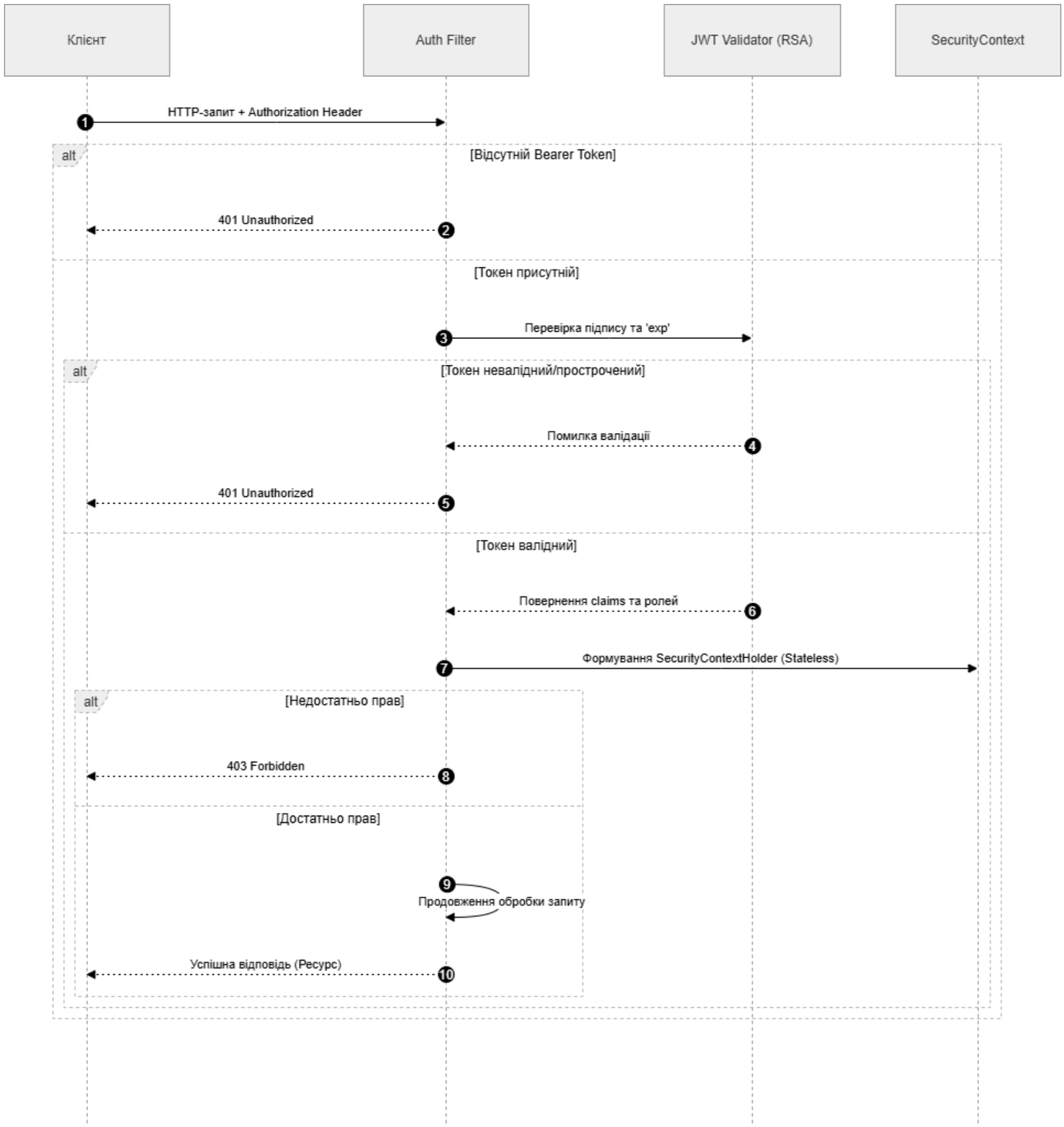
Рисунки

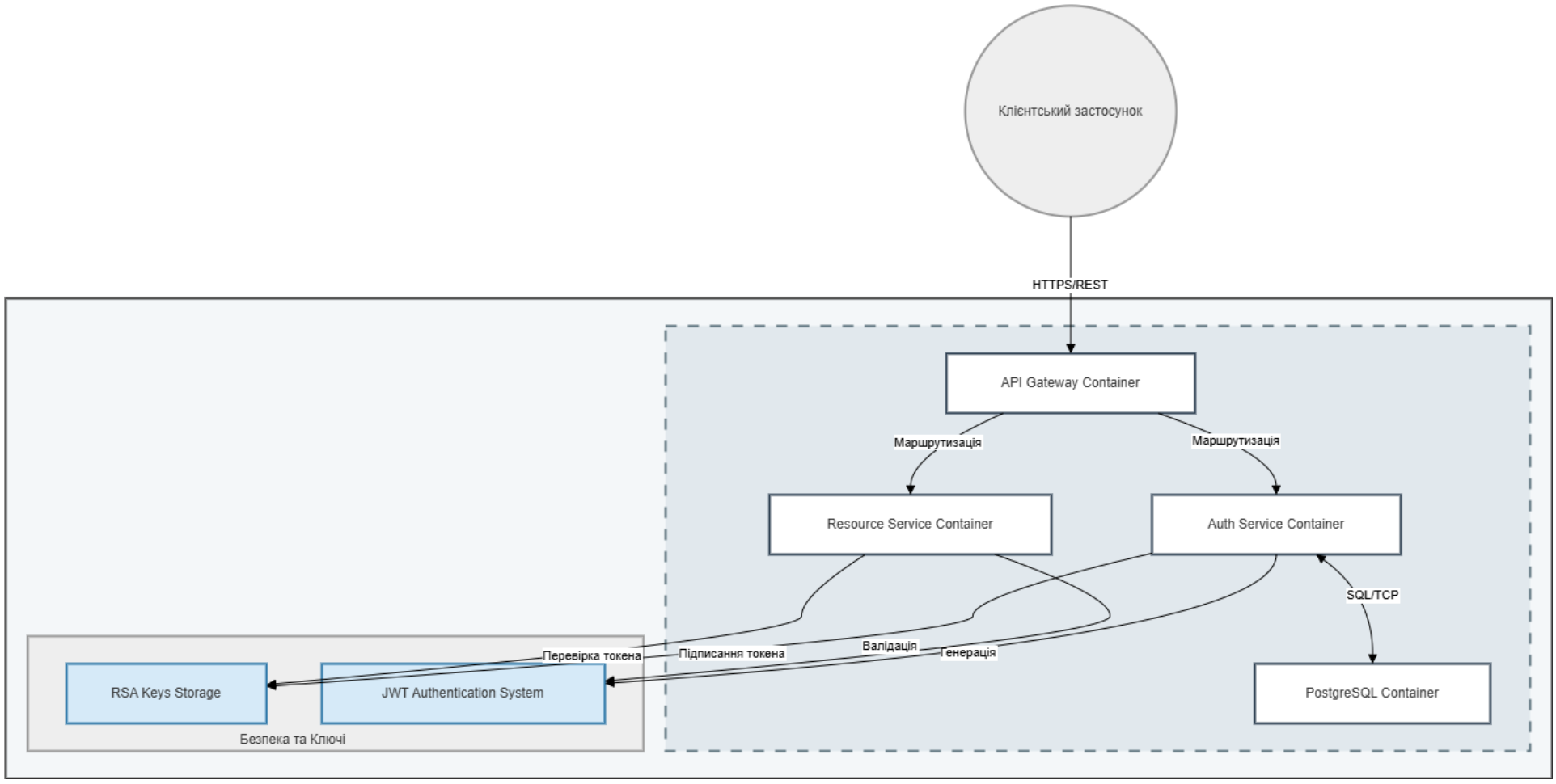


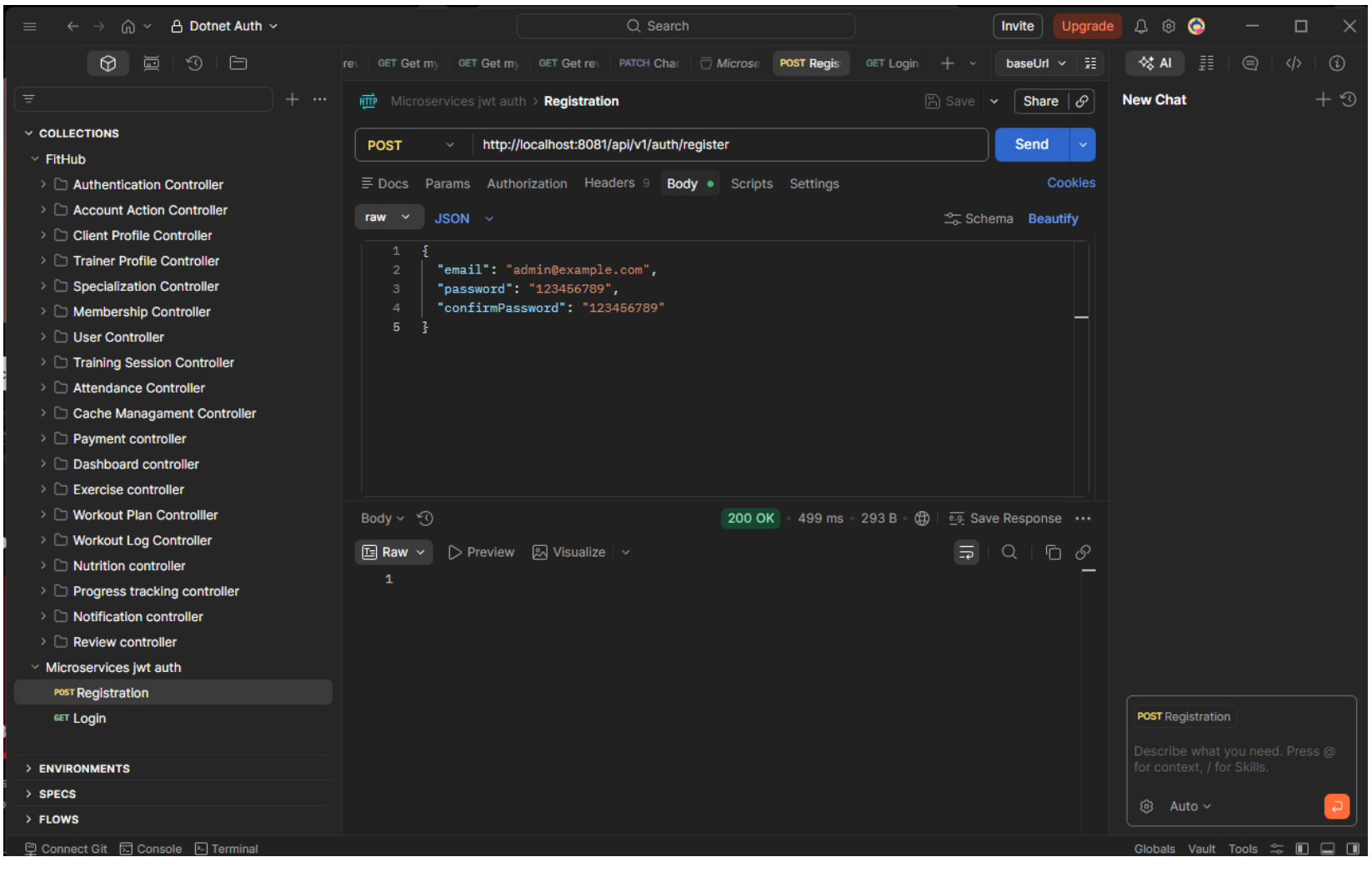

Клієнт
Auth Service
PostgreSQL
Resource Service
Введення логіна та пароля
Перевірка даних
Результат перевірки
alt
[Дані некоректні]
Помилка автентифікації
[Дані коректні]
Генерація та підпис JWT
Передача JWT
Запит із JWT у заголовку
Перевірка підпису, строку, claims
alt
[Токен невалідний]
Помилка 401 Unauthorized
[Токен валідний]
Надання доступу до ресурсу


Користувач/Клієнт
API Gateway
Config Server
Auth Service
Resource Service
PostgreSQL DB
Доставка секретів (RSA/Secrets)
Доставка секретів (RSA/Secrets)
HTTPS (TLS 1.3) запит
Запит на автентифікацію
Перевірка облікових даних
Дані користувача
JWT Токен
Відповідь з токеном
Запит до ресурсу (з JWT)
Валідація токена
Доступ дозволено
Дані ресурсу


Vault / Git Storage (Secrets)
Config Server (Centralized)
Auth Service
Resource Service
Етап ініціалізації (Startup)
Запит зашифрованих властивостей
Повернення (Encrypted YAML)
par
[Отримання налаштувань]
Запит конфігурації (через Bootstrap)
Передача секретів у RAM (DB pass, RSA Private Key)
Запит конфігурації (через Bootstrap)
Передача секретів у RAM (RSA Public Key)
Секрети існують лише в оперативній пам'яті


Клієнт
Auth Filter
JWT Validator (RSA)
SecurityContext
1
HTTP-запит + Authorization Header
alt
[Відсутній Bearer Token]
401 Unauthorized
2
[Токен присутній]
Перевірка підпису та 'exp'
3
alt
[Токен невалідний/прострочений]
Помилка валідації
4
401 Unauthorized
5
[Токен валідний]
Повернення claims та ролей
6
Формування SecurityContextHolder (Stateless)
7
alt
[Недостатньо прав]
403 Forbidden
8
[Достатньо прав]
9
Продовження обробки запиту
Успішна відповідь (Ресурс)
10


Клієнтський застосунок
HTTPS/REST
API Gateway Container
Маршрутизація
Маршрутизація
Resource Service Container
Auth Service Container
SQL/TCP
Перевірка токена
Підписання токена
Валідація
Генерація
RSA Keys Storage
JWT Authentication System
Безпека та Ключі
PostgreSQL Container


Dotnet Auth
Search
Invite
Upgrade
baseUrl
AI
COLLECTIONS
FitHub
Authentication Controller
Account Action Controller
Client Profile Controller
Trainer Profile Controller
Specialization Controller
Membership Controller
User Controller
Training Session Controller
Attendance Controller
Cache Managament Controller
Payment controller
Dashboard controller
Exercise controller
Workout Plan Controlller
Workout Log Controller
Nutrition controller
Progress tracking controller
Notification controller
Review controller
Microservices jwt auth
POST Registration
GET Login
ENVIRONMENTS
SPECS
FLOWS
Microservices jwt auth › Registration
Save
Share
POST
http://localhost:8081/api/v1/auth/register
Send
Docs
Params
Authorization
Headers 9
Body
Scripts
Settings
Cookies
raw
JSON
Schema
Beautify
{
"email": "admin@example.com",
"password": "123456789",
"confirmPassword": "123456789"
}
Body
200 OK
499 ms
293 B
Save Response
Raw
Preview
Visualize
1
New Chat
POST Registration
Describe what you need. Press @ for context, / for Skills.
Auto
Connect Git
Console
Terminal
Globals
Vault
Tools

Microservices jwt auth > **Login**

Save | Share

POST | http://localhost:8081/api/v1/auth/login | Send

Docs | Params | Authorization | Headers 9 | Body | Scripts | Settings | Cookies

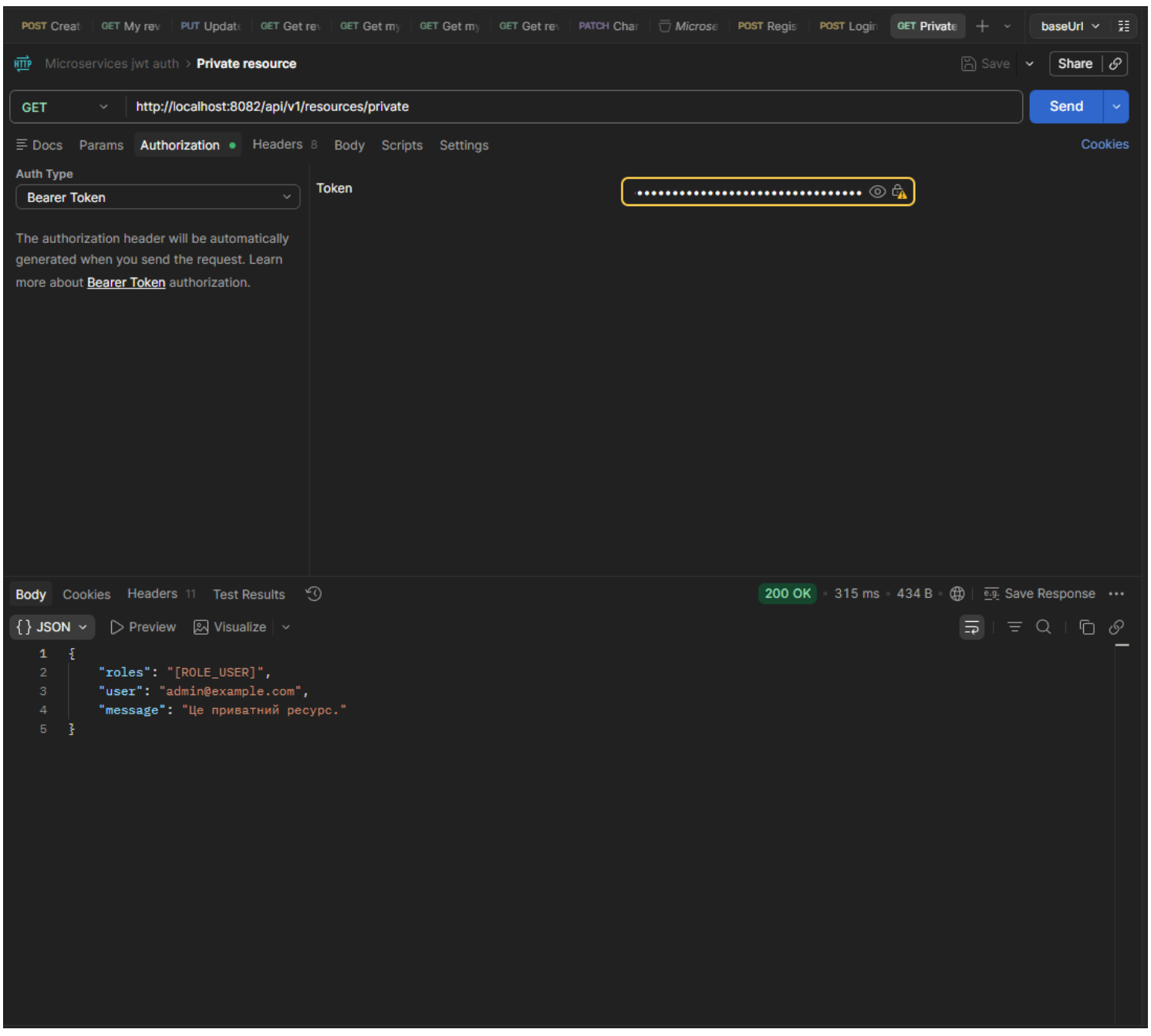
none | form-data | x-www-form-urlencoded | raw | binary | GraphQL | JSON | Schema | Beautify

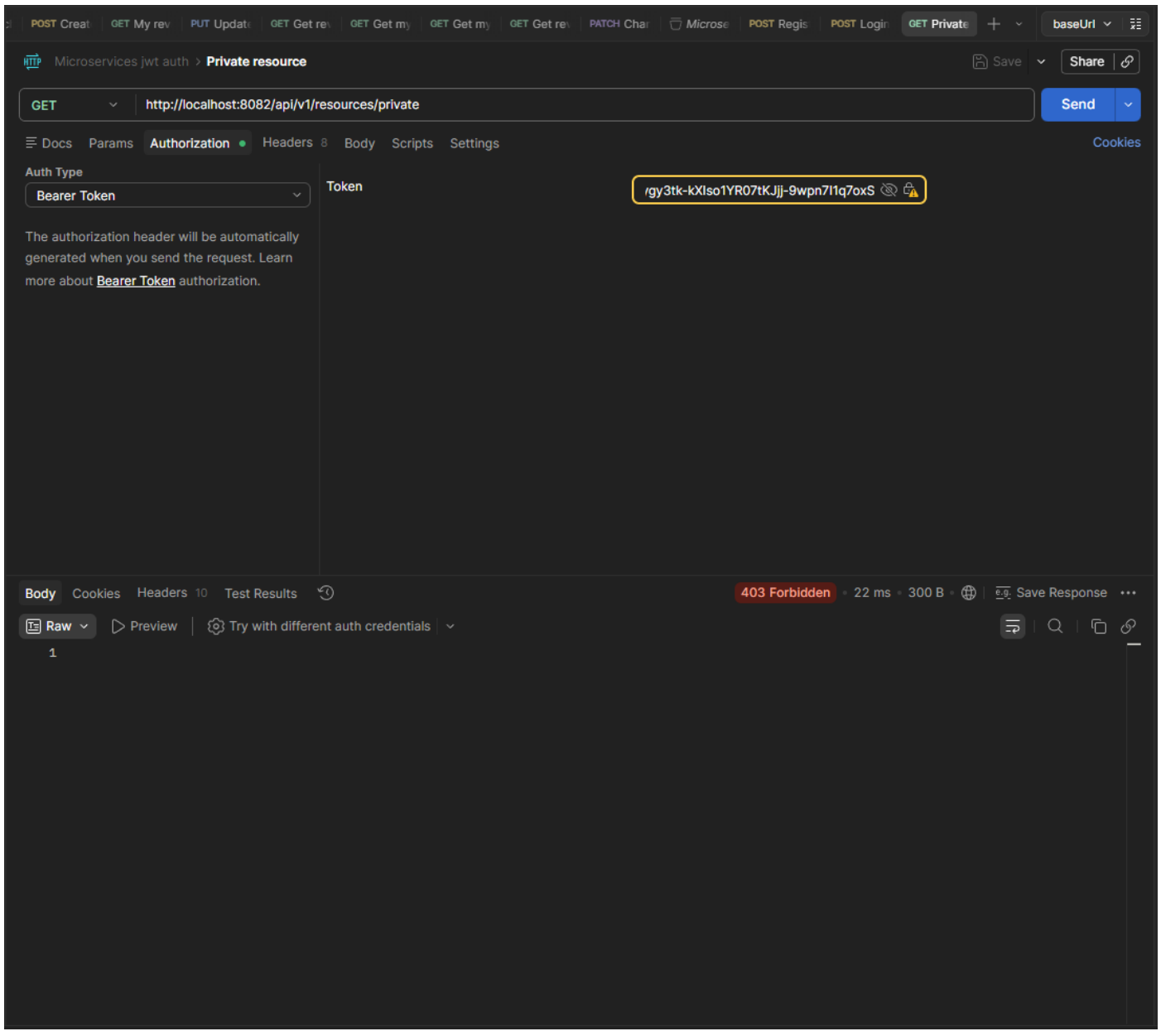
```
{
    "email": "admin@example.com",
    "password": "123456789"
}
```

Body | Cookies | Headers 11 | Test Results | 200 OK | 565 ms | 1.5 KB | Save Response

{ } JSON | Preview | Visualize

```
{
    "accessToken": "eyJhbGciOiJSUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoiQUNDRVNTX1RPS0VOIiwidXNlcklkIjoiMTM1N2M2MzMtMjgyMi00NTllLWI5Y2EtNmUyNmFhZGZkM2IwIiwicm9sZXMiOlsiVVNFUiJdLCJzdWIiOiJhZG1pbkBleGFtcGxlLmNvbSIsImlhdCI6MTc3Njk0Njk2MiwiZXhwIjoxNzc3MDMzMzYyfQ.
        BDTnZOSBdX3mWlKoASzh5xiHMZLhReosvPYTqPn8uJ7g_u4-JcuBculWfTvzKwwEsYj2FX3mjzOB6LCyyaGmlVd9tqUSxuAh9C1zbRM4KvbdUGu2vJZyGhptwYXU1Owxfz
        12qmL00TMC337qBlAFCxTKWU9_bFCjPSWCuu0T5yxy37KDQRwgZUt4c5mKFUMb6jMHYNjbtJFWag7_VGB1kMyu5W7jTjbfV9HMzAuIs22jJBwOP7XX0JQm66K9YKso9eWU
        9TzAFvKri78GdlcaXt2-gGLIlA17J7-7rZjdqa89FtZ5HDygy3tk-kXIso1YR07tKJjj-9wpn7I1q7oxSw",
    "refreshToken": "eyJhbGciOiJSUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoiUkVGUkVTSF9UT0tFTiIsInVzZXJJZCI6IjEzNTdjNjMzLTI4MjItNDU5ZS1iOWNhLTZlMjZhYWRmZDNiMCIsInJvbGVzIjpbIlVTRVIiXSwic3
        ViIjoiYWRtaW5AZXhhbXBsZS5jb20iLCJpYXQiOjE3NzY5NDY5NjIsImV4cCI6MTc3NzYzMDk2Mn0.
        KotgYUfFR9WTXTcI-6wde8g90N5j515LZByq0sEIrie3TrOTAD8A8B5TWpiFrsgMZQlCKBXcrqF9QLpRE7DTMXmqJ9TgCOlk6UlxgE1RC7n5lobCgVbGrieoHuc63xplP-
        E_YGegWWZIw4Dne-zE8DxBPFA-S2WNsW2AYJt-TP5Xq2QK9E15jPn1gd6RVHWlBsZLw_sOgSw5vUOeZdX18mloQBIJx_0Ps7HSYaaMHWoEsmziDvrEVDVtzI8ZbbshF1ZS
        pbmo7naTJvwXf7edOBfgBbSn0Whuk6xOkdb6WJGI9Ptt51zOVag5ZFoPOnkQzZ-nNNCAMHhkzDdq6zltKw",
    "tokenType": "Bearer "
}
```




POST Creat
GET My rev
PUT Updat
GET Get re
GET Get my
GET Get my
GET Get re
PATCH Char
Microse
POST Regis
POST Login
GET Private
baseUrl
Microservices jwt auth
Private resource
Save
Share
GET
http://localhost:8082/api/v1/resources/private
Send
Docs
Params
Authorization
Headers 8
Body
Scripts
Settings
Cookies
Auth Type
Bearer Token
The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization.
Token
/gy3tk-kXIso1YR07tKJjj-9wpn7l1q7oxS
Body
Cookies
Headers 10
Test Results
403 Forbidden
22 ms
300 B
Save Response
Raw
Preview
Try with different auth credentials
1