НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.428.2

«До захисту допущено»
Завідувач кафедри
_____ Едуард ЖАРІКОВ
«___»_____2025 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інженерія програмного забезпечення інформаційних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Програмна бібліотека для надійної та безпечної обробки API-ключів»

Виконав (-ла):
студент (-ка) ІІ курсу, групи ІП-42мп
Москаленко Владислав Юрійович _____

Керівник:
Професор кафедри ІПІ ФІОТ, д.т.н., професор,
Павлов Олександр Анатолійович _____

Рецензент:
доцент кафедри ІСТ, к.т.н., доц.,
Солдатова Марія Олександрівна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2025 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____2025р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Москаленку Владиславу Юрійовичу

1. Тема дисертації « Програмна бібліотека для надійної та безпечної обробки API-ключів», науковий керівник дисертації Павлов Олександр Анатолійович, д.т.н, професор, затверджені наказом по університету від «06» листопада 2025 р. № 4841-с

2. Термін подання студентом дисертації «15» грудня 2025 р.

3. Об’єкт дослідження – процес створення програмного бібліотеки для надійної та безпечної обробки API-ключів

4. Предмет дослідження – методи безпечної й масштабованої обробки API-ключів

5. Перелік завдань, які потрібно розробити – аналіз існуючих бібліотек та інженерних рішень; формування вимог до програмної бібліотеки; модифікація існуючих інженерних рішень для підвищення швидкості та надійності обробки API-ключів; розробка програмної бібліотеки; дослідження ефективності розробленої програмної бібліотеки.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 1 плакат

7. Орієнтовний перелік публікацій – одна публікація

8. Консультанти розділів дисертації

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

9.Дата видачі завдання «1» вересня 2025 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Аналіз існуючих джерел	01.09.2025	
2	Аналіз існуючих бібліотек та інженерних рішень	10.09.2025	
3	Удосконалення існуючих підходів обробки API-ключів	20.09.2025	
4	Постановка та формалізація задачі	05.10.2025	
5	Розробка архітектури програмної бібліотеки	15.10.2025	
6	Реалізація програмної бібліотеки	01.11.2025	
7	Виконання експериментальних досліджень	10.11.2025	
8	Оформлення пояснювальної записки	20.11.2025	
9	Подання дисертації на попередній захист	26.11.2025	
10	Подання дисертації на захист	15.12.2025	

Студент Владислав МОСКАЛЕНКО

Науковий керівник Олександр ПАВЛОВ

РЕФЕРАТ

Розмір пояснювальної записки – 130 аркушів, містить 10 ілюстрацій, 36 таблиць, 2 додатків, 27 посилань на джерела.

Актуальність теми. Актуальність розробки програмної бібліотеки для надійної та безпечної обробки API-ключів зумовлена потребою у підвищенні безпеки машинної авторизації у розподілених і мікросервісних системах. Сучасні сервіси потребують ефективних рішень для зберігання, оновлення та відкликання ключів без простою системи. Також є необхідність в підвищенні ефективності перевірки ключів для уникнення звернення до централізованої бази даних. Запропонована бібліотека спрямована на усунення недоліків традиційних підходів, що мають обмежену масштабованість і недостатній рівень захисту.

Мета дослідження. Підвищити надійність та ефективність обробки API-ключів шляхом використання розширених JWT-токенів із підтвердженням володіння та без застосування централізованих перевірок у базі даних.

Об’єкт дослідження: процес створення програмного бібліотеки для надійної та безпечної обробки API-ключів.

Предмет дослідження: методи безпечної й масштабованої обробки API-ключів.

Для реалізації поставленої мети **сформульовані наступні завдання:**

– провести аналіз існуючих бібліотек для обробки API-ключів;

– провести аналіз існуючих інженерних рішень обробки API-ключів;

– сформувати вимоги до програмної бібліотеки для забезпечення надійності та безпеки обробки API-ключів;

– модифікувати існуючі інженерні рішення для підвищення швидкості та надійності обробки API-ключів;

– розробити архітектуру бібліотеки обробки API-ключів із підтримкою підтвердження володіння токенів;

– реалізувати обробку ключів використовуючи розподілений кеш, уникаючи постійних звернень до централізованої бази даних;

– реалізувати механізм ротації ключів без простою системи;

– забезпечити можливість інтеграції бібліотеки в екосистему фреймворку Spring;

– оцінити ефективність і безпеку запропонованого підходу експериментальним шляхом.

Наукова новизна результатів магістерської дисертації полягає в тому, що удосконалено інженерні рішення ефективного представлення API-ключів у вигляді розширених JWT-токенів, які підтримують підтвердження володіння, а також удосконалено алгоритм відкликання API-ключів на основі епохального механізму та використання Bloom-фільтрів.

Практичне значення отриманих результатів полягає в тому, що розроблена Java бібліотека може бути інтегрована у будь-яку систему машинної авторизації для безпечної обробки API-ключів в екосистемі фреймворку Spring. Вона забезпечує швидке відкликання, безперервну ротацію та масштабовану взаємодію між сервісами, що підвищує рівень безпеки та надійності системи.

Зв'язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Апробація. Наукові положення дисертації пройшли апробацію на науково-практичної конференції молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ» (SoftTech-2025).

Публікації. Наукові положення дисертації опубліковані в:

1) Москаленко В. Ю., Головченко М.М., Павлов О.А. Програмна бібліотека для надійної та безпечної обробки арі-ключів // Інженерія програмного забезпечення і передові інформаційні технології (SoftTech-2025 Осінь) : матеріали тез доповідей IX

Міжнародній науково-практичній конференції молодих вчених та студентів (м. Київ, 26-28 листопада 2025). – К. : КПІ ім. Ігоря Сікорського, 2025.

Ключові слова: API-КЛЮЧ, МАШИННА АВТОРИЗАЦІЯ, JWT, ПІДТВЕРДЖЕННЯ ВОЛОДІННЯ ТОКЕНОМ, ВІДКЛИКАННЯ КЛЮЧІВ, РОТАЦІЯ КЛЮЧІВ, МАСШТАБУВАННЯ, БЕЗПЕКА.

ABSTRACT

Explanatory note size – 130 pages, contains 10 illustrations, 36 tables, 2 applications, 27 references.

Topicality. The relevance of developing a software library for reliable and secure processing of API keys is due to the need to improve the security of machine authorization in distributed and microservice systems. Modern services require effective solutions for storing, updating and revoking keys without system downtime. There is also a need to improve the efficiency of key verification to avoid accessing a centralized database. The proposed library is aimed at eliminating the shortcomings of traditional approaches, which have limited scalability and an insufficient level of protection.

The aim of the study. Improve the reliability and efficiency of API key processing by using extended JWT tokens with proof of ownership and without the use of centralized database checks.

The object of research: the process of creating a software library for reliable and secure processing of API keys.

The subject of research: methods for secure and scalable API key processing.

To achieve this goal, the **following tasks** were formulated:

– analyze existing libraries for processing API keys;

– analyze existing engineering solutions for processing API keys;

– formulate requirements for a software library to ensure the reliability and security of processing API keys;

– modify existing engineering solutions to increase the speed and reliability of processing API keys;

– develop the architecture of an API key processing library with support for confirming token ownership;

– implement key processing using a distributed cache, avoiding constant access to a centralized database;

– implement a key rotation mechanism without system downtime;

– ensure the ability to integrate the library into the Spring framework ecosystem;

– evaluate the effectiveness and security of the proposed approach experimentally.

The scientific novelty of the results of the master's dissertation lies in the fact that engineering solutions for the effective representation of API keys in the form of extended JWT tokens that support proof of ownership have been improved, as well as the API key revocation algorithm based on the epochal mechanism and the use of Bloom filters has been improved.

The practical value of the obtained results is that the developed Java library can be integrated into any machine authorization system for secure processing of API keys in the Spring framework ecosystem. It provides fast revocation, continuous rotation, and scalable interaction between services, which increases the level of security and reliability of the system.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Computer Science and Software Engineering of the National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute».

Approbation. The scientific provisions of the dissertation were tested at the The scientific provisions of the dissertation were tested at the scientific and practical conference of young scientists and students "SOFTWARE ENGINEERING AND ADVANCED INFORMATION TECHNOLOGIES" (SoftTech-2025).

Publications. The scientific provisions of the dissertation were published in:

1) Moskalenko V. U., Golovchenko M.M., Pavlov O.A. Software library for reliable and secure API-key processing // Software engineering and advanced information technologies (SoftTech-2025 Autumn): materials of abstracts of reports of the IX International Scientific and Practical Conference of Young Scientists and Students (Kyiv, November 26-28, 2025). – K.: Igor Sikorsky Kyiv Polytechnic Institute, 2025.

Keywords: API KEY, MACHINE AUTHORIZATION, JWT, TOKEN OWNERSHIP PROOF, KEY REVOCATION, KEY ROTATION, SCALABILITY, SECURITY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (програмний інтерфейс застосунків) – набір правил, протоколів і специфікацій, що дає змогу одному програмному компоненту взаємодіяти з іншим. API визначає формати запитів і відповідей, підтримувані методи та політики доступу, забезпечуючи стандартизований обмін даними між сервісами або модулями.

IDE – Integrated Development Environment (інтегроване середовище розробки) – програмний комплекс, що об’єднує інструменти для написання, редагування, тестування та налагодження програмного коду. Зазвичай містить редактор коду, компілятор або інтерпретатор, відладчик, підсвітку синтаксису та підтримку систем контролю версій, спрощуючи процес розробки програмного забезпечення.

REST – Representational State Transfer (передача представлення стану) – архітектурний стиль проєктування веб-сервісів, заснований на принципах клієнт-серверної взаємодії, безстанності та уніфікованих інтерфейсах. REST-сервіси використовують стандартні HTTP-методи для виконання операцій над ресурсами, представленими у вигляді URL-адрес.

KMS – Key Management Service (сервіс керування ключами) – спеціалізована система або хмарний сервіс для безпечного зберігання, створення, ротації та використання криптографічних ключів. KMS забезпечує контроль доступу, аудит операцій та апаратний або програмний захист ключа.

DPoP – Demonstration of Proof-of-Possession (доказ володіння ключем) – механізм, який підтверджує, що клієнт, який надсилає токен, справді володіє криптографічним ключем, прив’язаним до цього токена. DPoP запобігає атакам повторного використання та крадіжці токенів, оскільки кожен запит додатково підписується окремим ключем клієнта.

HTTP – HyperText Transfer Protocol (протокол передавання гіпертексту) – основний протокол обміну даними у веб-мережі, що визначає формат та правила

передачі запитів і відповідей між клієнтом і сервером. HTTP підтримує різні методи взаємодії та використовується для побудови більшості сучасних веб-сервісів.

JWT – JSON Web Token (токен у форматі JSON) – компактний формат передачі даних між сторонами у вигляді цифрово підписаних токенів. JWT містить заголовок, набір атрибутів та підпис, що дозволяє безпечно передавати інформацію про аутентифікацію, авторизацію та інші атрибути без необхідності постійних запитів до бази даних.

IAM – Identity and Access Management (керування ідентичностями та доступом) – набір технологій і процесів, призначених для створення, керування та контролю доступу користувачів або сервісів до ресурсів. IAM-системи забезпечують аутентифікацію, авторизацію, визначення ролей, політик доступу, а також аудит дій для гарантування безпеки в організаціях і хмарних середовищах.

ВСТУП

У сучасних умовах активного розвитку вебсервісів, мікросервісних архітектур і розподілених систем проблема безпечної авторизації через API залишається актуальною. Більшість систем взаємодіють за допомогою API-ключів, однак традиційні підходи до їх перевірки часто є централізованими, що призводить до затримок, надмірного навантаження на базу даних і зниження масштабованості систем. Додатковою проблемою є складність ротації ключів без простою сервісів, а також відсутність універсального механізму підтвердження володіння ключем, який би мінімізував ризик їх компрометації.

Світові тенденції розв’язання цих проблем спрямовані на використання короткотривалих токенів, цифрових підписів і механізмів підтвердження володіння API-ключем, що дозволяють забезпечити безпеку доступу без потреби у централізованих перевірках. Проте відсутні універсальні бібліотеки, які поєднують ці підходи в ефективне, незалежне від конкретної платформи рішення для керування API-доступом.

Актуальність теми зумовлена необхідністю підвищення надійності, масштабованості та безпеки процесів генерації, перевірки та ротації API-ключів у сучасних мікросервісних системах. Для досягнення цього передбачається реалізувати підтримку токенів на основі розширених JWT-токенів із підтвердженням володіння (DPoP), забезпечити можливість динамічної ротації ключів без простою системи, а також мінімізувати кількість централізованих перевірок за рахунок криптографічних методів та локального кешування результатів перевірки.

Мета дослідження – підвищити надійність та ефективність обробки API-ключів шляхом використання розширених JWT-токенів із підтвердженням володіння та без застосування централізованих перевірок у базі даних.

Об’єкт дослідження – процес створення програмної бібліотеки для надійної та безпечної обробки API-ключів.

Предмет дослідження – алгоритмічне та програмне забезпечення для безпечної й масштабованої обробки API-ключів.

Заплановано виконати такі основні завдання:

- провести аналіз існуючих бібліотек для обробки API-ключів;
- провести аналіз існуючих інженерних рішень обробки API-ключів;
- сформувати вимоги до програмної бібліотеки для забезпечення надійності та безпеки обробки API-ключів;
- модифікувати існуючі інженерні рішення для підвищення швидкості та надійності обробки API-ключів;
- розробити архітектуру бібліотеки обробки API-ключів із підтримкою підтвердження володіння токенів;
- реалізувати обробку ключів використовуючи розподілений кеш, уникаючи постійних звернень до централізованої бази даних;
- реалізувати механізм ротації ключів без простою системи;
- забезпечити можливість інтеграції бібліотеки в екосистему фреймворку Spring;
- оцінити ефективність і безпеку запропонованого підходу експериментальним шляхом.

Результати дослідження мають практичну цінність для створення засобів авторизації в системах, що потребують надійного та ефективного управління API-доступом.

1. АНАЛІЗ СУЧАСНИХ ДОСЛІДЖЕНЬ В ОБЛАСТІ ОБРОБКИ API-КЛЮЧІВ

1.1 Аналіз предметної області

У сучасних інформаційних системах, особливо у середовищі мікросервісної архітектури, однією з найважливіших задач є забезпечення безпечного обміну даними між сервісами. Для цього широко використовуються прикладні програмні інтерфейси (API), які надають можливість програмам взаємодіяти між собою через стандартизовані запити. Ключовим елементом у такій взаємодії є API-ключ – унікальний ідентифікатор, який дозволяє визначити джерело запиту та перевірити його достовірність. API-ключ застосовується для контролю доступу, обмеження частоти запитів, моніторингу використання сервісу та запобігання несанкціонованому доступу [1].

З розвитком розподілених і хмарних технологій зросла потреба у машинній авторизації, що передбачає аутентифікацію не користувача, а системного компонента чи окремого сервісу. На відміну від користувацької аутентифікації, машинна авторизація реалізується без участі людини, а її головною метою є захист внутрішньої взаємодії між мікросервісами, клієнтськими застосунками, API-шлюзами та зовнішніми інтеграціями. Це вимагає створення гнучких і безпечних механізмів перевірки достовірності запитів, які здатні працювати у середовищі з високим навантаженням і мінімальними затримками.

Надійна обробка API-ключів включає кілька взаємопов’язаних аспектів – зберігання, валідацію, оновлення та відкликання. Ключі, як правило, асоціюються з певним клієнтом або сервісом, що здійснює запит, і забезпечують контроль за його правами доступу. Основні проблеми традиційних систем керування ключами полягають у використанні статичних ідентифікаторів, які з часом можуть бути скомпрометовані через публічні сховища, журнали запитів або уразливі клієнтські застосунки. Окрім того, централізовані перевірки ключів часто призводять до

затримок при масштабуванні та підвищують навантаження на базу даних, що негативно впливає на ефективність системи.

З метою підвищення рівня безпеки та ефективності в сучасних архітектурах застосовуються спеціалізовані системи керування ключами (Key Management Systems, KMS). Вони дозволяють централізовано генерувати, зберігати та оновлювати ключі, контролювати доступ і здійснювати аудит дій із ними. Такі системи відіграють важливу роль у побудові довірених обчислювальних середовищ і часто інтегруються з API-шлюзами, щоб забезпечити єдину політику контролю доступу [2].

Загалом, предметна область обробки API-ключів охоплює комплекс завдань, спрямованих на захист міжсервісної взаємодії, запобігання несанкціонованому доступу та підтримку стабільної роботи розподілених систем. Для цього дослідники й розробники прагнуть створювати програмні бібліотеки, що дозволяють виконувати генерацію, перевірку та ротацію ключів у безпечний і масштабований спосіб. Такі рішення повинні поєднувати криптографічну надійність із високою продуктивністю, щоб відповідати вимогам сучасних хмарних і мікросервісних середовищ [3].

1.2 Аналіз літератури

1.2.1 Проведення systematic mapping study

Мета mapping study – систематично охопити наукові та технічні публікації, що стосуються проблематики обробки API-ключів, щоб визначити основні теми, розподіл типів досліджень і виявити непокриті теми у літературі. Далі описано вибір дослідницьких питань, ключові слова й пошукові запити, критерії включення і виключення, результати пошуку з підрахунками та категоризацію.

Таблиця 1.1 – Дослідницькі питання

ID	Дослідницьке питання (RQ)
RQ1	What approaches and models exist for API key management in distributed systems?
RQ2	How is JWT revocation addressed in the literature and what trade-offs are reported?

Продовження таблиці 1.1

ID	Дослідницьке питання (RQ)
RQ3	What research exists on proof-of-possession (PoP/DPoP) mechanisms for API tokens?
RQ4	What operational practices and architectures are proposed for key rotation and zero-downtime rollovers?
RQ5	What empirical evidence exists regarding performance and scalability of API key validation approaches?

Таблиця 1.2 – Пошукові запити

Тематика	Пошукові запити
General API key management	"API key management" OR "API key security"
JWT revocation	"JWT revocation" OR "JSON Web Token revocation"
Proof-of-possession / DPoP	"DPoP" OR "proof of possession" OR "token binding DPoP"
Key rotation	"API key rotation" OR "key rotation zero downtime"
KMS and architecture	"Key Management System" OR "KMS for APIs"
Performance & scalability	"performance API key validation" OR "scalability API"

Таблиця 1.3 – Критерії включення

IC1	Англомовні публікації
IC2	Публікації, що прямо розглядають API-ключі та методи їх обробки
IC3	Публікації, опубліковані після 2015 року

Таблиця 1.4 – Критерії виключення

EC1	Дублікати публікацій, знайдені в різних джерелах (в такому випадку розглядати лише одну версію)
EC2	Публікації лише по користувацькій авторизації без фокусу на машинній авторизації
EC3	Публікації, що є очевидними дублікатами

Таблиця 1.5 – Кількість публікацій за пошуковими запитами

Пошуковий запит	Кількість знайдених публікацій
"API key management" OR "API key security"	223
"JWT revocation" OR "JSON Web Token revocation"	24
"DPoP" OR "proof of possession" OR "token binding DPoP"	957
"API key rotation" OR "key rotation zero downtime"	15
"Key Management System" OR "KMS for APIs"	506
"performance API key validation" OR "scalability API"	53

Таблиця 1.6 - Категорії публікацій

Категорія	Опис	Кількість релевантних публікацій
Solution proposals	Нові методи або алгоритми для обробки API ключів	49
Evaluation research	Практичні оцінки, експерименти, заміри і порівняння	30
Experience papers	Опис впроваджень у виробництві, приклади використань	20
Philosophical	Огляди, систематизація проблеми	17

Таблиця 1.7 – Тематика досліджень обробки API ключів

Тематика	Опис	Кількість публікацій
Revocation mechanisms	Методи відкликання токенів, такі як чорні списки відкликаних токенів, а також структури такі як Bloom фільтри	8
DPoP / Token binding	Підходи підтвердження володіння токеном, клієнтські ключі прив’язані до токенів	35
KMS & infrastructure	Архітектури системи управління секретами, інтеграція з хмарними провайдерами	20

Продовження таблиці 1.7

Тематика	Опис	Кількість публікацій
Operational rotation	Кращі практики ротації ключів із нульовим часом простою	23
Performance studies	Оцінка впливу методів валідації на продуктивність обробки токенів включаючи такі метрики як пропускна здатність та затримка валідації токена	12
Threat analysis & reviews	Аналіз потенційних безпекових загроз таких як втрата втрата токена та перехоплення мережевого трафіку	18

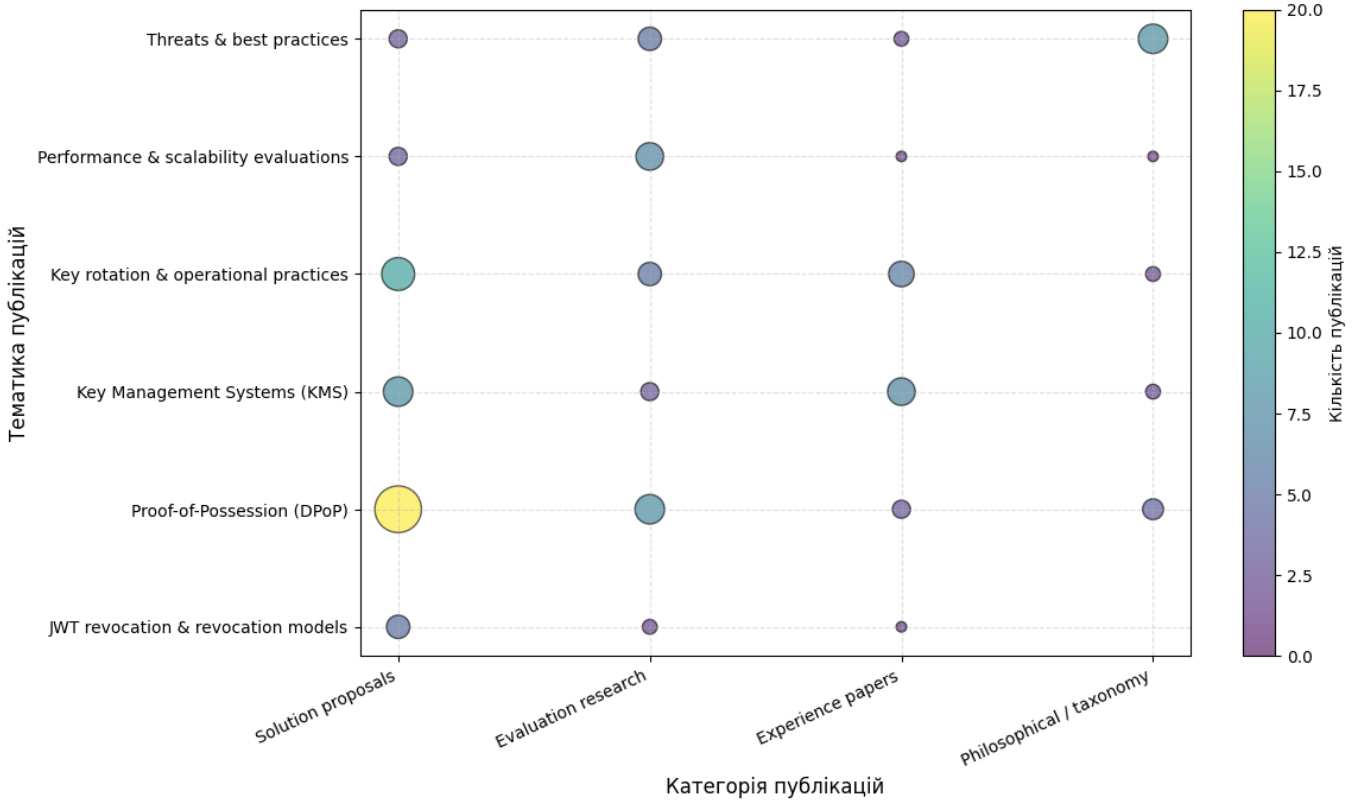


Рисунок 1.1 – Бульбашкова діаграма кількості публікацій за тематикою та категоріями

Отриманий розподіл між тематиками та категоріями демонструє переважання досліджень типу solution proposals, що свідчить про значну активність у розробленні

нових методів і технічних рішень для безпечної обробки API-ключів. Значна частка робіт належить до evaluation research, де автори оцінюють продуктивність та ефективність запропонованих підходів, особливо у контексті підтвердження володіння токеном (DPoP) і моделей ротації ключів. Experience papers вказують на практичну зацікавленість індустрії у впровадженні таких рішень, зокрема в області керування ключів. Категорія philosophical охоплює систематизацію знань, огляди загроз та аналітичні дослідження, що формують теоретичну основу подальших робіт. У цілому результати вказують на те, що наукова спільнота зосереджена насамперед на розробці практичних і технологічних рішень, тоді як аналітичні й узагальнювальні дослідження залишаються поки поза увагою, що створює потенціал для подальших систематичних порівнянь і стандартизації підходів у сфері машинної авторизації API.

1.3 Аналіз сучасного стану предметної області

У сучасних розподілених інформаційних системах, що базуються на мікросервісній архітектурі, API (Application Programming Interface) є ключовим інструментом взаємодії між компонентами програмного забезпечення. Безпечна та надійна обробка API-ключів виступає фундаментальним елементом у забезпеченні довіри між сервісами та контролі доступу до ресурсів. API-ключ виконує функцію машинного ідентифікатора, що дозволяє сервісам аутентифікуватися без участі користувача, тому його захист напряму впливає на цілісність і безпечність усієї системи [4].

Традиційні підходи до зберігання та валідації API-ключів, які передбачають централізовану перевірку в базі даних, поступово втрачають ефективність у масштабованих системах. Така архітектура створює збільшує затримки запитів і ускладнює виконання ротації або відкликання ключів у реальному часі. Вразливість статичних ключів підтверджується численними інцидентами витоку, що відбувалися через публічні сховища коду, журнали запитів чи незахищені мобільні застосунки.

Найбільшою проблемаю є неправильне управління ключами та токенами, що належить до основних джерел витоків даних у корпоративних і хмарних API [5].

Одним із найпоширеніших способів підвищення безпеки є перехід від статичних API-ключів до токенів, що мають обмежений строк дії. Серед найвідоміших рішень – JSON Web Token (JWT), який забезпечує можливість аутентифікації без постійних звернень до бази даних. Однак використання JWT створює нові проблеми, зокрема відсутність вбудованого механізму відкликання токенів, що робить критично важливим питання керування станом авторизації. Існують різні стратегії вирішення даної проблеми, зокрема використання епохальних списків відкликаних токенів або Bloom-фільтрів для масштабованої перевірки відкликаних токенів без зниження продуктивності [6].

Іншим напрямом розвитку є механізми підтвердження володіння токеном (DPoP), які доповнюють токен криптографічним доказом володіння приватним ключем. Це дозволяє запобігти використанню перехоплених токенів, адже кожен запит підписується унікальним ключем клієнта. DPoP розглядається як стандартний спосіб захисту токенів у системах машинної авторизації. Використання PoP-механізмів у поєднанні з JWT або OAuth 2.0 поступово стає основою для побудови безпечних API, зокрема у хмарних платформах [7].

Паралельно активно розвиваються системи керування ключами, що забезпечують централізовану генерацію, шифрування, зберігання та ротацію ключів. Такі сервіси, як AWS KMS та Azure Key Vault, інтегруються безпосередньо з API-шлюзами, надаючи можливість безпечного обміну криптографічними матеріалами між сервісами. Основною тенденцією є перехід до децентралізованої валідації ключів, коли ключ перевіряється локально, але при цьому зберігається можливість централізованого відкликання через публічні реєстри або журнали ротації [8].

Сучасні дослідження акцентують увагу на проблемі динамічної ротації ключів без простоїв системи. Такі механізми дозволяють оновлювати пари ключів під час роботи сервісів без порушення їхньої працездатності, що є критично важливим для

мікросервісних середовищ і CI/CD-пайплайнів. Додатково досліджуються моделі ефемерних ключів, які зменшують ризик компрометації за рахунок обмеженого часу життя кожного ключа. Важливим напрямом також є інтеграція з секретними сховищами, які забезпечують автоматичну ротацію та політики доступу на рівні контейнерів або оркестраторів [9].

Таким чином, сучасний стан предметної області характеризується переходом від статичних, централізованих моделей обробки ключів до динамічних, децентралізованих і криптографічно обґрунтованих рішень, орієнтованих на масштабованість і низьку затримку перевірок. У науковій і технічній літературі формується консенсус, що майбутні системи машинної авторизації повинні поєднувати DPoP-механізми, ефективну ротацію ключів і локальні перевірки стану токенів без звернення до центральної бази даних. Це створює основу для подальших досліджень і впровадження бібліотек, здатних забезпечити безпечну, гнучку та продуктивну обробку API-ключів у розподілених середовищах.

1.4 Ефективність обробки API-ключа

Ефективність обробки API-ключів є одним із ключових аспектів побудови надійних систем машинної авторизації у розподілених архітектурах. На відміну від користувацької аутентифікації, у машинній взаємодії між сервісами валідація API-ключів відбувається на рівні кожного запиту. Це зумовлює високі вимоги до продуктивності, масштабованості та мінімізації накладних витрат при перевірці достовірності ключа. В умовах високонавантажених систем навіть незначне збільшення часу перевірки на кожен запит може призвести до суттєвого зниження пропускної здатності сервісу, тому дослідження ефективних методів обробки API-ключів набуває особливої актуальності.

Одним із найпоширеніших форматів, що використовуються для машинної авторизації, є JSON Web Token (JWT), який може виконувати роль розширеного API-ключа. У дослідженні [10] оцінено продуктивність різних стратегій обробки токенів у

системах із розподіленою архітектурою. Автори запропонували новий метод відкликання ключів, який мінімізує час перевірки дійсності за рахунок використання компактних структур даних замість традиційних централізованих чорних списків відкликаних токенів. Результати експериментів показали, що запропонований підхід забезпечує істотно кращу масштабованість і нижчу затримку при обробці запитів порівняно з базовими схемами, у яких кожна перевірка здійснюється через базу даних. Такі висновки є релевантними для API-ключів, оскільки їх валідація також передбачає перевірку стану ключа чи вінчинний чи відкликаний і підпису.

Подальші експериментальні роботи [11] деталізували порівняння накладних витрат між різними підходами до перевірки стану ключа, включно з використанням структури Bloom-фільтра, епохальних-списків і кешування. Дослідження показало, що найбільший вплив на загальний час обробки мають дві операції – перевірка підпису токена і перевірка його чинності. Оптимізація будь-якої з них може скоротити загальну затримку більш ніж на 40 %, особливо у сценаріях, де одночасно обробляються тисячі запитів на секунду. Для API-ключів це означає доцільність використання попередньо кешованих публічних ключів, що зменшує кількість криптографічних операцій у кожному циклі перевірки.

Питання ефективності криптографічних перевірок, що є невід’ємною частиною обробки API-ключів, розглянуто також у роботі [12]. Автори порівняли продуктивність різних алгоритмів підпису JSON Web Token (HMAC, RSA, ECDSA) за параметрами часу створення, перевірки та розміру підпису. Згідно з результатами дослідження, симетричні алгоритми (зокрема HMAC) демонструють найменший час перевірки підпису, проте поступаються асиметричним (RSA, ECDSA) у питаннях гнучкості та безпеки. З практичної точки зору для обробки API-ключів у внутрішніх системах доцільно використовувати симетричні схеми, тоді як для публічних API, де важливі міжорганізаційні взаємодії, перевагу слід надавати асиметричним методам з кешуванням ключів для зменшення затримок.

На основі проведених досліджень можна зробити висновок, що ефективність обробки API-ключів визначається балансом між безпекою та продуктивністю. У системах із великою кількістю запитів оптимальним є поєднання таких підходів:

– локальна валідація ключів, без звернення до централізованої бази;

– використання ключів або токенів з обмеженим терміном придатності для зменшення ризику компрометації;

– кешування публічних ключів;

– застосування компактних структур даних для швидкого визначення відкликаних ключів.

Такий підхід дозволяє досягти високої пропускної здатності системи без втрати безпеки, що є критичним для машинної авторизації між мікросервісами та компонентами API-інфраструктури. З огляду на сучасні тенденції, подальші дослідження мають бути спрямовані на створення бібліотек, здатних автоматично оптимізувати процес валідації залежно від навантаження та політики ротації ключів.

1.5 Способи відкликання API-ключів

Відкликання API-ключів є одним із найважливіших процесів у системах машинної авторизації, оскільки саме воно забезпечує здатність швидко нейтралізувати скомпрометовані або надалі невалідні ключі без порушення роботи решти компонентів системи. У розподілених системах, де обробка запитів здійснюється одночасно на багатьох вузлах, централізоване керування статусом ключів створює значні виклики для продуктивності та надійності. Класичні підходи передбачають використання централізованих реєстрів, у яких зберігаються відкликані ключі або токени, однак такі рішення не масштабуються при великій кількості клієнтів і викликають затримки через необхідність постійних звернень до бази даних. Тому сучасні підходи до відкликання ключів орієнтуються на децентралізовані та кешовані механізми, які дозволяють локальну перевірку статусу ключа без втрати консистентності системи [13].

Одним із ефективних напрямів є використання епохальних механізмів відкликання, у яких кожен ключ або токен асоціюється з певною часовою епохою, а відкликання відбувається шляхом зміни активної епохи. Це усуває потребу у зберіганні великих списків відкликаних ключів і зменшує обсяг перевірок під час валідації. Такий підхід дозволяє досягти практично миттєвого відкликання при низьких накладних витратах, що робить його придатним для високонавантажених API-систем і мікросервісних платформ. Однак недоліком залишається складність управління епохами при великій кількості взаємозалежних ключів, тому його часто комбінують із додатковими механізмами ротації або токенами з обмеженим часом дії [14].

У системах, які використовують JWT як форму API-ключа, відсутність вбудованого механізму відкликання створює потребу у зовнішніх рішеннях. Одним із таких підходів є використання чорних списків, у яких зберігаються ідентифікатори токенів, позначених як недійсні. Проте зі збільшенням кількості користувачів та обсягів трафіку розмір таких списків стає критичним чинником, що знижує ефективність перевірки. З метою зменшення накладних витрат у сучасних дослідженнях запропоновано використовувати структури даних, оптимізовані для швидких перевірок – зокрема Bloom-фільтри, які забезпечують постійну часову складність пошуку та мінімізують використання пам’яті. Такий підхід дозволяє проводити валідацію відкликаних ключів без суттєвого впливу на продуктивність системи [15].

Ще одним напрямом розвитку є децентралізоване відкликання на основі криптографічних доказів або прозорих журналів ротації. Замість централізованої бази даних, інформація про відкликання публікується у формі хеш-дерев або блоків, підписаних довіреними сервісами. Це дозволяє кожному вузлу самостійно перевіряти актуальність ключа без звернення до центрального сховища, що підвищує стійкість до відмов і зменшує ризик компрометації єдиного вузла. Такі підходи активно

досліджуються у контексті децентралізованих ідентифікаторів та архітектур, де швидкість перевірки й автономність є критично важливими параметрами [16].

У хмарних інфраструктурах і сервісах типу Key Management Systems (KMS) відкликання ключів реалізується через централізовані контролери доступу, які оновлюють стан ключів у реальному часі та забезпечують миттєву синхронізацію змін із підключеними клієнтами. Такі системи, як AWS KMS або Azure Key Vault інтегрують механізми політик доступу, автоматичної ротації та відкликання ключів при порушенні безпекових умов. Хоча це рішення залишається централізованим, воно ефективне для корпоративних і хмарних середовищ завдяки високому рівню узгодженості та можливості аудиту операцій із ключами [17].

Таким чином, сучасні тенденції у сфері відкликання API-ключів демонструють перехід від статичних, централізованих моделей до динамічних і децентралізованих архітектур, які поєднують ефективність перевірки, низьку затримку та стійкість до компрометації. Найперспективнішими вважаються підходи, що комбінують локальну перевірку стану ключа, ідентифікатори з обмеженим терміном дії, ефективні структури даних для відфільтрування відкликаних токенів. Такі рішення створюють основу для побудови бібліотек, які забезпечують безпечну й масштабовану обробку ключів у сучасних API-системах [18].

1.6 Масштабування обробки API-ключів

Масштабування обробки API-ключів є критичним завданням у побудові розподілених систем машинної авторизації, оскільки обсяг запитів до API може зростати експоненційно зі збільшенням кількості клієнтів або мікросервісів. Традиційні централізовані підходи, які передбачають перевірку ключів через єдину базу даних або авторизаційний сервер, не здатні забезпечити необхідний рівень масштабованості. Це призводить до збільшення часу обробки, підвищення навантаження на центральні вузли та зниження загальної доступності сервісу. Для подолання цих обмежень у сучасних системах застосовуються децентралізовані та

кешовані схеми перевірки ключів, які дозволяють виконувати валідацію на локальному рівні без звернення до центрального сховища. Такий підхід мінімізує затримки, зменшує кількість мережевих запитів і підвищує стійкість до збоїв центрального вузла [19].

Одним із напрямів підвищення масштабованості є впровадження розподілених кешів і вузлових перевірок. У цій моделі ключі або токени перевіряються на рівні вузлів, розміщених ближче до користувача чи клієнтського сервісу, наприклад, у API-шлюзах. Розподілені кеші такі як Redis, дозволяють тимчасово зберігати результати перевірки або частину стану ключів, знижуючи кількість звернень до бази даних. Такий підхід активно використовується у великих API-платформах і хмарних середовищах, де пропускна здатність системи є визначальним фактором продуктивності. Важливою перевагою є можливість швидкої інвалідації кешу при ротації або відкликанні ключів, що дозволяє підтримувати баланс між швидкодією та актуальністю даних [20].

Суттєвим внеском у масштабування процесу валідації ключів стали асинхронні та неблокуючі моделі обробки запитів, що реалізуються у сучасних фреймворках та бібліотеках для мікросервісної архітектури. Замість послідовної обробки кожного запиту застосовуються реактивні потоки і неблокуючі API, що дозволяють паралельно виконувати перевірку сотень або тисяч ключів. Це особливо актуально для Java-екосистеми, де бібліотеки на базі Project Reactor або Spring WebFlux забезпечують високу продуктивність при одночасному збереженні коректності перевірки токенів. Така модель дає змогу ефективно масштабувати систему горизонтально, розподіляючи навантаження між вузлами без збільшення затримок [21].

Додатковим напрямом масштабування є використання мікросервісної сегментації розподілу ключів. У великих платформах обробка ключів розподіляється між кількома незалежними компонентами: один відповідає за генерацію, інший – за перевірку, а окремі модулі здійснюють аудит і моніторинг використання. Така архітектура дозволяє масштабувати кожну частину системи незалежно, збільшуючи

продуктивність саме в тих точках, де виникають пікові навантаження. Крім того, застосовується розподіл ключів за сегментами, що дає можливість ефективно управляти доступом у багатокористувацьких середовищах та уникати колізій при перевірці [22].

У хмарних екосистемах масштабування обробки ключів досягається за допомогою автоматизованих систем керування ключами (KMS), що підтримують горизонтальне масштабування і синхронізацію станів між регіонами. Такі рішення, як AWS KMS та Google Cloud KMS, забезпечують розподілене зберігання криптографічних матеріалів, швидке оновлення політик доступу та ефективну реплікацію метаданих про ключі. Завдяки цьому можливо забезпечити практично миттєву валідацію запитів навіть у географічно розподілених інфраструктурах. Використання KMS-інтеграцій у поєднанні з вузлоперевіркою дозволяє поєднати централізоване керування політиками з децентралізованим виконанням перевірок, що є основою масштабованих архітектур машинної авторизації [23].

Сучасні дослідження також розглядають автоматичну балансировку навантаження між сервісами перевірки ключів. Для цього застосовуються спеціальні механізми балансування навантаження, які запобігають перевантаженню окремих вузлів. В таких системах обробка ключів відбувається незалежно від аутентифікації користувача, тобто авторизаційні перевірки здійснюються лише на рівні машинних ідентифікаторів. Це дозволяє мінімізувати об’єм стану системи та досягти високого рівня доступності [24].

Отже, масштабування обробки API-ключів базується на поєднанні кількох принципів – локальної перевірки, асинхронної обробки, розподіленого кешування, автоматизованої ротації та глобальної синхронізації станів ключів. Такі підходи дозволяють забезпечити баланс між безпекою та продуктивністю системи, а також гарантувати стабільну роботу сервісів навіть за умов великих навантажень і частих змін ключів у середовищі машинної авторизації.

1.7 Аналіз існуючих програмних рішень

1.7.1 Аналіз Java бібліотеки apikey-authentication-spring-boot-starter

Бібліотека net.skobow/apikey-authentication-spring-boot-starter [25] є відкритим рішенням, призначеним для інтеграції механізму аутентифікації запитів на основі API-ключів у вебзастосунках. Її головна мета полягає в забезпеченні простої та стандартизованої перевірки API-ключів без необхідності ручного налаштування механізмів безпеки. Вона розповсюджується у вигляді Spring Boot Starter-пакета, що дозволяє підключати її як звичайну залежність і автоматично активувати перевірку ключів у вхідних запитах. Після отримання запиту бібліотека перехоплює його, зчитує ключ із HTTP-заголовка та виконує його валідацію. У разі відсутності або недійсності ключа повертається статус 401.

Функціонально бібліотека підтримує основні можливості, необхідні для базової перевірки ключів: автоматичну конфігурацію без додаткових класів, налаштування параметрів через конфігураційний файл, сумісність із системою безпеки Spring і гнучкість у виборі джерела перевірки (файл, база даних або зовнішній сервіс). Вона дозволяє визначати маршрути, що потребують обов’язкової наявності ключа, а також винятки для відкритих ресурсів. Завдяки цьому бібліотека є практичним засобом для швидкої інтеграції контролю доступу у REST-сервіси без значних витрат часу на конфігурацію.

Разом з тим, рішення має низку суттєвих обмежень. Воно не реалізує повного життєвого циклу API-ключів – зокрема, відсутні механізми їх генерації, ротації, відкликання або визначення часу життя (TTL). Бібліотека не підтримує криптографічні методи перевірки володіння ключем, не забезпечує кешування результатів перевірки, не має аудиту використання ключів. Таким чином, вона виконує лише роль фільтра аутентифікації, без реалізації управління ключами на рівні безпеки та масштабованості.

Дана бібліотека може розглядатися як приклад базової реалізації механізму перевірки запитів. Вона демонструє простий, модульний і сумісний із Spring Security підхід, проте не відповідає вимогам до сучасних систем захисту ключів. Для досягнення поставленої в роботі мети потрібне розширення подібного рішення шляхом додавання механізмів криптографічної генерації, доказового володіння (DPoP), локальної перевірки, автоматичної ротації й відкликання ключів, а також системи аналітики та аудиту використання.

1.7.2 Аналіз Java бібліотеки api-key-authentication

Бібліотека 42BV/api-key-authentication [26] є відкритим рішенням для Java-екосистеми, створеним з метою спрощення інтеграції аутентифікації запитів на основі API-ключів у REST-сервіси. Її головна функція полягає в централізованій перевірці ключів без необхідності ручного додавання валідації в кожен контролер. Бібліотека інтегрується у систему безпеки застосунку, де на рівні обробки запитів автоматично зчитує API-ключ із заголовка HTTP, перевіряє його дійсність і, у разі успішної валідації, створює об’єкт аутентифікації, який визначає користувача або клієнтський сервіс. Якщо ключ недійсний, запит відхиляється з помилкою 401.

Основною перевагою бібліотеки є її гнучкість і розширюваність. Вона дозволяє розробнику самостійно визначати джерело та спосіб перевірки ключа – від локальної бази даних до зовнішніх систем чи кешів. Завдяки підтримці стандартних механізмів авторизації на основі ролей або анотацій, бібліотека легко інтегрується у вже наявну політику безпеки проєкту. Також реалізовано можливість змінювати ім’я HTTP-заголовка, у якому передається ключ, що забезпечує сумісність із різними API-інтерфейсами. Таким чином, рішення забезпечує ефективну й просту в користуванні основу для впровадження перевірки доступу через API-ключі у корпоративних або мікросервісних системах.

Разом із тим, бібліотека має обмеження, що не дозволяють вважати її повноцінною системою управління ключами. Вона не забезпечує механізмів

генерації, ротації, відкликання чи встановлення часу життя ключів. Відсутні засоби доказу володіння, криптографічного підпису запитів, кешування перевірок і аудит використання ключів. Таким чином, рішення виконує лише функцію перевірки достовірності ключа, не охоплюючи більш широкі завдання його життєвого циклу.

В підсумку можна зазначити, що дана бібліотека може розглядатися як базовий приклад реалізації перевірки валідності ключа. Вона забезпечує мінімальні витрати ресурсів і високу сумісність, однак не відповідає вимогам до систем, що потребують криптографічного захисту, динамічної ротації або доказу володіння ключем. Тому її доцільно розглядати як відправну точку для створення розширеного рішення, у якому буде реалізовано механізми повного життєвого циклу API-ключів, аналітики, прозорого відкликання й локальної валідації без залежності від централізованої бази даних.

1.7.3 Аналіз бібліотеки .Net бібліотеки Authentication.ApiKey

Бібліотека AspNetCore.Authentication.ApiKey [27] є відкритим рішенням для екосистеми .NET, яке реалізує аутентифікацію запитів на основі API-ключів у вебзастосунках, побудованих із використанням ASP.NET Core. Вона призначена для швидкого впровадження перевірки ключів у стандартний конвеєр обробки запитів без необхідності ручного створення фільтрів чи middleware. Основна ідея полягає у наданні легкої схеми аутентифікації, що дозволяє серверу приймати запити лише від клієнтів із дійсними ключами.

Бібліотека підтримує декілька джерел отримання ключа – HTTP-заголовок, параметри запиту або значення у маршруті. Перевірка валідності реалізується через інтерфейс IApiKeyProvider або делегат події OnValidateKey, що дозволяє підключати будь-яке джерело даних: базу даних, кеш або зовнішній сервіс. Після успішної валідації бібліотека формує об’єкт користувача із набором прав, що можуть бути використані для подальшої авторизації. Вона також підтримує кілька схем

аутентифікації одночасно, що дозволяє створювати багаторівневі політики доступу до API.

Функціонально AspNetCore.Authentication.ApiKey забезпечує базові можливості перевірки аутентичності: конфігурацію кількох джерел ключів, гнучку інтеграцію у політики безпеки, генерацію стандартної відповіді у разі помилки та підтримку широкого діапазону версій .NET. Така архітектура робить її зручною для побудови мікросервісів або легких REST-інтерфейсів, де необхідна проста й надійна аутентифікація без складних протоколів.

Разом з тим, бібліотека не охоплює повного життєвого циклу API-ключів. Вона не підтримує генерацію, ротацію, відкликання, обмеження на час життя токена або механізми підтвердження володіння. Також відсутні можливості аудиту, кешування перевірок і криптографічного зберігання ключів. Тому її слід розглядати як ефективне, але базове рішення, орієнтоване насамперед на спрощення інтеграції перевірки API-ключів у .NET-додатках.

Дана бібліотека є прикладом альтернативного рішення поза Java-екосистемою, що демонструє типову реалізацію перевірки ключів на рівні веб-сервера. Вона може використовуватися як базова реалізація для побудови більш складних систем, у яких передбачено керування життєвим циклом ключів, криптографічний захист і аналітику використання.

У таблиці 1.8 наведено порівняльну характеристику функціональності розроблюваної бібліотеки в даній дисертації з аналогами.

Таблиця 1.8 – Порівняльна характеристика бібліотеки з аналогами

Функціональна характеристика	Бібліотека в межах дисертації	.Net Authentication.ApiKey	apikey-authentication-spring-boot-starter	api-key-authentication
1. Перевірка валідності API-ключа	+	+	+	+

Продовження таблиці 1.8

Функціональна характеристика	Бібліотека в межах дисертації	.Net Authentication.ApiKey	apikey-authentication-spring-boot-starter	api-key-authentication
2. Простота інтеграції у веб-сервіси	+	+	+	-
3. Можливість конфігурації відкритих / захищених ендпоїнтів	+	+	+	-
4. Можливість відкликання API-ключа	+	-	-	-
5. Криптографічне підтвердження володіння API-ключем	+	-	-	-

1.8 Постановка задачі

Аналіз сучасних підходів до обробки API-ключів показав, що більшість існуючих рішень не забезпечують оптимального поєднання надійності, безпеки та масштабованості у середовищах машинної авторизації. Тому в даній роботі пропонується розробити програмну бібліотеку, яка забезпечуватиме безпечну, надійну та масштабовану обробку API-ключів для машинної взаємодії між сервісами.

Мета даної роботи полягає у підвищенні надійності та ефективності обробки API-ключів шляхом використання розширених JWT-токенів із підтвердженням володіння та без застосування централізованих перевірок у базі даних.

Для досягнення поставленої мети у роботі необхідно виконати такі завдання:

– провести аналіз існуючих бібліотек для обробки API-ключів;

– провести аналіз існуючих інженерних рішень обробки API-ключів;

– сформувати вимоги до програмної бібліотеки для забезпечення надійності та безпеки обробки API-ключів;

– модифікувати існуючі інженерні рішення для підвищення швидкості та надійності обробки API-ключів;

– розробити архітектуру бібліотеки обробки API-ключів із підтримкою підтвердження володіння токенів;

– реалізувати обробку ключів використовуючи розподілений кеш, уникаючи постійних звернень до централізованої бази даних;

– реалізувати механізм ротації ключів без простою системи;

– забезпечити можливість інтеграції бібліотеки в екосистему фреймворку Spring;

– оцінити ефективність і безпеку запропонованого підходу експериментальним шляхом.

Результатом виконання цих завдань має стати програмна бібліотека, яка поєднує безпеку криптографічних методів із високою швидкодією, забезпечує гнучке керування станом API-ключів, підтримує масштабування без втрати продуктивності та може бути використана як у корпоративних, так і в хмарних системах машинної авторизації.

Висновки до розділу

У першому розділі було проведено комплексне дослідження предметної області, що стосується розроблення програмної бібліотеки для надійної та безпечної обробки API-ключів, які застосовуються у системах машинної авторизації для доступу до API.

На початку розділу виконано змістовний аналіз і дослідження предметної області, у межах якого було визначено основні поняття, терміни та концепції, що характеризують процеси аутентифікації сервісів, перевірки достовірності ключів та управління їхнім життєвим циклом. У результаті узагальнено особливості використання API-ключів у розподілених і хмарних системах, а також виявлено

основні проблеми, пов’язані з безпекою, продуктивністю та масштабованістю таких рішень.

Далі було проведено систематичне дослідження наукових джерел, у якому було сформульовано дослідницькі питання, визначено ключові слова, критерії включення та виключення, здійснено пошук релевантних публікацій у наукових базах та проаналізовано результати. Отримані дані дозволили класифікувати наявні наукові роботи за тематичними напрямами та типами досліджень, що стало основою для подальшого аналізу сучасних методів обробки API-ключів.

У підрозділах, присвячених аналізу сучасного стану предметної області, було узагальнено підходи до ефективної валідації, відкликання та масштабування обробки API-ключів. Проведено огляд технічних рішень і алгоритмічних моделей, спрямованих на підвищення безпеки та швидкодії процесів перевірки ключів. Особливу увагу приділено механізмам відкликання ключів у реальному часі, використанню підтвердження володіння токеном, системам керування ключами і практикам ротації ключів.

Також було розглянуто ефективність обробки ключів у контексті оптимізації криптографічних перевірок і зменшення накладних витрат при авторизації запитів. Аналіз показав доцільність використання локальної валідації, кешування результатів перевірки, короткотривалих ключів та розподілених структур даних для підвищення масштабованості системи.

Проведено аналіз існуючих програмних рішень у сфері керування API-ключами, включно з корпоративними й хмарними платформами, які реалізують часткову підтримку динамічної ротації, відкликання та централізованого контролю доступу. Визначено недоліки таких рішень, що стали передумовою для постановки власної задачі розроблення бібліотеки з покращеними характеристиками безпеки та продуктивності.

У завершальній частині розділу сформульовано постановку задачі, де визначено мету, предмет і об’єкт дослідження, а також окреслено основні завдання, що мають

бути виконані у межах даної роботи. До них віднесено аналіз існуючих підходів до обробки ключів, розробку моделей локальної перевірки та відкликання, створення механізмів ротації без простоїв, забезпечення масштабованості бібліотеки та проведення експериментальної оцінки ефективності розробленого рішення.

Таким чином, перший розділ заклав теоретичну та аналітичну основу для подальшого етапу розроблення програмної бібліотеки, спрямованої на підвищення надійності, безпеки та ефективності обробки API-ключів у сучасних системах машинної авторизації.

2 АНАЛІЗ ПІДХОДІВ ОБРОБКИ API-КЛЮЧІВ

2.1 Підходи до представлення API-ключа

У сучасних системах машинної авторизації метод представлення API-ключа відіграє вирішальну роль у забезпеченні балансу між безпекою, продуктивністю та гнучкістю інтеграції. Вибір способу представлення ключа визначає, як саме система здійснює ідентифікацію, перевірку достовірності та контроль доступу між сервісами. Еволюція методів зумовлена переходом від статичних, простих ключів до більш складних структурованих моделей, здатних забезпечувати криптографічний захист і масштабованість.

Традиційні підходи передбачали використання статичних символьних ключів, які зберігалися у базі даних і передавались у запиті як частина заголовка або параметра. Такі ключі містили лише ідентифікаційну інформацію — наприклад, унікальний рядок, який система перевіряла за допомогою пошуку у сховищі. Цей метод був простим у реалізації, проте мав низку істотних недоліків. По-перше, він не забезпечував достатнього рівня безпеки, оскільки компрометація ключа дозволяла зловмиснику безперешкодно здійснювати запити від імені легітимного клієнта. По-друге, кожна операція перевірки вимагала звернення до централізованої бази даних, що створювало додаткове навантаження та підвищувало затримки в розподілених системах. По-третє, такі ключі не містили контекстної інформації про термін дії, рівень доступу або дозволені ресурси, що ускладнювало реалізацію гнучких політик безпеки.

У межах даної роботи аналізується підхід представлення API-ключа у вигляді розширеного JWT токена з криптографічним підписом, який поєднує властивості самодостатності, компактності та можливості валідації. Такий ключ формується за принципом зашифрованого контейнера, що містить інформацію про ідентифікацію суб’єкта, термін дії, атрибути доступу та додаткові параметри. Основна відмінність цього підходу полягає в тому, що перевірка дійсності ключа здійснюється локально —

шляхом розшифрування та перевірки цифрового підпису, що виключає потребу у зовнішньому запиті до бази даних або авторизаційного сервера. Це не лише підвищує продуктивність системи, але й забезпечує можливість її роботи в умовах розподіленої архітектури, де вузли можуть функціонувати автономно.

Окрім цього, розширене представлення ключа дозволяє реалізувати підхід підтвердження володіння, коли ключ прив’язується до певного сервісу або вузла системи. У цьому випадку токен не лише містить криптографічну інформацію, але й вимагає супровідного підтвердження у вигляді підписаного запиту. Такий механізм значно підвищує стійкість до атак повторного використання, оскільки навіть перехоплений токен не може бути повторно використаний без відповідного криптографічного ключа.

Таким чином, проаналізований метод представлення API-ключа забезпечує поєднання самодостатності, безпеки, продуктивності та масштабованості, що робить його придатним для використання у розподілених мікросервісних середовищах. Він усуває потребу у централізованому зберіганні стану, зменшує навантаження на бази даних, забезпечує можливість локальної перевірки дійсності та створює основу для реалізації сучасних механізмів підтвердження володіння і керування життєвим циклом ключів.

2.2 Інженерне рішення для підтвердження володіння токеном

Одним із ключових аспектів забезпечення безпеки в системах машинної авторизації є не лише перевірка достовірності токена, а й підтвердження факту володіння ним суб’єктом, що здійснює запит. Традиційні методи перевірки зосереджуються виключно на аутентичності токена, тобто на тому, чи був він виданий уповноваженою системою. Однак такий підхід не захищає від атак повторного використання, коли зловмисник може перехопити чинний токен і повторно використати його для доступу до сервісу. Вирішення цієї проблеми забезпечують методи підтвердження володіння токеном, які додають додатковий рівень безпеки,

вимагаючи криптографічного доказу того, що запит дійсно надходить від власника токена.

У традиційних моделях авторизації токен сприймається як самодостатній об’єкт, який можна перевірити за підписом, але який не потребує додаткового підтвердження від власника. У результаті, якщо такий токен буде перехоплений, він може бути використаний будь-ким, хто його має. Цей підхід характерний для систем користувацької аутентифікації, де короткий час життя токена частково компенсує ризики компрометації. Однак у машинній авторизації, де сервіси здійснюють сотні запитів щосекунди, а ключі часто мають триваліший строк дії, такий ризик є значно вищим. Тому постає необхідність у механізмі, який дозволяє перевірити не лише валідність токена, але й приналежність його до конкретного сервісу.

Суть методу підтвердження володіння полягає у тому, що під час кожного запиту система, яка надсилає запит, повинна додатково підписати дані запиту своїм приватним криптографічним ключем, пов’язаним із токеном. Сторона серверу, у свою чергу, виконує перевірку підпису за допомогою відповідного відкритого ключа, який зберігається в токені. Таким чином, навіть якщо токен буде викрадений, без наявності приватного ключа його використання буде неможливим.

Реалізація підтвердження володіння токеном ґрунтується на використанні одноразового підпису запиту, де кожен HTTP-запит супроводжується підписом, сформованим на основі унікальних даних — наприклад, ідентифікатора запиту, часу відправлення та вмісту запиту. Це дозволяє унеможливити повторне використання одного й того самого підпису, навіть якщо запит буде перехоплений.

У рамках даної роботи аналізується вдосконалений підхід підтвердження володіння токеном, який базується на поєднанні криптографічного підпису з динамічними параметрами запиту. Під час видачі токена сервісу також генерується унікальна пара ключів — відкритий і приватний. Відкритий ключ асоціюється із токеном, а приватний зберігається локально у сервісі, який використовує цей токен. При кожному запиті формується хеш від комбінації параметрів запиту

(ідентифікатора, часу, адреси тощо), який підписується приватним ключем і додається до заголовків запиту. Сторона сервера, отримавши запит, виконує перевірку підпису, тим самим підтверджуючи, що запит дійсно походить від суб'єкта, який володіє приватним ключем. Це дозволяє досягти високого рівня захисту навіть у розподіленому середовищі без необхідності зберігати стан сесій.

2.3 Підходи запобігання перевикористання API-ключа з підтвердженням

Однією з найпоширеніших загроз у системах машинної авторизації є повторне використання API-ключа — ситуація, коли зловмисник перехоплює чинний токен або запит та намагається повторно надіслати його до сервера з метою отримання несанкціонованого доступу. Навіть за наявності механізмів підтвердження володіння DPoP (Demostrating Proof-of-Possession) така загроза залишається актуальною, якщо не реалізовано додаткових перевірок, які унеможливлюють використання одного й того ж токена більше одного разу або поза його контекстом. Тому для забезпечення цілісності системи необхідно поєднувати криптографічне підтвердження володіння із механізмами, що гарантують одноразовість запиту та контроль послідовності виконання.

Основна ідея методів запобігання перевикористання токена полягає у тому, що кожен запит має бути унікально ідентифікованим у межах певного часово проміжку, а сервер повинен мати змогу відхилити повторні запити, навіть якщо вони підписані правильним ключем. Для цього використовуються такі концепції, як унікальні ідентифікатори запитів, часові мітки та епохальна перевірка дійсності токенів. Сукупність цих механізмів створює захист, який робить неможливим повторне відтворення запиту поза межами визначеного часу або контексту.

Одним із базових методів є використання одноразових ідентифікаторів запитів. Перед надсиланням запиту клієнт генерує випадкове число або унікальний хеш, який включається до його заголовків і підписується приватним ключем власника токена. Сервер, отримуючи запит, перевіряє підпис і зберігає цей ідентифікатор у

тимчасовому кеші. Якщо надходить інший запит з тим самим ідентифікатором, система розпізнає спробу повторного використання та відхиляє її. Оскільки ідентифікатор має короткий термін життя, це також дозволяє уникнути накопичення надмірних записів у пам’яті, зберігаючи систему ефективною навіть при великому навантаженні.

Іншим важливим методом є використання часових міток, які забезпечують перевірку актуальності запиту у момент його отримання. Кожен запит містить часову позначку, що вказує момент його формування, і підписується приватним ключем. Сервер порівнює вказаний час з поточним, і якщо різниця перевищує допустимий інтервал (наприклад, кілька секунд), запит вважається простроченим і відхиляється. У поєднанні з підтвердженням володіння ключем часові мітки формують надійний часовий бар’єр для авторизаційних операцій.

У межах запропонованої бібліотеки ці принципи об’єднуються у єдиний механізм перевірки, який використовує поєднання ідентифікаторів, часових міток і підтвердження володіння API-ключем. Така система забезпечує криптографічну унікальність кожного запиту: навіть якщо токен ідентичний, новий ідентифікатор і нова мітка часу формують іншу підписану структуру. Внаслідок цього будь-яка спроба повторити раніше відправлений запит буде автоматично відхилена, оскільки підпис не відповідатиме поточним параметрам запиту. Для зберігання даних про використані ідентифіктори передбачається використання ефективних розподілених структур з автоматичним очищенням, що мінімізує витрати пам’яті й дозволяє масштабувати систему без втрати продуктивності.

2.4 Інженерне рішення відкликання API-ключів

Відкликання API-ключів є ключовим елементом системи безпечної їх обробки, оскільки саме цей процес забезпечує термінове припинення дії скомпрометованих або більше неактуальних токенів без необхідності зупинки чи перезапуску системи. Відкликання особливо важливе у випадках, коли API-ключ був виданий довготривало,

або якщо з певних причин втрачено контроль над приватним ключем, який його підписував. Головна мета відкликання полягає у тому, щоб система могла швидко розпізнати та заблокувати використання недійсних API-ключів, зберігаючи при цьому високу продуктивність та масштабованість.

Існує два основні підходи до реалізації відкликання токенів — централізований і розподілений з використанням епох. Централізований підхід передбачає зберігання списку недійсних токенів у базі даних. Кожен запит перевіряється на наявність API-ключа у цьому списку. Якщо він присутній, система відхиляє запит. Такий підхід забезпечує високу точність, однак має суттєві обмеження щодо продуктивності, оскільки вимагає частих звернень до централізованого сховища. У розподілених або мікросервісних системах це призводить до додаткових затримок, перевантаження мережі та потенційних конфліктів між вузлами. Тому централізована модель підходить лише для невеликих або середніх за навантаженням систем, де обсяг токенів, що підлягають відкликанню, відносно невеликий.

На зміну традиційним підходам приходять розподілені підходи відкликання, серед яких особливу увагу заслуговує епохальний підхід. Його суть полягає в тому, що система ділить життєвий цикл токенів на послідовні часові інтервали — епохи. Кожен токен пов’язується з певною епохою, а сервер зберігає лише номер поточної активної епохи. Усі токени, пов’язані з попередніми епохами, автоматично втрачають чинність без необхідності індивідуальної перевірки або синхронізації між сервісами. Цей підхід має суттєві переваги: він виключає потребу у централізованій базі відкликань, усуває затримки, пов’язані з її оновленням, та дозволяє системі реагувати на загрози в режимі реального часу.

Для підвищення ефективності та мінімізації обсягу пам’яті, що використовується для зберігання відкликаних токенів, пропонується застосування компактних структур даних, зокрема Bloom-фільтрів. Bloom-фільтр являє собою ефективну структуру даних, яка дозволяє швидко перевіряти наявність API-ключа у множині відкликаних. Її головна перевага — це надзвичайно мала вимога до пам’яті:

замість повного зберігання токенів зберігаються лише їх хеші, розподілені по бітовому масиву. Перевірка наявності токена у Bloom-фільтрі здійснюється дуже швидко, що дозволяє інтегрувати цей механізм навіть у високонавантажені системи без помітного зниження швидкодії.

Ще одним важливим компонентом системи відкликання є розподілене кешування станів токенів, яке дозволяє ефективно поширювати інформацію про відкликання між різними вузлами системи. Для цього можуть використовуватися високопродуктивні розподілені сховища, такі як Redis, які підтримують швидкий обмін даними та автоматичне оновлення записів. Це забезпечує узгодженість даних між сервісами без необхідності централізованого контролера.

Отже, відкликання токенів у системах машинної авторизації повинно реалізовуватися із врахуванням балансу між безпекою, ефективністю та масштабованістю. Проаналізований підхід, що поєднує епохальний принцип, Bloom-фільтри та розподілене кешування, дозволяє забезпечити швидке й надійне відкликання.

2.5 Підходи до ротації ключів

Ротація ключів є невід’ємною складовою забезпечення безпечної та надійної обробки API-ключів у системах машинної авторизації. Її основне призначення полягає у періодичній заміні криптографічних ключів, які використовуються для підпису або шифрування JWT-токенів, без необхідності зупинки роботи системи. Такий підхід мінімізує ризики, пов’язані з потенційним компрометуванням ключів, підвищує рівень криптографічної стійкості системи та дозволяє підтримувати безперервну роботу сервісів навіть у разі планових оновлень або виявлення інцидентів безпеки.

У традиційних рішеннях ротація ключів часто реалізується вручну або через централізовані механізми без належної синхронізації між сервісами. Це призводить до тимчасової втрати сумісності токенів, коли частина системи вже використовує

новий ключ, а інша ще перевіряє підписи старим. Як наслідок — виникають помилки валідації токенів, відмова в авторизації та потенційне порушення доступності сервісів. Для уникнення таких ситуацій необхідно впроваджувати поступову ротацію ключів, яка передбачає паралельну підтримку кількох версій ключів протягом певного перехідного періоду.

Одним із найефективніших підходів є використання механізму ідентифікації ключів. Кожен токен містить у собі унікальний ідентифікатор ключа, яким він був підписаний. Це дозволяє системі під час перевірки токена автоматично визначити, який саме ключ необхідно використати для валідації підпису. Такий підхід дає змогу зберігати кілька активних ключів одночасно, що робить процес ротації безпечним і прозорим — старі ключі можуть залишатися доступними для перевірки доти, доки всі API-ключі, видані з їх використанням, не втратять чинність.

З метою підвищення надійності та узгодженості процесу ротації у розподілених системах доцільним є використання централізованого або розподіленого сховища ключів (KMS). Таке сховище забезпечує централізований контроль життєвого циклу ключів — від створення і зберігання до ротації та відкликання. У випадку розподіленої інфраструктури KMS може функціонувати у вигляді кількох взаємопов’язаних вузлів, що гарантує відмовостійкість і мінімізує затримки під час обміну ключами. Крім того, за допомогою KMS можна автоматизувати графік ротацій, встановивши політику оновлення на основі часу або кількості підписаних токенів, що дозволяє підтримувати стабільний рівень безпеки без участі оператора.

У сучасних високонавантажених сервісах ефективним є також епохальний підхід до ротації, при якому ключі прив’язуються до певних часових епох. Кожен JWT-токен містить ідентифікатор епохи, що дозволяє системі визначити, який набір ключів був чинним на момент його створення. Зміна активної епохи автоматично переводить систему на новий набір ключів, а попередній залишається доступним лише для перевірки токенів старих епох протягом короткого часу. Такий підхід забезпечує

плавність переходу між ключами та виключає необхідність негайного відкликання всіх попередніх токенів.

Для прискорення операцій перевірки підписів у процесі ротації може застосовуватися розподілене кешування ключів, що зберігає активні ключі у пам’яті вузлів або у високошвидкісному сховищі (наприклад, Redis). Це дозволяє уникнути затримок при зверненні до віддаленого KMS і забезпечує стійкість системи до збільшення навантаження. Кешування особливо ефективне у мікросервісних архітектурах, де кожен сервіс може швидко оновлювати свій локальний набір ключів при зміні активної версії.

Таким чином, підходи ротації ключів мають поєднувати автоматизованість, безперервність та відмовостійкість. Запропонований підхід дозволяє мінімізувати ризики компрометації, зберегти сумісність між версіями токенів і гарантувати стабільну роботу системи під час оновлення ключів.

Висновки по розділу

У другому розділі було проаналізовано напрями дослідження, що спрямоване на розробку програмної бібліотеки для надійної та безпечної обробки API-ключів у системах машинної авторизації. У межах розділу було детально розглянуто підходи, що лежать в основі побудови безпечної інфраструктури управління ключами.

На початку розділу були проаналізовані підходи представлення API-ключів, що використовуються у сучасних розподілених системах. Розглянуто переваги використання компактних JWT-токенів, які базуються на розширеній структурі підписаних даних, що забезпечують як аутентичність, так і контроль над життєвим циклом ключа. Проаналізовано підхід представлення ключів, який дозволяє зменшити залежність від централізованих сховищ і підвищує швидкість обробки запитів.

Далі було розглянуто підходи підтвердження володіння токеном, спрямовані на запобігання несанкціонованому використанню API-ключів. У цьому контексті описано механізми криптографічного підтвердження володіння, що дозволяють кожному клієнту доводити право на використання токена без передачі секретних

даних. Проаналізований підхід забезпечує додатковий рівень безпеки, знижуючи ризик повторного використання токенів у випадку їх перехоплення.

Також у розділі було досліджено підходи запобігання перевикористання API-ключів з підтвердженням володіння ними, які забезпечують захист від атак дублювання запитів. Розглянуто підходи до створення унікальних криптографічних підписів для кожної взаємодії між сервісами та можливість застосування одноразових маркерів, що унеможливлює багаторазове використання одного й того ж JWT-токена.

Особливу увагу було приділено підходам відкликання токенів, як одному з найважливіших аспектів забезпечення безпеки у машинній авторизації. Проаналізовано епохальний підхід до відкликання токенів, який дозволяє швидко знечинити компрометовані токени без централізованого сховища відкликань. Розглянуто також можливість застосування компактної структури даних Bloom-фільтрів для оптимізації перевірок, що значно підвищує швидкодію системи при збереженні високого рівня безпеки.

Крім того, у розділі описано підходи ротації ключів, які забезпечують періодичне оновлення підписуючих ключів без простою системи. Запропоновано підхід із використанням ідентифікаторів ключів, що дозволяє зберігати сумісність між старими та новими токенами під час ротації. Також розглянуто застосування розподіленого кешування та епохального механізму, які гарантують безперервність роботи системи під час оновлення ключів.

Отже, у другому розділі було сформовано цілісну основу для подальшої реалізації програмної бібліотеки, що поєднує безпеку, масштабованість і надійність обробки API-ключів. Отримані результати створюють теоретичну базу для практичної реалізації запропонованих рішень.

3 МОДЕЛЮВАННЯ, КОСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Опис запропонованого рішення

Запропоноване рішення ґрунтується на узагальненні підходів, розглянутих у попередньому розділі, та спрямоване на створення програмної бібліотеки, яка забезпечує безпечну, масштабовану та ефективну обробку API-ключів у розподілених системах. Основою є відмова від традиційної практики централізованих перевірок ключів у базі даних та перехід до моделі, у якій критичні операції аутентифікації виконуються локально, без мережевих затримок і залежності від централізованих сховищ.

У межах бібліотеки API-ключ подається у вигляді розширеного JWT-токена, який містить необхідні атрибути для самодостатньої перевірки. Такий підхід дозволяє сервісам у мікросервісній архітектурі здійснювати локальну валідацію підпису, часу дії та атрибутів ключа за допомогою попередньо отриманих ключів підписування. Обраний формат API-ключа оптимізує коло операцій, що виконуються під час перевірки, зменшує залежність від зовнішніх компонентів і спрощує масштабування системи. Представлення API-ключа у вигляді підписаного токена надає можливість забезпечити цілісність та невідмовність без використання централізованого сховища секретів.

У бібліотеці застосовано підходи підтвердження володіння токеном, що забезпечують перевірку не лише дійсності API-ключа, але й того, що його використовує саме той клієнт, для якого даний API-ключ призначений. Використання криптографічних доказів володіння дозволяє унеможливити виконання запиту з іншим пристроєм або після перехоплення API-ключа. Такий підхід застосовується у користувацькій авторизації, але у бібліотеці його адаптовано для машинної аутентифікації між сервісами, що суттєво зменшує ризики повторного використання украдених ключів у розподілених системах.

Для ефективного відкликання API-ключів запропоновано комбінувати два механізми: епохальний підхід та компактні структури даних. Епохальний підхід дозволяє миттєво зробити всі API-ключі певного періоду недійсними, змінюючи її значення на стороні їх видачі. Такий механізм виключає потребу в централізованому сховищі відкликань і забезпечує миттєве поширення рішення на всі компоненти системи. Додатково бібліотека передбачає можливість використання Bloom-фільтрів для оптимізованої перевірки відкликаних API-ключів, що дозволяє масштабувати систему без збільшення затримок та навантаження.

Розв’язання задачі ротації ключів передбачає використання моделі, у якій ключі підписування можуть оновлюватися без простою системи. Пропонується підтримувати одночасну дію кількох версій ключів підписання та позначати їх спеціальними ідентифікаторами. Це дозволяє правильно перевіряти JWT-токени, видані попередніми версіями ключів, аж до завершення їх терміну дії. Такий підхід забезпечує плавність переходу, сумісність між компонентами та безпомилкову валідацію API-ключів у розподілених середовищах.

Для забезпечення масштабованості бібліотека відходить від централізованих структур і використовує розподілені структури для зберігання допоміжних даних, зокрема кеші авторизаційної інформації. Передбачається можливість інтеграції з розподіленими кешами, що дозволяє уникнути вузьких місць і забезпечити стабільну швидкодію навіть у випадку значного збільшення навантаження. Такий підхід знижує ризик відмов, спричинених перевантаженням централізованої бази даних, і дозволяє бібліотеці стабільно функціонувати у високонавантажених середовищах.

У результаті запропоноване рішення забезпечує можливість локальної валідації ключів, криптографічного підтвердження володіння, швидкого відкликання, безперервної ротації підписувальних ключів і масштабованої роботи без єдиного централізованого вузла. Це формує стійку основу для побудови програмної бібліотеки, яка може бути інтегрована у широкий спектр сучасних API-орієнтованих систем.

3.2 Архітектура програмного забезпечення

Архітектура запропонованої програмної бібліотеки розроблена з урахуванням вимог до масштабованості, надійності та безпечності обробки API-ключів у системах машинної аутентифікації. Бібліотека функціонує як окремий модуль у складі сервісу-постачальника API та забезпечує централізовану валідацію ключів, перевірку дозволів, відкликання API-ключів та ротацію ключів підписання. Високорівневу взаємодію бібліотеки із зовнішніми компонентами наведено на рисунку 3.1.

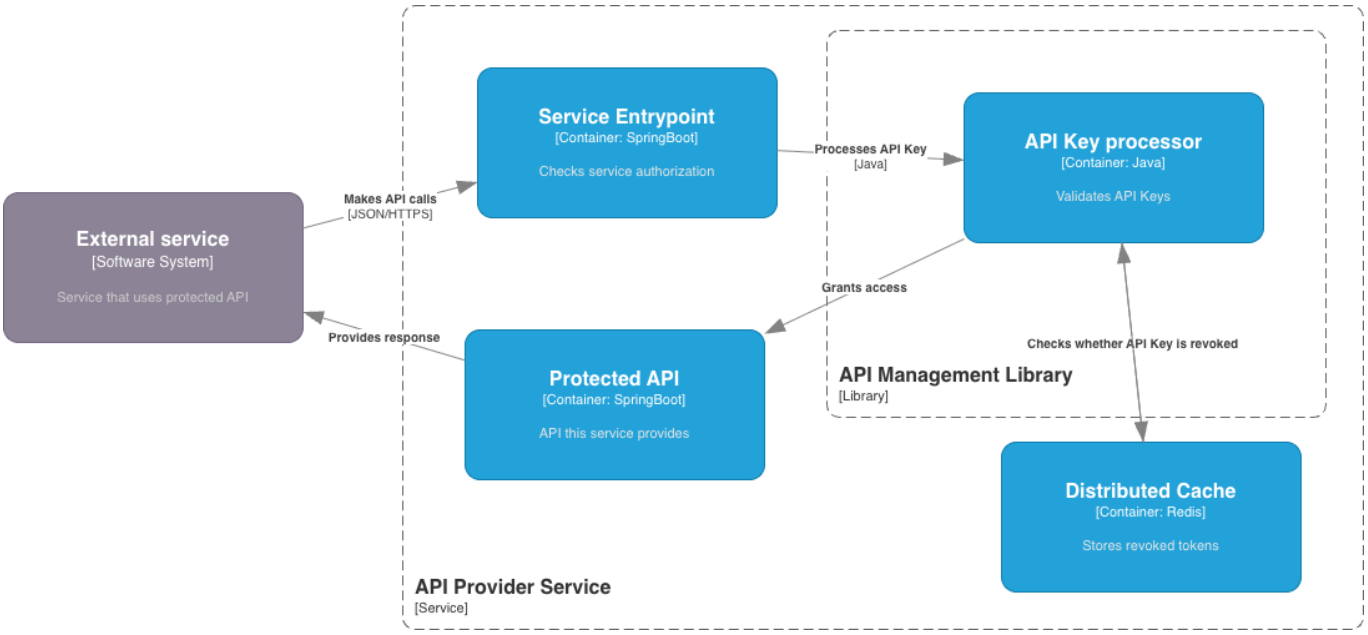


Рисунок 3.1 – Діаграма верхньорівневої архітектури впровадженої бібліотеки

Як зображено на рисунку 3.1, зовнішній програмний компонент, який взаємодіє із захищеним API, надсилає запити до сервісу-постачальника API. Запити спочатку надходять до компонента Service Entrypoint, що виконує первинну авторизаційну перевірку та ініціює процес обробки API-ключа. Після цього ключ передається до програмної бібліотеки API Key Management Library, яка відповідає за повний цикл перевірок: чинність API-ключа, актуальність підписувальних ключів, авторизаційні дозволи клієнта та можливе відкликання ключа.

Бібліотека забезпечує ізоляцію критичних операцій обробки ключів, зменшуючи кількість дубльованої логіки у сервісах та знижуючи ризики помилкового впровадження. У випадку успішної перевірки доступ надається до компонента Protected API, який реалізує бізнес-логіку сервісу-постачальника.

У системі застосовується розподілене кешування, яке слугує високопродуктивним джерелом актуальної інформації про відкликані ключі, підписувальні ключі та додаткові службові дані. Це дає змогу масштабувати бібліотеку горизонтально без втрати синхронізації між різними екземплярами сервісу.

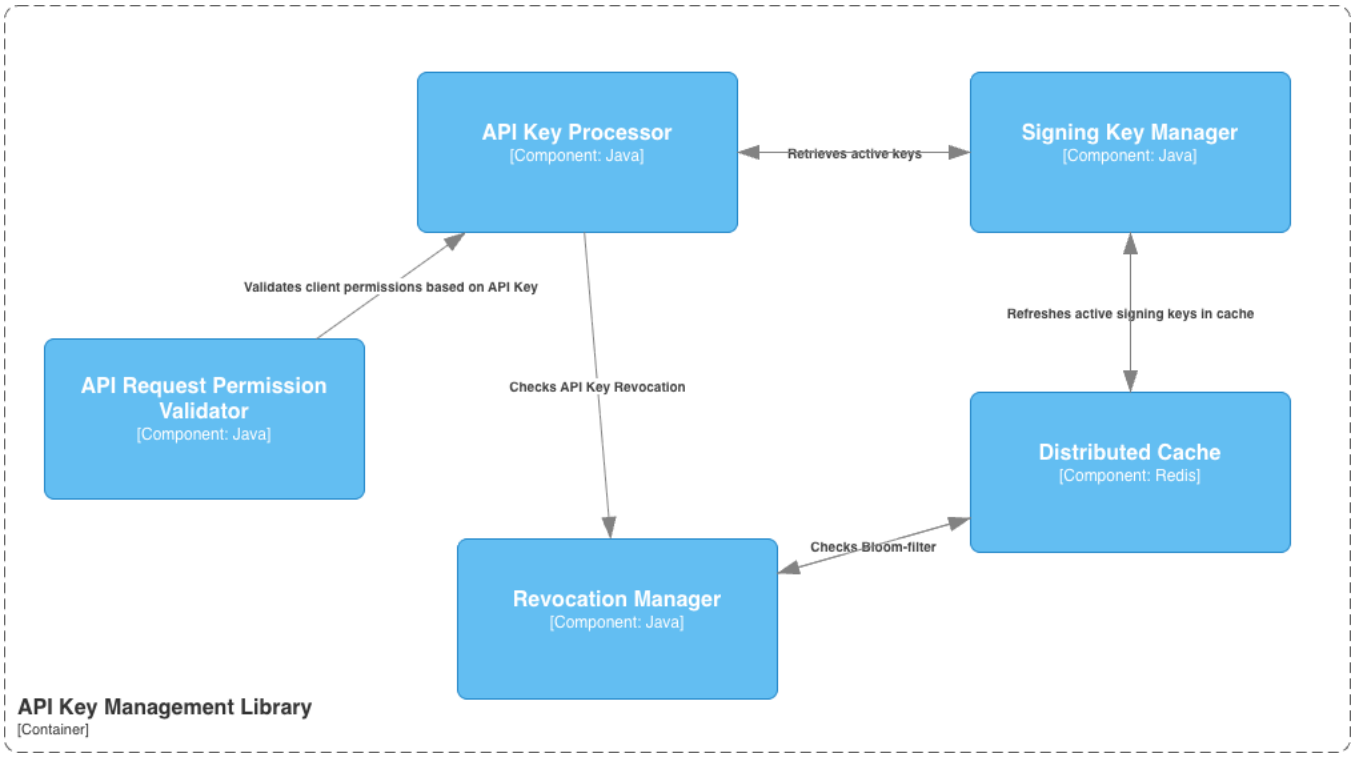


Рисунок 3.2 - Архітектура бібліотеки API Key Management Library

Більш детальну структуру внутрішніх компонентів бібліотеки наведено на рисунку 3.2. Архітектура побудована за модульним принципом, де кожен компонент виконує строго визначену функцію, що формує чіткий поділ відповідальностей і підвищує надійність системи.

Компонент API Key Processor є центральною точкою обробки API-ключів. Він виконує:

– первинний синтаксичний та криптографічний аналіз ключа;

– перевірку підпису за активними підписувальними ключами;

– взаємодію з іншими модулями бібліотеки для підтвердження дозволів, актуальності та не відкликаного стану ключа.

API Key Processor звертається до Signing Key Manager та Revocation Manager, а також передає отримані дані модулю перевірки дозволів.

Компонент Signing Key Manager зберігає та оновлює активні ключі, якими підписуються або верифікуються API-ключі. У системі реалізовано модель ротації підписувальних ключів із гарантованою безперервністю роботи. Менеджер періодично оновлює дані в розподіленому кеші та надає API Key Processor найактуальніші ключі для перевірки підпису.

Компонент API Request Permission Validator відповідає за визначення того, чи має клієнт право виконувати певний тип операції. На основі метаданих API-ключа або пов’язаних з ним атрибутів він перевіряє, чи допускає політика доступу виконання відповідного запиту. Такий підхід мінімізує ризики некоректного доступу та дозволяє централізовано визначати множину наданих дозволів.

Компонент Revocation Manager модуль забезпечує швидку перевірку того, чи відкликаний API-ключ. Він використовує компактні структури даних, зокрема Bloom-фільтри, для зберігання інформації про відкликані ключі. Це зменшує обсяг пам’яті та дозволяє виконувати перевірки з мінімальною затримкою навіть у високонавантажених системах.

Revocation Manager регулярно синхронізує свій стан з розподіленим кешем, що уможливлює роботу в умовах горизонтального масштабування.

Компонент Distributed Cache відповідає за розподілене кешування. Він використовується як високошвидкісне сховище метаданих, зокрема:

– активних підписувальних ключів,

– інформації про відкликані ключі,

– допоміжних службових даних.

Наявність розподіленого кешу дозволяє досягнути високу доступність даних та низьку затримку під час обробки API-ключів, а також мінімізує потребу у зверненні до централізованої бази даних.

Таким чином, архітектура запропонованого програмного забезпечення складається з двох взаємопов’язаних рівнів: інтеграції бібліотеки у сервіс-постачальник API та внутрішньої модульної структури самої бібліотеки. Високорівнева модель демонструє роль бібліотеки як центрального механізму авторизаційної логіки, тоді як деталізована внутрішня архітектура відображає чітку організацію компонентів, орієнтовану на безпечність, ефективність та масштабованість під час обробки API-ключів. Усі компоненти функціонують узгоджено, забезпечуючи надійну роботу системи навіть у високонавантажених середовищах.

3.3 Коструювання програмного забезпечення

У даному підрозділі описано конструювання бібліотеки керування API-ключами: спроєктовані класи, їхні обов’язки, взаємозв’язки та реалізована функціональність. На рисунку 3.3 представлено діаграму основних класів та зв’язків між ними.

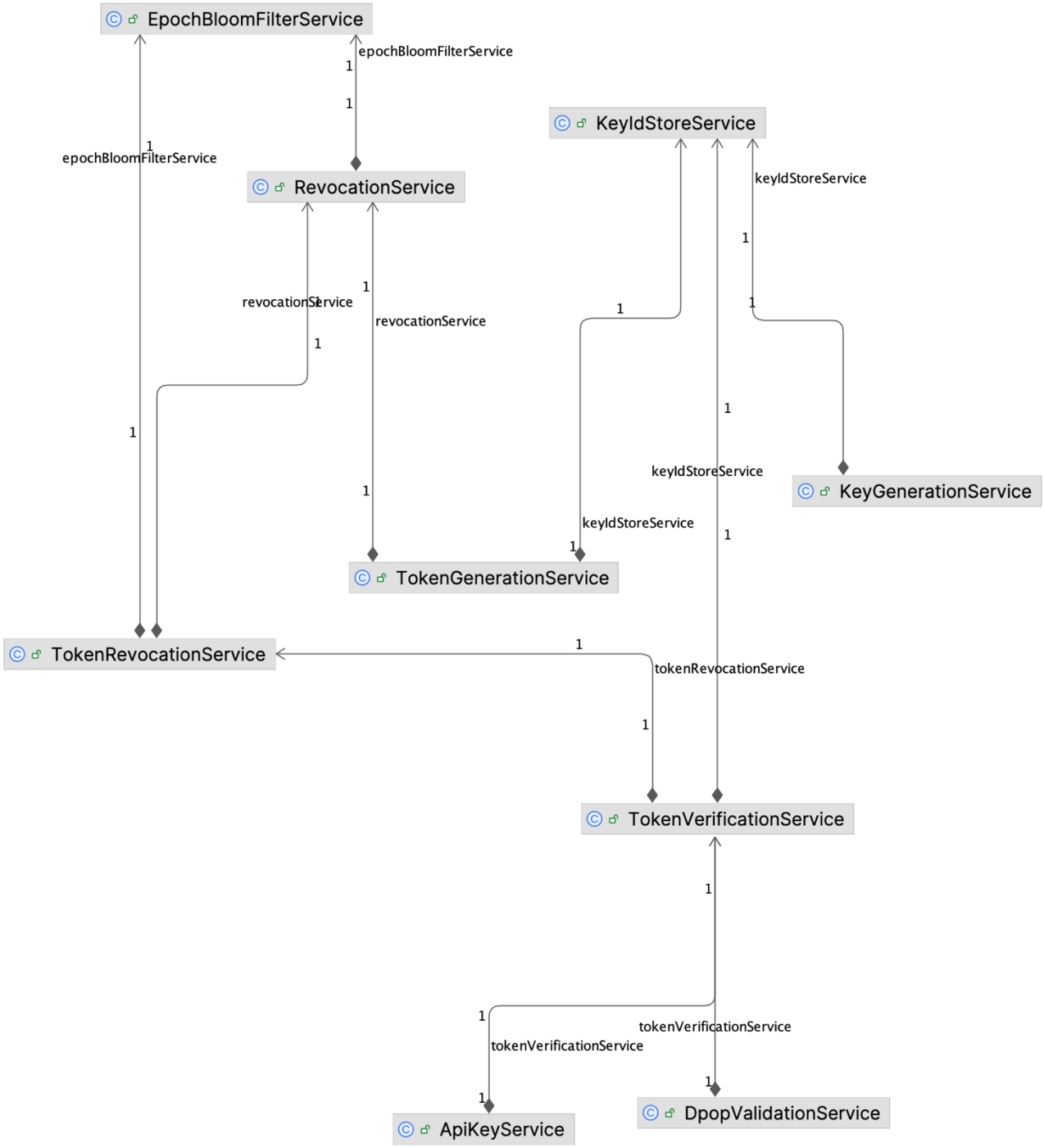


Рисунок 3.3 - Діаграма класів використаних в бібліотеці

Бібліотека сконструйована за принципом чіткої декомпозиції відповідальностей: класи поділені на групи, що відповідають за генерацію та ротацію ключів, зберігання ключів, генерацію і верифікацію API-ключів, відкликання і

забезпечення DPoP захисту. Інтерфейси використовуються для абстрагування залежностей (сховищ ключів, розподіленого кешування, Bloom-операцій), що забезпечує можливість тестування і можливість підміни реалізацій без зміни логіки вищого рівня. Така організація класів підвищує модульність системи.

Перший пакет класів відповідає за генерацію та керування ключами:

– KeyIdStoreService — сервіс, що керує ідентифікаторами підписувальних ключів активних і попередніх і відповідає за атомарну зміну активного ідентифікатора ключа для підтримки ротації ключів без простою системи.

– KeyStorageProvider — абстракція для системи керування ключами підпису. Реалізації можуть зберігати ключі у файловому сховищі, хмарному середовищі або іншому сервісі. Це дозволяє відділити політику керування id від фізичного розміщення ключів.

– KeyStorageException — спеціалізоване виключення для повідомлення про помилки сховища (наприклад, недоступність або некоректні права доступу).

Ця група взаємодіє з компонентами генерації й верифікації токенів: TokenGenerationService запитує ідентифікатор активного ключа у KeyIdStoreService і отримує приватний ключ через KeyStorageProvider. TokenVerificationService запитує публічний ключ для перевірки підпису.

Другий пакет класів надає сервіси генерації і верифікації токенів:

– TokenGenerationService — реалізовує логіку поєднання необхідних атрибутів і підпису JWT-токена. Він отримує ідентифікатор активного ключа підписання (через KeyIdStoreService), завантажує приватний ключ (через KeyStorageProvider), формує необхідні атрибути (jti, iat, exp, rev_epoch, rev_index, scopes, pop тощо) та підписує сформований JWT-токен. Повертає компактний JWT рядок клієнту. Також даний сервіс конфігурується через TokenGenerationConfig.

– TokenVerificationService — зосереджує процес валідації вхідного API-ключа: витягує ідентифікатор ключа підпису (kid) з заголовка, завантажує або бере з кешу відповідний публічний ключ (через KeyStorageProvider і KeyIdStoreService),

перевіряє підпис, обов'язкові атрибути (iss, sub, aud, iat, exp, jti), перевіряє DPoP-структуру та викликає підсистему відкликання для оптимізованої перевірки відкликання. Повертає результат стану API-ключа.

Третій пакет класів відповідає за відкликання API-ключів:

– RevocationConfig — конфігураційний клас, що містить ключі і параметри для роботи з епохами і Bloom-фільтрами.

– EpochBloomFilterService — сервіс, який інкапсулює операції над Bloom-фільтрами, організованими по епохах; відповідає за створення, оновлення, перевірку приналежності ідентифікатора API-ключа до множини відкликаних у поточній або попередніх епохах.

– TokenRevocationService — високорівневий сервіс відкликання, що координує логіку інвалідації JWT-токена: зберігає відкликані API-ключі, додаючи ідентифікатор API-ключа (JTI) у Bloom-фільтр, відповідає за оновлення індексів відкликання та задання rev_index у нових JWT-токенах. TokenRevocationService надає методи перевірки токенів, реалізуючи оптимізований шлях перевірки із використанням rev_epoch.

– RedisStore — інтерфейс для підтримки операцій розподіленого кешу Redis: зберігання епох, індексів, часу тривалості чинності API-ключів, забезпечення атомарних операцій, що важливо під час ротацій і оновлень.

Узгоджена робота цих класів дозволяє проводити масштабоване відкликання з малим обсягом даних в Redis завдяки поєднанню епох і Bloom-фільтрів.

Четвертий набір класів відповідає за перевірку підтвердження володіння токеном:

– DpopConfig — конфігурація для DPoP валідації API-ключів.

– DpopValidationService — сервіс, що виконує перевірку структури DPoP-підтвердження: розбір payload|signature, співставлення метаданих запиту (method, uri, query, timestamp) з підтвердженням володіння, перевірка цифрового підпису за

публічним ключем з атрибутів токена, перевірка на перевикористання через RedisStore. Повертає DpopValidationResult з докладною інформацією.

DPoP-сервіс тісно взаємодіє з TokenVerificationService та з RedisStore. Це гарантує, що підпис запиту відповідає ключу, зазначеному у токені, і що підтвердження володіння не може бути повторно використане у межах часу його дійсності.

Також бібліотека надає допоміжні класи ApiKeyService та ApiRequestPermissionValidator. Вони призначені для роботи з правами доступу у токені: перевірка прав доступу та отримання списку доступних ресурсів. Окремий валідатор може оцінювати, чи дозволено конкретний запит на основі атрибутів токена. Це дозволяє відокремити бізнес-логіку перевірки прав від базової верифікації токена.

Конструювання передбачає чітку модель обробки виключень: помилки доступу до зовнішніх компонентів відрізняються від логічних помилок валідації (протермінований токен, некоректний підпис, відкликаний API-ключ). KeyStorageException використовується для сигналізації помилок сховища; TokenValidationResult повертає семантично зрозумілу причину відхилення (Expired, SignatureError, Revoked, Malformed). Це дозволяє вбудовувати адаптивні політики повторної спроби або логування інцидентів.

Архітектура класів спроектована так, щоб полегшити додавання нових реалізацій компонентів бібліотек. Інтерфейси RedisStore та BloomFilterOperations є мінімальними й атомарними, що дозволяє замінювати реалізації без змін у логіці RevocationService. Така модульність сприяє поступовій еволюції системи без необхідності рефакторингу всього коду.

3.3.1 Вибір мови програмування

Для реалізації програмної бібліотеки було розглянуто кілька сучасних мов, що поширено застосовуються у сфері розроблення серверних систем і засобів безпеки: Java, Python та Node.js (JavaScript/TypeScript). Кожна з цих мов має власну нішу

застосування та специфіку, однак особливості задачі — обробка API-ключів у високонавантажених розподілених системах — зумовлюють необхідність у поєднанні таких властивостей: стабільність, безпечність типів, розвинена криптографічна екосистема, зрілі бібліотеки для роботи з токенами та підтримка промислових стандартів.

Python та Node.js були відкинуті через динамічну типізацію та інтерпретовану природу, що ускладнює однакову поведінку у різних середовищах і збільшує накладні витрати під час інтенсивних криптографічних операцій.

Java, у свою чергу, поєднує такі переваги:

- зріла екосистема криптографічних бібліотек (JCA/JCE, Nimbus JOSE JWT);
- повна сумісність із промисловими фреймворками безпеки (Spring Security);
- стабільна модель пам’яті та потоків для систем реального навантаження;
- кросплатформеність та передбачувана поведінка JVM;
- широка підтримка у корпоративному секторі, що спрощує інтеграцію бібліотеки замовниками.

У результаті зваженого порівняння мова Java була обрана як оптимальна основа для реалізації бібліотеки.

3.3.2 Вибір фреймворку

Після визначення мови програмування було проведено аналіз фреймворків прикладного рівня, зокрема: Spring Framework, Quarkus, Jakarta EE.

Jakarta EE надає широкий стандарт сервлетів, інтерфейси для безпекових модулів, однак вимагає повноцінного контейнера застосунку та не є оптимальною платформою для створення окремих бібліотек без сильної прив’язки до серверного середовища. Quarkus орієнтовані на мікросервісні середовища та забезпечують високу продуктивність, але не мають настільки розвиненої екосистеми, як Spring, і поступаються у зрілості засобів конфігурації та аспектно-орієнтованого програмування.

Spring Framework має низку істотних переваг:

– гнучкий механізм інверсії керування (IoC) та ін’єкції залежностей;

– підтримку модульності через Spring Boot Starter підходи, що дозволяє упаковувати бібліотеку як стартовий модуль;

– розвинуту систему інтеграції з Redis, кешуванням, безпекою та криптографією;

– підтримку спрощеного конфігурування для сторонніх бібліотек, що робить бібліотеку легко інтегрованою у будь-який Spring-застосунок;

– широке поширення у промислових системах, що забезпечує легкість впровадження.

Отже, Spring Framework став найкращим вибором для створення бібліотеки, орієнтованої на корпоративне застосування, гнучку інтеграцію та масштабованість.

3.3.3 Вибір середовища розробки

Середовища розробки, розглянуті у процесі оцінювання: IntelliJ IDEA, Eclipse, а також редактор загального призначення — VS Code.

VS Code поступаються у функціональності рефакторингу та інструментах статичного аналізу коду. Eclipse, хоча й має довгу історію, значно програє IntelliJ IDEA у продуктивності індексації, якості автоматичного доповнення коду та стабільності роботи з великими проєктами.

IntelliJ IDEA володіє такими ключовими перевагами:

– найкращий у класі механізм індексації та автодоповнення Java-коду;

– вбудовані засоби статичного аналізу, підсвічування помилок і рефакторингу;

– інтеграція зі Spring Framework на рівні аналізу конфігурації та залежностей;

– підтримка студентської безкоштовної ліцензії, що робить її доступною для навчальних та дослідницьких проєктів;

– зручність роботи з Git, Docker, Maven/Gradle і моделюванням UML.

Ці властивості дозволяють значно скоротити час розробки, зменшити кількість помилок та забезпечити послідовне застосування патернів проєктування.

3.4 Інтеграція розробленої бібліотеки

Інтеграція розробленої бібліотеки керування API-ключами у Spring Boot середовище передбачає послідовне виконання низки кроків, що забезпечують коректне підключення залежностей, налаштування конфігураційних параметрів, автоматичну реєстрацію необхідних компонентів та включення допоміжних механізмів на приклад для ротації ключів чи підтримки Redis. Впровадження бібліотеки здійснюється згідно з принципами Spring Boot автоконфігурації, що значно спрощує процес адаптації рішення у будь-який існуючий мікросервіс чи корпоративну систему.

Першим етапом інтеграції є підключення залежностей. Додаток має містити саму бібліотеку управління API-ключами та модуль роботи з Redis, оскільки останній використовується для механізмів відкликання токенів і підтримки Bloom-фільтрів. Завдяки Spring Boot залежності цього типу автоматично активують відповідні автоконфігурації, зокрема створення Redis-клієнта у вигляді шаблону, який використовується бібліотекою для виконання операцій над сховищем даних. Підключення залежностей дозволяє Spring знайти та активувати класи автоконфігурації бібліотеки.

Наступним кроком інтеграції є налаштування конфігураційних параметрів у файлі application.properties. Бібліотека передбачає використання групи параметрів, за допомогою яких задається середовище її роботи. Конфігурації включають параметри щодо секретів та ключів підпису, шляхи до директорії ключового сховища, тривалість життя токенів, параметри відкликання, конфігурацію Bloom-фільтрів, специфічні налаштування механізму DPoP, а також параметри доступу до Redis. Таке

конфігураційне оформлення дозволяє здійснити інтеграцію бібліотеки декларативним способом без модифікації коду.

Після налаштування конфігурації відбувається автоматична ініціалізація бінів, що визначені в бібліотеці. Механізм автоконфігурації Spring Boot забезпечує створення таких компонентів, як сервіси генерації та верифікації токенів, сервіс відкликання API-ключів, механізми перевірки DPoP-підтверджень, сервіси роботи з ключами підпису, а також інтеграційні обгортки для Redis та Bloom-фільтрів. Автоконфігурація застосовується лише у тому випадку, якщо користувач проєкту не визначив власні реалізації відповідних сервісів, що дозволяє за потреби розширювати або перевизначати функціональність. Завдяки цьому інтеграція бібліотеки у Spring середовище не вимагає ручного створення базових компонентів, а користувачеві достатньо задати конфігурацію.

У разі, якщо застосунок має підтримувати автоматичну ротацію ключів підпису, потрібно активувати механізм планування задач у Spring. Після цього бібліотека зможе виконувати періодичні дії згідно з визначеним часовим проміжком, генерувати нові ключі, переміщувати попередні ключі у резервний набір та очищувати застарілі записи. Це забезпечує безперервність валідації API-ключів навіть у випадку їх підпису різними ключами в різні моменти часу.

Після підключення бібліотеки до контексту Spring стають доступними REST-інтерфейси, які вона надає. Основний набір включає ендпоінти для управління ключами підпису, що дозволяє отримувати набір публічних ключів, генерувати нові ключі, виконувати ротацію та контролювати стан сховища. Друга група ендпоінтів відповідає за операції з API-ключами у вигляді розширених JWT-токенів: генерацію токенів, їхню валідацію, відкликання, перевірку HTTP-запитів на основі атрибутів токена та виконання верифікації DPoP-доказів. Оскільки контролери надаються бібліотекою автоматично, користувачеві не потрібно реалізовувати власні REST-класи для цих функцій.

У системах, де використовується механізм авторизації на основі дозволів, бібліотека пропонує декілька способів інтегрування цієї логіки у Spring. Розробник може застосовувати анотації над методами або класами для визначення необхідних атрибутів, використовувати реєстр маршрутів для централізованого опису політики доступу або покладатися на інтегрований перехоплювач, який перехоплює HTTP-запити і виконує необхідні перевірки. Перехопювач забезпечує витяг API-ключа, криптографічну перевірку, оцінку прав доступу, а також за потреби — перевірку DPoP-підпису, що дає змогу без додаткового коду посилити захист ресурсів.

Окрему роль у інтеграції відіграє Redis, який забезпечує підтримку відкликань, обробку Bloom-фільтрів та зберігання тимчасових даних. Spring Boot автоматично створює необхідні компоненти доступу до Redis, а бібліотека накладає на них логіку, пов’язану зі зберіганням міток відкликання, підтримкою епох відкликання та виконанням швидких перевірок на факт відкликання токена. За потреби користувач може визначити власну реалізацію сховищ або підключити зовнішні KMS-системи.

У результаті інтеграція розробленої бібліотеки у Spring-середовище здатна відбуватися без значних модифікацій бізнес-логіки застосунку. Після підключення залежностей, визначення конфігурації та забезпечення доступності Redis розробник отримує повністю готову систему керування API-ключами з підтримкою криптографічної генерації, перевірки, відкликання, ротації ключів та DPoP-механізмів. Завдяки автоматичній конфігурації Spring та модульності бібліотеки інтеграція виконується з мінімальними витратами та високою гнучкістю для подальшого розширення функціональності.

Висновки до розділу

У третьому розділі було здійснено комплексне проєктування та конструювання програмного забезпечення для бібліотеки керування API-ключами, що охопило як концептуальні, так і технічні аспекти її реалізації. Спочатку було представлено загальну логіку запропонованого рішення, обґрунтовано вибір механізмів роботи з

API-ключами, ротацією ключів підписання, відкликання та підтвердженням володіння токенів. Далі було розроблено архітектуру програмного забезпечення на двох рівнях абстракції та проаналізовано взаємодію компонентів відповідно до структурних діаграм.

У процесі конструювання системи було сформовано набір класів та сервісів, визначено їхні функції, відповідальності та взаємозв’язки, а також описано внутрішню логіку реалізації ключових модулів без урахування інтеграції зі Spring. Окремо проведено аналіз засобів розробки, у якому обґрунтовано вибір мови Java, фреймворку Spring і середовища Intellij IDEA як оптимальних інструментів для реалізації такого типу системи.

Завершальним етапом у розділі був детальний опис інтеграції розробленої бібліотеки у середовище Spring, де розглянуто механізми підключення залежностей, конфігурації параметрів, автоматичної ініціалізації сервісів, ротації ключів, роботи з Redis та використання REST-інтерфейсів, що надає застосунку повноцінну інфраструктуру для безпечної генерації, валідації та відкликання API-ключів.

4 ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

4.1 Опис експериментального дослідження

Експериментальне дослідження спрямоване на кількісне та якісне оцінювання ефективності запропонованої бібліотеки для безпечної та високопродуктивної обробки API-ключів у порівнянні з традиційним підходом, що передбачає централізовану перевірку ключів за допомогою реляційної бази даних. Для забезпечення відтворюваності, об’єктивності отриманих результатів та їх подальшого аналізу було сформовано чітке експериментальне середовище, визначено контрольну (базову) систему, набір порівнюваних характеристик і методику проведення вимірювань. Експерименти проводилися на одному екземплярі сервісу, розгорнутому на локальному комп’ютері, за однакових умов для обох оцінюваних підходів, що дозволяє виключити вплив сторонніх факторів і забезпечує чистоту дослідження.

4.1.1 Визначення базової системи порівняння

Базова система порівняння представляє собою типовий сервіс валідації API-ключів, реалізований із використанням реляційної бази даних як централізованого сховища. Під час кожного запиту сервіс здійснює звернення до БД для визначення фактичного статусу API-ключа, включно з перевіркою того, чи не був він відкликаний. Такий підхід є широко застосовуваним у сучасних корпоративних системах, однак характеризується залежністю від затримок у запитах до БД, наявністю потенційної точки відмови та обмеженою масштабованістю при збільшенні навантаження.

Вибір саме цієї системи як базової є обґрунтованим, оскільки вона репрезентує традиційний механізм валідації API-ключів, на тлі якого можна кількісно продемонструвати переваги запропонованої бібліотеки. В обох випадках було забезпечено однакову функціональність, що унеможливлює вплив логічних відмінностей на результати. Для коректності порівняння конфігурація середовища

виконання, обчислювальні ресурси, модель даних і навантажувальні патерни залишалися незмінними впродовж усього циклу експериментів.

4.1.2 Визначення метрик порівняння

Для оцінювання ефективності роботи системи були обрані дві ключові метрики, що дають змогу найбільш повно охарактеризувати продуктивність процесу обробки API-ключів і відобразити можливі переваги запропонованого рішення.

Перша метрика — це пропускна здатність сервісу, що визначається як кількість успішно опрацьованих запитів за одну секунду на одному екземплярі сервісу. Ця характеристика є критично важливою в умовах високонавантажених систем, де швидкість валідації ключів безпосередньо впливає на загальну результативність і відгук мікросервісної інфраструктури.

Друга метрика — це середня затримка обробки запиту, що відображає час, необхідний сервісу для виконання операції валідації API-ключа. Затримка включає процеси внутрішньої обробки, можливі виклики до БД та інші допоміжні операції. Вимірювання цієї метрики дозволяє оцінити не лише пікові значення, але й загальну швидкодію системи під різними рівнями навантаження.

Використання цих двох показників забезпечує комплексне розуміння того, як поведеться система за умови зростання числа клієнтів, зміни конфігурацій API-ключів та наявності відкликаних ключів. Крім того, поєднання пропускної здатності із середньою затримкою дозволяє виявити як можливі “вузькі місця”, так і місця оптимізації.

4.1.3 Опис проведення експериментів

Експериментальна частина дослідження передбачає послідовне вимірювання продуктивності двох варіантів системи — базової та системи, реалізованої з використанням розробленої бібліотеки. Для кожної системи було проведено три основні експериментальні серії, що відрізнялися відсотком відкликаних API-ключів у

вибірці. Це дозволило оцінити не лише загальну швидкодію, але й чутливість обох підходів до різних часток відкликаних ключів, що є важливим у деяких реальних сценаріях, де відкликання може відбуватися досить часто.

У ході першого експерименту досліджувалася продуктивність системи за умови відсутності відкликаних API-ключів. Такий сценарій характеризує ідеальні умови, коли всі ключі вважаються дійсними і не потребують додаткових перевірок. Він дозволяє оцінити базову пропускну здатність сервісу та характер зміни затримок із ростом числа паралельних клієнтів.

У межах другого експерименту частка відкликаних API-ключів становила п’ять відсотків від загального обсягу запитів. Такий рівень є типовим для систем, у яких відкликання здійснюється нерегулярно або відбувається на рівні окремих мікросервісів. Дослідження дає змогу визначити вплив перевірки відкликання на швидкодію обох систем та оцінити різницю між централізованим та оптимізованим механізмами обробки.

Третій експеримент проводився за умови десятивідсоткової частки відкликаних ключів. Збільшення цього показника досягає рівня сценаріїв підвищеної інтенсивності відкликань, що є характерними, наприклад, для інцидентів безпеки, ротації ключів або систем зі значною кількістю короткотривалих API-ключів. У такій конфігурації можна виявити вплив зростаючого навантаження на механізм відкликання та визначити стійкість запропонованого підходу.

Для кожного експерименту застосовувалося зростаюче навантаження, що моделювалося зміною числа паралельних клієнтських потоків. Для кожного рівня навантаження вимірювалися пропускна здатність та середня затримка обробки запиту. Дані фіксувалися незалежно для обох реалізацій сервісу, що дозволило надалі провести їх порівняльний аналіз.

4.2 Проведення експериментального дослідження

Експериментальне дослідження полягало у проведенні серії вимірювань пропускної здатності та середньої затримки обробки запитів при змінних умовах навантаження та різних частках відкликаних API-ключів. Було виконано три експериментальні сценарії як для базового сервісу, що здійснює валідацію ключів за допомогою запитів до реляційної бази даних, так і для сервісу, побудованого із використанням розробленої оптимізованої бібліотеки.

В обох випадках один екземпляр сервісу обробляв потік запитів від 1 до 100 потоків, при цьому кожен потік надсилав однакову кількість запитів — 1000. Такий підхід забезпечує контрольоване масштабування навантаження та дозволяє об'єктивно оцінити зміну продуктивності залежно від кількості паралельних клієнтів.

4.2.1 Дослідження пропускної здатності та затримки за відсутності відкликань API-ключів

Перший етап експериментів направлений на оцінювання максимальної продуктивності обох систем за ідеальних умов, коли в наборі API-ключів відсутні відкликані значення. Цей сценарій дозволяє визначити верхню межу ефективності кожної реалізації та порівняти їхню базову швидкодію без додаткових операцій відкликання.

Результати вимірювань для сервісу, побудованого на традиційній базі даних, наведено в таблиці 4.1. Як показано у таблиці, пропускна здатність системи зростає разом із збільшенням кількості потоків, але після досягнення приблизно 20 потоків приріст продуктивності суттєво сповільнюється. Середня затримка збільшується майже лінійно та сягає понад 60 мс при використанні 100 потоків. Такі результати вказують на суттєву залежність системи від затримок під час звернення до бази даних.

Результати для сервісу, що працює на основі розробленої бібліотеки, подано в таблиці 4.2. У порівнянні з базовою системою цей сервіс демонструє значно вищу пропускну здатність на усіх рівнях навантаження. При великій кількості потоків (50–

100) пропускна здатність перевищує показники базового сервісу більш ніж у 4 рази, а середня затримка залишається нижчою у кілька разів.

Таблиця 4.1 – Продуктивність базового сервісу за відсутності відкликаних ключів

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
1	1000	214.55	4.12
5	5000	1009.69	4.43
10	10000	1533.74	5.94
20	20000	1614.07	11.80
50	50000	1565.04	31.18
100	100000	1569.76	62.57

Таблиця 4.2 – Продуктивність сервісу на основі розробленої бібліотеки за відсутності відкликаних ключів

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
1	1000	228.62	3.85
5	5000	1381.60	3.10
10	10000	2699.06	3.21
20	20000	4554.77	3.90
50	50000	6201.94	7.42
100	100000	6087.91	15.63

Аналіз наведених результатів дозволяє зробити висновок, що використання розробленого підходу до валідації ключів забезпечує значно вищу ефективність обробки запитів у сценаріях без відкликань. Поліпшення проявляється як у збільшенні пропускної здатності, так і у помітному зниженні середньої затримки.

4.2.2 Дослідження пропускної здатності та затримки за різних відсотків частки відкликаних API-ключів

На другому етапі експериментів оцінювалася чутливість обох систем до наявності відкликаних API-ключів. Для цього було виконано два окремі сценарії: при частці відкликаних ключів 5 % та 10 %. У цих умовах система на базі даних виконує додаткові операції пошуку та порівняння, тоді як оптимізована бібліотека застосовує власні внутрішні структури даних для швидкого визначення стану ключа.

Результати для базового сервісу при 5% відкликаних ключів наведено в таблиці 4.3, а при 10% у таблиці 4.4. Характерною рисою є незначне погіршення пропускної здатності та зростання затримки порівняно з випадком без відкликань. Це відповідає очікуваній поведінці, оскільки збільшення частки відкликаних ключів збільшує кількість додаткових звернень до бази даних.

Результати для сервісу на основі розробленої бібліотеки при 5% відкликань представлено в таблиці 4.5, а при 10% — у таблиці 4.6. Попри збільшення частки відкликаних ключів, сервіс продовжує демонструвати значно вищу пропускну здатність, ніж базовий. Лише при одиничному потоці та 10% відкликань спостерігається помітне зниження продуктивності; однак при збільшенні числа потоків пропускна здатність залишається стабільно високою.

Таблиця 4.3 – Продуктивність базового сервісу при 5% відкликаних ключів

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
1	1000	195.75	4.38
5	5000	972.63	4.37
10	10000	1526.10	5.66
20	20000	1553.86	11.63
50	50000	1457.74	31.76
100	100000	1564.93	59.74

Таблиця 4.4 – Продуктивність базового сервісу при 10% відкликаних ключів

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
1	1000	187.89	4.35

Продовження таблиці 4.4

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
5	5000	1013.27	3.92
10	10000	1540.59	5.33
20	20000	1455.36	11.86
50	50000	1478.90	29.56
100	100000	1483.54	59.57

Таблиця 4.5 – Продуктивність сервісу на основі розробленої бібліотеки при 5% відкликань

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
1	1000	210.41	4.16
5	5000	1124.86	3.84
10	10000	1873.44	4.63
20	20000	2941.41	6.05
50	50000	4049.88	11.32
100	100000	4007.65	23.18

Таблиця 4.6 – Продуктивність сервісу на основі розробленої бібліотеки при 10% відкликань

Threads	Total Requests	Throughput (req/s)	Avg Latency (ms)
1	1000	139.67	6.36
5	5000	1114.35	3.77
10	10000	1905.11	4.43
20	20000	2732.00	6.36
50	50000	3948.77	11.28
100	100000	3990.60	22.75

Аналіз показує, що збільшення частки відкликаних ключів має різний вплив на дві системи. У базовій системі спостерігається стабільне зниження пропускної здатності, тоді як оптимізований сервіс демонструє значно слабшу залежність від частки відкликань. Це свідчить про кращу масштабованість та ефективність запропонованого рішення на навантажених сервісах.

4.2.3 Порівняння результатів та узагальнення

Для комплексної візуалізації та порівняння отриманих результатів було побудовано чотири графіки, які відображають залежності пропускної здатності та середньої затримки від кількості потоків. На рисунку 4.1 наведено залежність середньої затримки від числа потоків для трьох сценаріїв (0 %, 5 %, 10 %) для базового сервісу. На графіку чітко видно зростання затримки зі збільшенням навантаження, а також вплив частки відкликань, який є незначним на середніх і великих значеннях потоків.

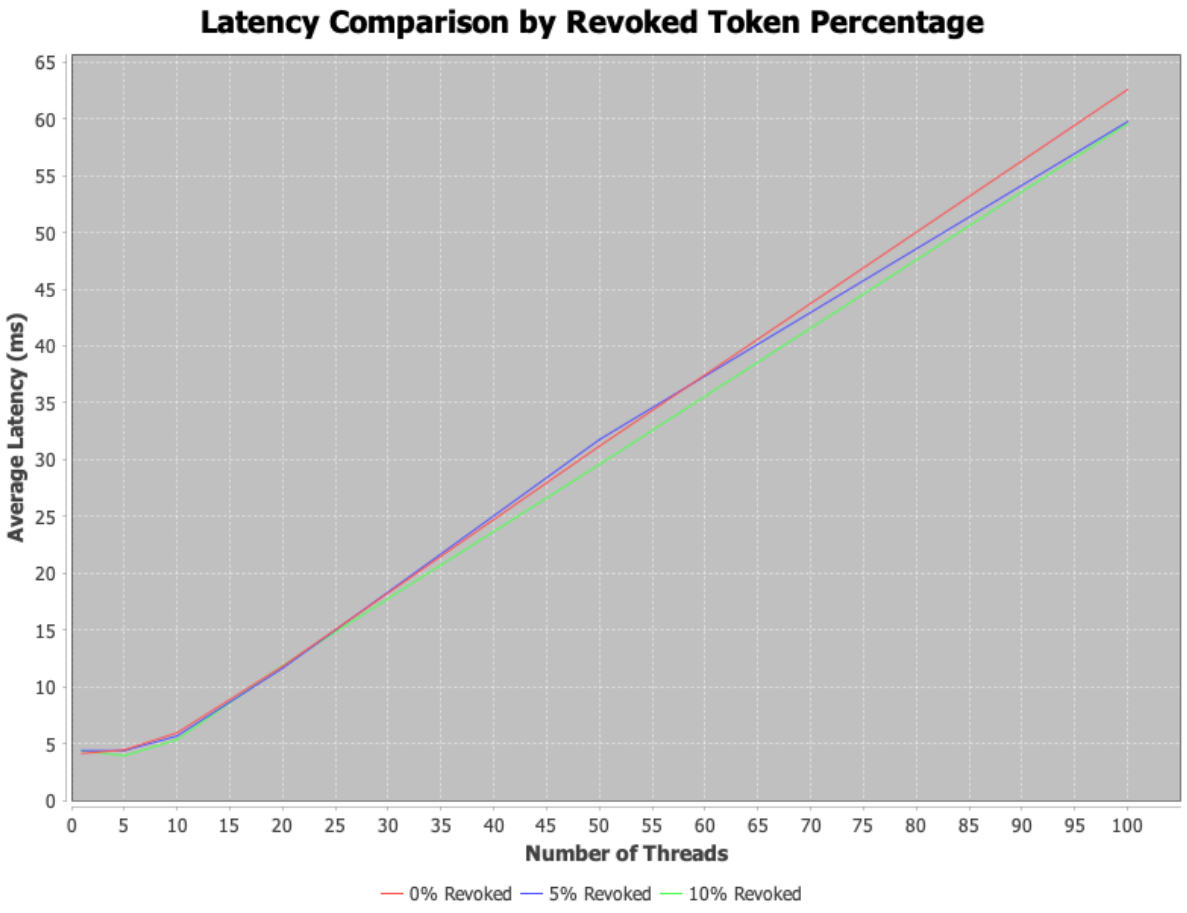


Рисунок 4.1 - залежність середньої затримки від числа потоків для базового сервісу

На рисунку 4.2 представлено аналогічну залежність для сервісу на основі розробленої бібліотеки. Цей графік демонструє значно повільніше зростання

затримки, особливо в діапазоні від 1 до 50 потоків, що свідчить про кращу оптимізацію внутрішньої обробки ключів.

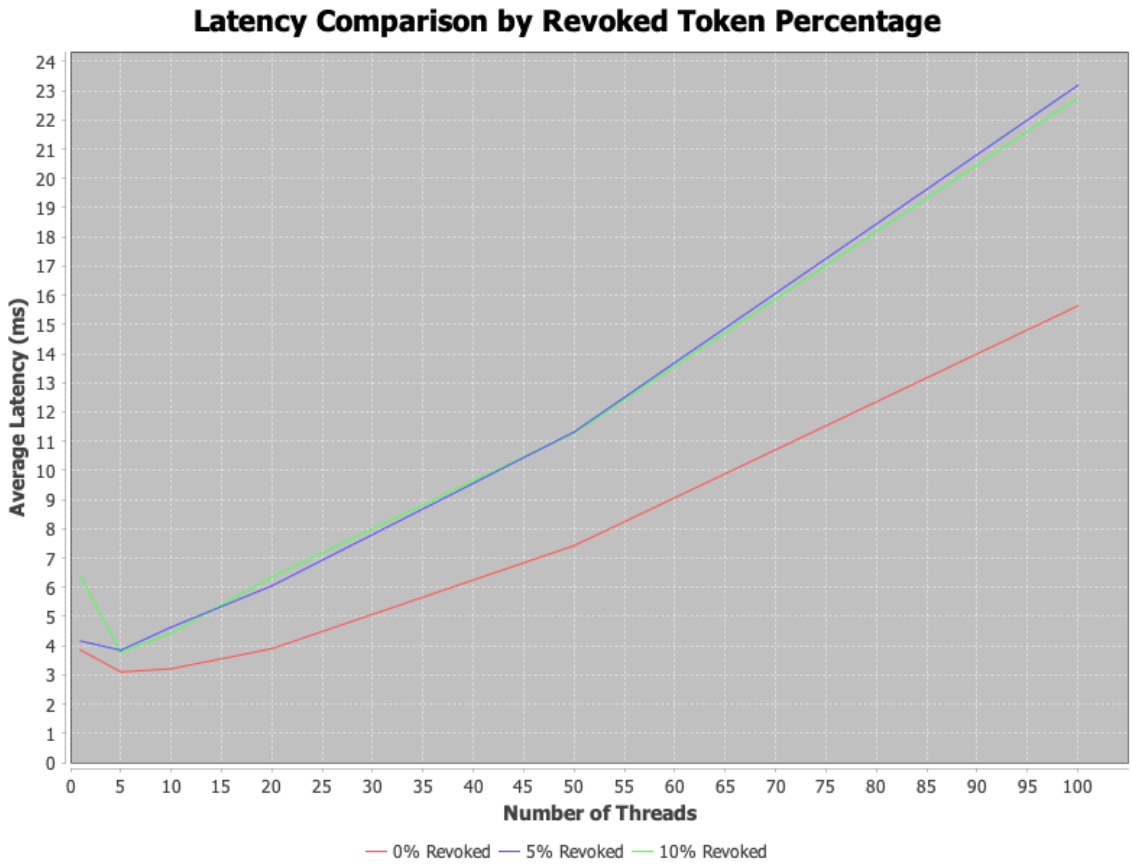


Рисунок 4.2 - залежність середньої затримки від числа потоків для сервісу заснованого на розробленій бібліотеці

На рисунку 4.3 показано залежність пропускної здатності від кількості потоків для трьох сценаріїв у базовій системі. Пропускна здатність зростає до певної межі, після чого майже стабілізується через обмеження, пов’язані з пропускною здатністю бази даних.

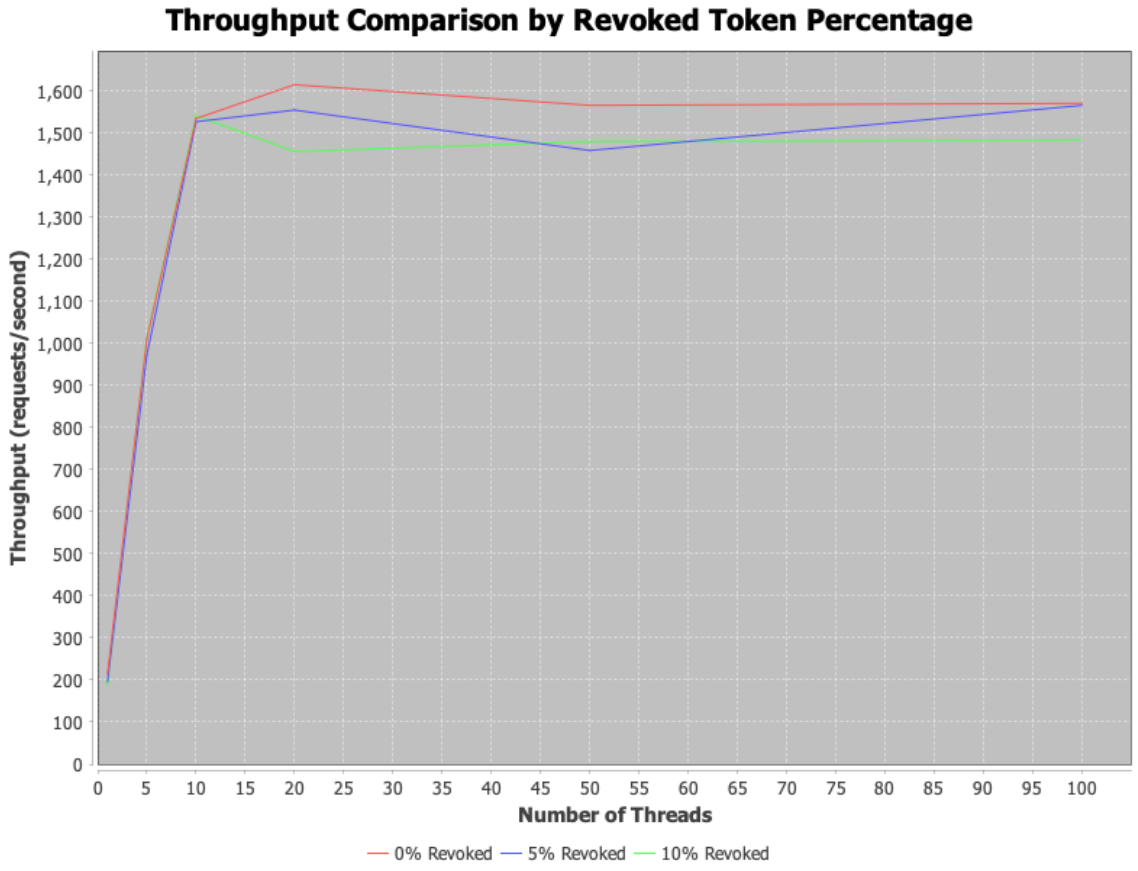


Рисунок 4.3 - залежність пропускної здатності від числа потоків для базового сервісу

На рисунку 4.4 подано результати для сервісу з використанням оптимізованої бібліотеки. Графік демонструє істотно вищу пропускну здатність, а також кращу рівномірність масштабування при збільшенні кількості потоків. Це дозволяє стверджувати, що розроблена бібліотека забезпечує суттєво вищу продуктивність і здатність до масштабування.

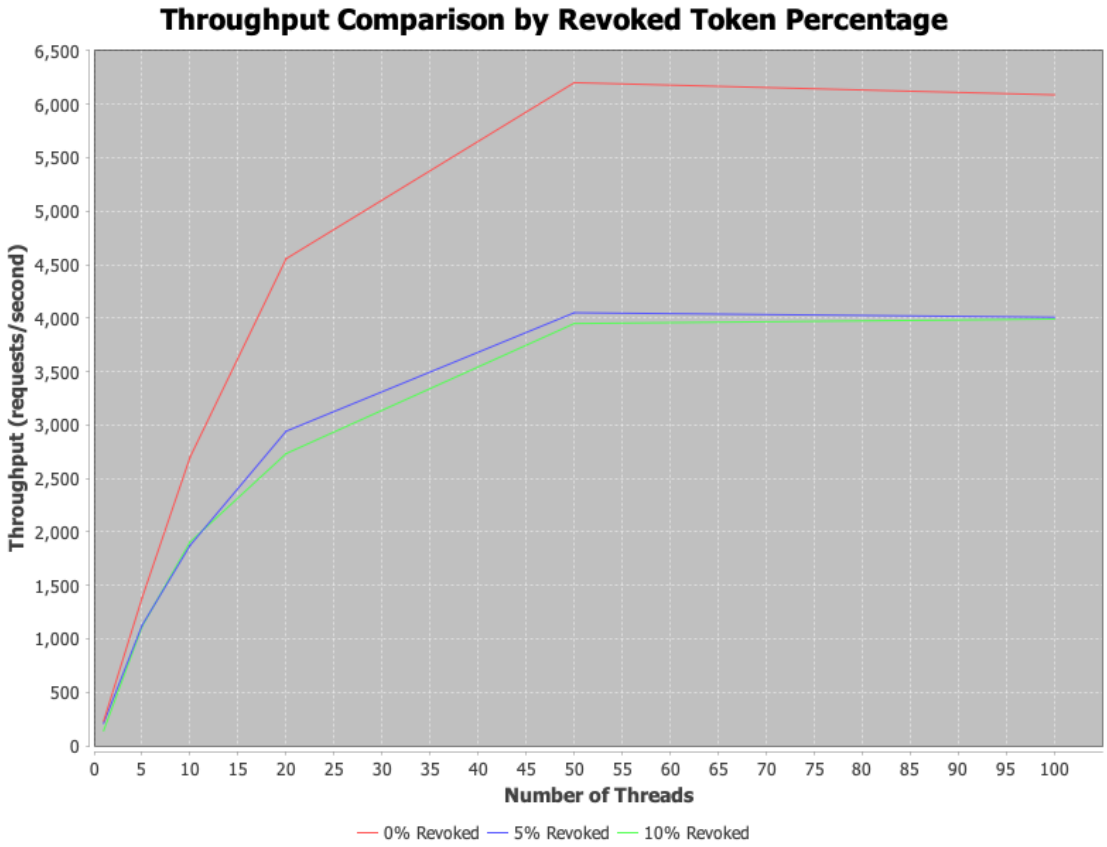


Рисунок 4.4 - залежність пропускної здатності від числа потоків для сервісу заснованого на розробленій бібліотеці

Порівняння даних, представлених у таблицях і графіках, дає змогу узагальнено стверджувати, що запропонована бібліотека забезпечує значний приріст продуктивності та стабільності при зростанні навантаження. Зниження середньої затримки у кілька разів та збільшення пропускної здатності в 2–3 рази є суттєвими аргументами на користь використання покращеного підходу до обробки API-ключів.

4.3 Аналіз достовірності результатів

Достовірність отриманих експериментальних даних ґрунтується на відтворюваності середовища для обох досліджуваних систем, а також на фундаментальних властивостях обчислювальних операцій, залучених у процесі валідації API-ключів. Порівняння продуктивності базового сервісу, що використовує реляційну базу даних, та сервісу, побудованого на основі розробленої бібліотеки,

спирається на відмінності у властивостях операцій, які ці системи виконують при обробці запитів.

Для базового сервісу головним елементом обробки запиту є звернення до бази даних для визначення стану API-ключа, включно з перевіркою факту його можливого відкликання. Як відомо, операції доступу до зовнішніх сховищ є суттєво дорожчими за операції, що виконуються в оперативній пам’яті, навіть у випадку використання оптимізованих SQL-запитів або індексів. Час виконання таких операцій залежить від швидкодії дискового підсистеми, мережевих затримок, блокувань та інших факторів. Це робить їх джерелом затримок у процесі обробки запитів, що підтверджено й у існуючих дослідженнях, у тому числі при розробленні самої бібліотеки.

У системі, реалізованій на основі розробленої бібліотеки, логіка визначення стану API-ключа не потребує звернення до зовнішнього сховища, окрім використання високошвидкісного розподіленого кешу. Основні операції виконуються локально в оперативній пам’яті сервісу — це перевірка цифрового підпису JWT-токена, валідація полів заголовка, а при наявності відкликань — перевірка приналежності ключа до відповідної епохи зі зверненням до структури швидкого доступу (Bloom-фільтра). Особливістю цих операцій є їх низька затримка. Криптографічні операції перевірки підпису виконуються у межах мікросекунд чи одиниць мілісекунд на сучасному CPU, а операції роботи з Bloom-фільтрами або локальними структурами даних мають практично константну складність. Це підтверджується як теоретично, так і практично, у технічному описі бібліотеки.

Саме ця різниця у характері операцій і забезпечує достовірність спостережуваних відмінностей у пропускній здатності та затримках, що робить експериментальні дані узгодженими з теоретичними положеннями.

4.4 Порівняння надійності систем

Надійність системи управління API-ключами визначається здатністю протистояти компрометації API-ключів та запобігати несанкціонованому доступу. У

базовому сервісі модель захисту базується виключно на секретності API-ключа, тобто у разі його втрати, забезпечити захист від несанкціонованих запитів стає надзвичайно складно. При втраті ключа користувач або система повинні вручну ініціювати його відкликання, після чого сервіс оновить статус API-ключа у базі даних.

Запропонована бібліотека вирішує цю проблему за рахунок впровадження механізму підтвердження володіння API-ключем. У поданому технічному описі розглянуто реалізацію DPoP-механізму, який прив’язує API-ключ до криптографічної пари ключів, що знаходиться виключно у власника. Це означає, що для виконання кожного запиту необхідно не лише знати API-ключ, але й мати доступ до приватного ключа, яким підписується DPoP-підтвердження. Навіть якщо зловмисник отримає API-ключ, він не зможе сформувати коректний підписаний доказ, що робить ключ практично непридатним для атак.

Таким чином, з точки зору надійності, бібліотека забезпечує принципово вищий рівень захисту порівняно з базовим сервісом, зводячи до мінімуму вплив компрометації ключа на безпеку системи.

4.5 Порівняння масштабованості систем

Масштабованість є критичною властивістю сервісів, що використовуються у високонавантажених розподілених системах. У базовому сервісі операції валідації API-ключів безпосередньо залежать від продуктивності реляційної бази даних, яка виступає єдиною точкою централізованого звернення. Навіть при наявності індексів та оптимізацій пропускна здатність бази даних залишається обмеженою характеристиками апаратної інфраструктури. Горизонтальне масштабування є обмеженим: збільшення кількості екземплярів сервісу не призводить до лінійного приросту продуктивності, оскільки всі вони продовжують звертатися до одного й того ж сховища. Це створює «вузьке місце», яке визначає верхню межу масштабованості системи.

На противагу цьому, сервіс, побудований із використанням розробленої бібліотеки, не потребує централізованої бази даних для валідації ключів. Усі основні операції виконуються локально або у взаємодії з розподіленим кешем, який може масштабуватися горизонтально. Такий кеш дозволяє збільшувати кількість вузлів у міру росту навантаження, що забезпечує розподіл даних між кількома вузлами й підвищення загальної пропускної здатності.

Таким чином, запропоноване рішення забезпечує значно вищу масштабованість порівняно з базовим сервісом. Відсутність централізованої БД як точки обмеження, використання розподіленого кешу і локальних обчислень дозволяє системі працювати ефективно навіть при різкому зростанні обсягу навантаження.

Висновки до розділу

У даному розділі було проведено комплексне експериментальне дослідження запропонованої бібліотеки для обробки API-ключів та здійснено її порівняння з традиційним підходом, що базується на централізованій перевірці ключів у реляційній базі даних. На основі виконаної серії експериментів із варіюванням навантаження та різних часток відкликаних ключів отримано кількісні результати щодо пропускної здатності та середньої затримки обробки запитів для обох систем.

Дослідження засвідчило, що сервіс, реалізований із використанням розробленої бібліотеки, демонструє вищу продуктивність: пропускна здатність збільшується у декілька разів, а середня затримка зменшується навіть при високих рівнях паралелізації. Аналіз достовірності підтвердив, що відмінності між системами зумовлені фундаментальними особливостями їх архітектур — зокрема, тим, що звернення до бази даних мають значно більшу затримку порівняно з локальними операціями та обчисленнями.

Крім того, здійснено якісне порівняння надійності та масштабованості систем. Встановлено, що запропонована бібліотека забезпечує вищий рівень захищеності за рахунок підтвердження володіння ключем, що усуває ризики, властиві традиційному

сервісу. Аналіз масштабованості показав, що базовий сервіс обмежується продуктивністю БД, тоді як оптимізований підхід може горизонтально масштабуватися завдяки відсутності необхідності централізованих запитів.

5 МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЄКТУ

5.1 Опис ідеї проєкту

Таблиця 5.1 — Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користучача
Створення програмної бібліотеки для безпечної, надійної та масштабованої обробки API-ключів. Бібліотека забезпечує генерацію, верифікацію, ротацію, відкликання та підтвердження володіння (DPoP) без необхідності централізованих звернень до бази даних. Основою є використання розширених JWT-токенів із вбудованими метаданими безпеки, механізмів епохального відкликання та Bloom-фільтрів.	Корпоративні та хмарні платформи, що працюють з API-сервісами. Мікросервісні архітектури.	Зменшення затримок за рахунок відсутності постійних запитів до БД. Підвищення безпеки завдяки механізмам DPoP та епохального відкликання. Нульовий час простою під час ротації ключів. Повна сумісність із середовищем Spring, спрощена інтеграція.
	API-gateway та системи доступу на основі токенів. Розподілені системи, де важлива низька затримка обробки запитів.	

Таблиця 5.2 — Опис ідеї стартап-проєкту

№	Техніко-економічні характеристики ідеї	Продукція конкурентів	W	N	S
1	Підтримка безпечної ротації ключів з нульовим часом простою	Більшість конкурентних рішень не підтримують ротацію або виконують її з необхідністю перезавантаження сервісів			S
2	Локальна верифікація API-ключів без звернення до центральної БД	Типові рішення виконують перевірку ключів у БД, що створює затримку та збільшує навантаження			S

Продовження таблиці 5.2

№	Техніко-економічні характеристики ідеї	Продукція конкурентів	W	N	S
3	Наявність Proof-of-Possession (DPoP) з механізмами захисту від перевикористання	Більшість реалізацій не підтримують DPoP або реалізують його частково			S
4	Механізм епохального відкликання токенів із Bloom-фільтрами	Рішення конкурентів зазвичай використовують статичні чорні списки або повну інвалідацію токенів			S
5	Підтримка індивідуальних ідентифікаторів API-ключів для гарантії коректного відкликання	Конкуренти не поєднують Bloom-фільтри з JTI-ключами, що знижує точність			S
6	Інтеграція зі Spring та можливість швидкого впровадження	Аналоги нерідко потребують значних змін конфігурації або ручної інтеграції			S

5.2 Технологічний аудит ідеї проєкту

Таблиця 5.3 — Технологічна здійсненність ідеї проєкту

№	Ідея проєкту	Технології реалізації	Наявність технологій	Доступність технологій
1	Безпечна генерація та ротація ключових пар	Java Cryptography Architecture (JCA), Spring Security, інтеграція з файловими чи KMS-сховищами	Так	Висока
2	Локальна верифікація JWT без БД	JWT-бібліотеки (JJWT, Nimbus JOSE), асиметричні алгоритми підпису RS256	Так	Висока
3	Механізм відкликання токенів на основі епох і Bloom-фільтрів	Redis, Bloom-фільтри, атомарні Redis-операції	Так	Висока

Продовження таблиці 5.3

№	Ідея проєкту	Технології реалізації	Наявність технологій	Доступність технологій
4	Підтримка DPoP та захист від атак перевикористання	Валідація цифрових підписів, Redis TTL-кешування, криптографічні алгоритми перевірки	Так	Висока

Обрана технологія реалізації ідеї проєкту:

1 — Реалізація на основі криптографічних механізмів Java та розподілених інструментів Redis.

Висновок: технологічна реалізація продукту є повністю можливою, оскільки всі необхідні інструменти та технологічні компоненти є доступними, зрілими та широко підтримуваними. Застосування Java-криптографії, JWT-бібліотек, Redis та Bloom-фільтрів забезпечує надійність, масштабованість і високу швидкість роботи, що відповідає вимогам до сучасних систем управління API-ключами.

5.3 Аналіз ринкових можливостей запуску стартап-проєкту

Таблиця 5.4 — Попередня характеристика потенційного ринку

№	Показники стану ринку	Характеристика
1	Кількість головних гравців	Помірна кількість: існують окремі компоненти безпеки (OAuth-сервери, API-gateway), проте спеціалізовані рішення для API-ключів із DPoP та епохальним відкликанням практично відсутні
2	Загальний обсяг продаж	Сегмент інструментів безпеки API та управління доступом демонструє стабільне зростання
3	Динаміка ринку	Висхідна: через збільшення кількості хмарних сервісів та мікросервісів попит на безпечні механізми управління API-ключами зростає

Продовження таблиці 5.4

№	Показники стану ринку	Характеристика
4	Наявність обмежень для входу	Низькі: відсутність жорсткої стандартизації дозволяє створювати нові архітектури управління API-ключами
5	Вимоги до стандартизації	Переважно рекомендаційні: базуються на JWT, DPoP, Redis — відкриті стандарти без високих бар'єрів
6	Середня норма рентабельності	Висока: продукти з безпеки API добре монетизуються завдяки SaaS-ліцензіям та корпоративним контрактам

Висновок: з огляду на збільшення попиту на інструменти API-безпеки, низьку конкуренцію у вузькому сегменті управління API-ключами та позитивну динаміку ринку, умови для запуску продукту є сприятливими та комерційно привабливими.

Таблиця 5.5 — Характеристика потенційних клієнтів стартап-проєкту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці груп	Вимоги споживачів
1	Захист API-ключів від витоків	IT-компанії, що працюють з розподіленими сервісами	Високі вимоги до безпеки	Низькі затримки, криптографічні гарантії, простота інтеграції
2	Масштабовані механізми управління ключами	Хмарні провайдери, SaaS-платформи	Велике навантаження, необхідність автоматизації	Висока продуктивність, безшовна ротація
3	Захист від перевикористання запитів і підтвердження володіння	API-gateway, фінтех та медичні системи	Висока чутливість до атак	Гарантований захист від підробок, точність відкликань

Таблиця 5.6 — Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	Конкуренти	Можлива поява аналогів з подібними механізмами	Удосконалення функціоналу, підтримка унікальних можливостей, оптимізація продуктивності

Продовження таблиці 5.6

№	Фактор	Зміст загрози	Можлива реакція компанії
2	Фінансування	Недостатність коштів на підтримку інфраструктури	Залучення інвестицій, поетапна розробка
3	Вихід аналогів	Швидке копіювання архітектури	Прискорений реліз MVP, активна маркетингова кампанія

Таблиця 5.7 — Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	Новий продукт	Нішевий ринок без прямих конкурентів	Активне бізнес-просування
2	Вихід аналогів	Формування високої потреби у покращених рішеннях	Розробка розширених фіч
3	Зворотний зв’язок	Поліпшення якості бібліотеки	Швидке реагування на потреби користувачів
4	Реклама та інтернет-платформи	Можливість масштабного охоплення	Використання статей та конференцій

Таблиця 5.8 — Ступеневий аналіз конкуренції на ринку

№	Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства та можливі дії
1	Тип конкуренції: монополістична	На ринку присутні продукти, що частково виконують окремі функції (OAuth-сервери, API-gateway), але не пропонують комплексного рішення з DPoP, епохальним відкликанням та ротацією без простою	Необхідність формувати унікальні переваги, впроваджувати відмінні функціональні можливості та підтримувати конкурентну цінову політику
2	Рівень конкурентної боротьби: глобальний	Виробники інструментів API-безпеки працюють міжнародно, пропонуючи різні підходи до керування доступом	Акцент на інноваційності, активне просування продукту на міжнародних професійних платформах та участь у технічних конференціях

Продовження таблиці 5.8

№	Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства та можливі дії
3	Галузева ознака: внутрішньогалузева	Продукт орієнтовано на ІТ-галузь, зокрема на розробників систем доступу, API-платформ та мікросервісних застосунків	Необхідність забезпечити просту інтеграцію, стандартизовані API та детальну документацію
4	Конкуренція за видами товарів: товарно-видова	Рішення на ринку є схожими за сферою застосування, але відрізняються набором функцій	Підвищення цінності продукту через додавання механізмів, що відсутні у конкурентів, зокрема DPoP, епохального відкликання та Bloom-фільтрів
5	Характер конкурентних переваг: цінові та нецінові	Конкуренти часто пропонують SaaS-моделі, але без спеціалізації під API-ключі	Формування конкурентної ціни, акцент на унікальних архітектурних рішеннях
6	Інтенсивність конкуренції: марочна	Потрібність власного бренду та репутації продукту для відокремлення від ринку	Розробка власної візуальної ідентичності та фірмового стилю, позиціонування продукту як інноваційного та технологічно лідируючого

Таблиця 5.9 — Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Характеристика впливу на стартап-проєкт
Прямі конкуренти в галузі	Прямих аналогів, що поєднують локальну верифікацію JWT, DPoP-перевірку та епохальне відкликання, практично немає; конкуренція носить непрямий характер
Потенційні конкуренти	Можлива поява нових бібліотек для керування API-ключами, особливо з боку великих cloud-провайдерів або постачальників API-gateway
Постачальники	Залежність від Redis, Bloom-фільтрів та криптографічних бібліотек; постачальники є стабільними, відкритими та не створюють загроз монополізації

Продовження таблиці 5.9

Складові аналізу	Характеристика впливу на стартап-проєкт
Клієнти	IT-компанії з високими вимогами до безпеки; мають середній рівень чутливості до ціни, але високі вимоги до надійності та масштабованості
Товари-замінники	OAuth2-сервери, API-gateway, кешуючі інструменти; вони не забезпечують DPoP або децентралізоване відкликання, тому заміна неповноцінна

Проаналізувавши можливості роботи на ринку з огляду на конкурентну ситуацію можна зробити висновок: оскільки кожний з існуючих продуктів не впливає у великій мірі на поточну ситуацію на ринку в цілому, кожний з існуючих продуктів має свою специфічну сферу використання та свої позитивні та негативні сторони щодо рішення певних типів задач, то робота та вихід на даний ринок є можливою і реалізованою задачею [23].

Для виходу на ринок продукт повинен мати функціонал що відсутній у продуктів-аналогів, повинен задовольняти потреби користувачів, мати необхідний та достатній функціонал з конфігурування, підтримку зі сторони розробників та можливість розробки спеціального функціоналу за відповідною ліцензією.

Таблиця 5.10 — Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування
1	Локальна верифікація API-ключів без БД	Забезпечує суттєве зниження затримки виконання запитів та високу масштабованість
2	Епохальне відкликання токенів	Дає змогу відкликати токени без підтримки глобальних чорних списків та без збільшення навантаження
3	Bloom-фільтри для швидких перевірок	Забезпечують майже миттєву перевірку відкликаних JTI при мінімальному обсязі пам’яті
4	Підтвердження володіння (DPoP)	Значно підвищує рівень безпеки, унеможливлюючи крадіжку токенів і атаки перевикористання

Продовження таблиці 5.10

№	Фактор конкурентоспроможності	Обґрунтування
5	Нульовий час простою під час ротації ключів	Надає перевагу при роботі у високонавантажених системах з безперервною доступністю
6	Простота інтеграції зі Spring	Знижує поріг входу для розробників та скорочує час впровадження
7	Відсутність необхідності у спеціалізованому обладнанні	Знижує витрати на інфраструктуру та підтримку
8	Висока продуктивність за рахунок Redis і атомарних інструкцій	Забезпечує стабільну роботу під великим навантаженням
9	Можливість гнучкої конфігурації	Дає змогу адаптувати бібліотеку під різні архітектури та формати розгортання
10	Відсутність прямих замінників	Створює унікальну конкурентну нішу на ринку систем управління API-ключами

Таблиця 5.11 — Порівняльний аналіз сильних та слабких сторін системи кешування мало змінних даних

№	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з запропонованим						
			-3	-2	-1	0	+1	+2	+3
1	Швидкість обробки запитів	18		+					
2	Затримка при валідації ключів	19			+				
3	Наявність механізмів підтверження володіння API-ключем	17		+					
4	Підтримка Bloom-фільтрів	19				+			
5	Простота інтеграції	17					+		
6	Можливість відкликання ключів	19				+			

Таблиця 5.12 — SWOT аналіз стартап-проєкту

Сильні сторони (S): – локальна верифікація токенів без звернення до БД; – підтримка підтвердження володіння (DPoP) для захисту від атак; – епохальне відкликання токенів з Bloom-фільтрами; – висока продуктивність і низька затримка; – проста інтеграція зі Spring; – низькі вимоги до інфраструктури.	Слабкі сторони (W): – залежність від Redis у багатовузловій конфігурації; – потреба у знаннях криптографії для розширених налаштувань; – відсутність широкої впізнаваності на ранніх етапах.
Можливості (O): – ріст ринку API-безпеки; – перехід компаній на мікросервіси та хмарні архітектури; – попит на інструменти проти витоку ключів; – можливість інтеграцій з API-gateway та cloud-платформами.	Загрози (T): – поява великих конкурентів (AWS, Google); – швидке копіювання архітектурних рішень; – нестабільність фінансування на ранній стадії; – можливі зміни стандартів безпеки.

Таблиця 5.13 — Альтернативи ринкового впровадження стартап-проєкту

№	Альтернатива (комплекс заходів)	Ймовірність отримання ресурсів	Строки реалізації
1	Публічний реліз основного функціоналу з безкоштовним доступом	Основний ресурс — інженерний час; ресурс наявний	2–3 місяці
2	Запуск рекламної кампанії на технічних платформах	Потребує власних коштів; можливе поступове фінансування	1–2 місяці
3	Публікація технічних статей і документації	Основний ресурс — час; ресурс наявний	2–4 тижні
4	Участь у конференціях, мітапах та хакатонах	Потребує часу та бюджету; ресурс обмежено, але доступний	1–3 місяці
5	Партнерство з Open-source спільнотами	Потребує часу на підтримку; ресурс наявний	1–2 місяці

5.4 Розроблення ринкової стратегії проєкту

Таблиця 5.14 — Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи	Готовність споживачів сприйняти продукт	Орієнтовний попит у межах сегменту	Інтенсивність конкуренції	Простота входу в сегмент
1	IT-компанії, що розробляють мікросервісні або високонавантажені системи	Висока - потребують швидкої та безпечної верифікації	Високий попит, особливо у fintech та e-commerce	Середня	Висока
2	Архітектори та backend-розробники, що працюють зі Spring	Висока - знайомі з JVM-технологіями	Високий, зростаючий попит	Низька	Висока
3	Платформи API-gateway та інтеграційні рішення	Середня - потребують сертифікації	Середній попит	Середня	Середня
4	Хмарні провайдери та SaaS-платформи	Середня - потребують масштабованих безпечних рішень	Високий у випадку інтеграції	Висока	Низька
5	Команди DevOps і безпеки	Середня - орієнтація на стабільність та контроль	Середній попит	Низька	Висока

Відповідно до проведеного аналізу, оптимальними цільовими групами для впровадження нового програмного продукту є:

– IT-компанії, що розробляють мікросервісні системи;

– Backend-розробники та архітектори, які працюють зі Spring та Java-екосистемою;

– Команди DevOps та безпеки, відповідальні за керування секретами та API-доступом.

Вибір саме цих сегментів обумовлений високою готовністю до сприйняття продукту, значним потенційним попитом, невисокою інтенсивністю конкуренції та відносно простою інтеграцією у їхні технологічні процеси. Таким чином, для стартап-проєкту обрано стратегію масового маркетингу з орієнтацією на широку спільноту Java/Spring-розробників та підприємств, що активно використовують API-інфраструктуру й прагнуть підвищити рівень безпеки без збільшення операційних витрат.

Таблиця 5.15 — Визначення базової стратегії розвитку

Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Надання функціональності, що відсутня у товарів-замінників; підтримка клієнтів; створення унікального інноваційного продукту	Проведення рекламної кампанії, акцент на унікальних механізмах (PoP, епохальне відкликання); інформування через технічні статті, соціальні мережі та професійні ресурси; прямий контакт зі споживачами (розробниками та IT-компаніями); формування довіри та лояльності	Зниження ступеня замінності продукту; високий рівень прихильності клієнтів; унікальні властивості продукту; відмітні технічні характеристики (локальна верифікація, Bloom-фільтри, DPoP)	Стратегія диференціації

Таблиця 5.16 — Визначення базової стратегії конкурентної поведінки

Чи є проєкт «першопрохідцем» на ринку	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, які?	Стратегія конкурентної поведінки
Ні, оскільки є товари-заміннники, але дані товари замінники не мають деякого необхідного функціоналу	Так, ціль компанії знайти нових споживачів та, частково, забрати існуючих у конкурентів задля задоволення потреб останніх	Компанія частково копіює характеристики товару конкурента, основна ціль компанії розробка нового унікального функціоналу, з підтримкою основного функціоналу конкурентів	Стратегія заняття конкурентної ніші

Таблиця 5.17 — Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проєкту	Вибір асоціацій, які мають сформувати комплексну позицію проєкту
1	Висока безпека API-ключів, захист від витоку та підміни	Стратегія диференціації	DPoP, епохальне відкликання, криптографічна аутентифікація	«Безпека корпоративного рівня», «Криптографічна надійність»
2	Низька затримка та висока швидкість обробки	Стратегія диференціації	Локальна верифікація JWT, Redis-кешування, Bloom-фільтри	«Миттєва перевірка», «Продуктивність для високих навантажень»

Продовження таблиці 5.17

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проєкту	Вибір асоціацій, які мають сформувати комплексну позицію проєкту
3	Простота інтеграції зі Spring та мінімальні вимоги до інфраструктури	Стратегія диференціації	Автоматична конфігурація, модульність, легкість налаштування	«Просте впровадження», «Готове рішення для Java/Spring»

Відповідно до проведеного аналізу встановлено, що:

Базовою стратегією розвитку є стратегія диференціації, оскільки продукт пропонує унікальний набір функцій, недоступних у конкурентів (DPoP, епохальне відкликання токенів, локальна верифікація без БД), та орієнтований на створення цінності через інноваційність.

Стратегією конкурентної поведінки визначено стратегію заняття конкурентної ніші, оскільки стартап орієнтується не на масове копіювання рішень, а на задоволення специфічних потреб розробників, архітекторів і компаній, що прагнуть підвищити безпеку API-доступу.

Обрані стратегії узгоджуються між собою та забезпечують ефективне позиціонування продукту на ринку технологій API-безпеки.

5.5 Розроблення маркетингової програми стартап-проєкту

Таблиця 5.18 — Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
1	Захист API-ключів від викрадення та перевикористання	DPoP, криптографічна перевірка запиту	Відсутність аналогічного DPoP-функціоналу у типових продуктів

Продовження таблиці 5.18

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
2	Висока продуктивність і мінімальна затримка верифікації	Локальна перевірка токенів без звернення до БД	У конкурентів — централізована валідація з більшими затримками
3	Масштабоване та точне відкликання токенів	Епохальна модель відкликання, Bloom-фільтри	Конкуренти використовують чорні списки з великим навантаженням
4	Простота впровадження в існуючі Java/Spring системи	Автоматична інтеграція, модульність, готові конфігурації	Більшість аналогів потребують складного налаштування або окремих серверів

Таблиця 5.19 — Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
1. Товар за задумом	Система безпечного управління API-ключами та відкликання токенів на базі Java, що забезпечує низьку затримку, масштабованість і криптографічний захист завдяки локальній перевірці та Bloom-фільтрам		
2. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	Підтримка DPoP	Нм	Тх
	Локальна верифікація JWT	Нм	Тх
	Епохальне відкликання токенів	Нм	Тх
	Інтеграція зі Spring Boot	Нм	Тх
	Висока продуктивність та низька затримка	Нм	Тх
3. Товар із підкріпленням	До продажу: наявна повна документація та приклади використання		
	Після продажу: технічна підтримка з боку розробника, регулярні оновлення безпеки		
За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності, патент			

Таблиця 5.20 — Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи	Верхня та нижня межі встановлення ціни
Середній рівень (API-gateway, Security-модулі)	Високий (корпоративні IAM-рішення)	Середній та високий рівень доходів IT-компаній та розробників	Нижня межа — середній ціновий рівень продуктів; верхня межа — нижче рівня корпоративних систем IAM

Таблиця 5.21 — Формування системи збуту

Специфіка закупівельної поведінки	Функції збуту постачальника	Глибина каналу збуту	Оптимальна система збуту
Клієнти самостійно шукають інструменти безпеки, порівнюючи функціонал	Надання інформації, швидке реагування на запити, продаж електронних ліцензій	Однорівневий канал	Прямий продаж через сайт/репозиторій + інтеграція у маркетплейси (Maven Central, GitHub Marketplace)

Таблиця 5.22 — Концепція маркетингових комунікацій

№	Специфіка поведінки цільових клієнтів	Канали комунікацій	Ключові позиції для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Активне споживання технічного контенту; пошук ефективних рішень	Статті, технічні блоги, конференції	Висока безпека та швидкість	Пояснити унікальні характеристики продукту	«Інновації, що підвищують безпеку»

Продовження таблиці 5.22

№	Специфіка поведінки цільових клієнтів	Канали комунікацій	Ключові позиції для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
2	Постійний пошук рішень для оптимізації	Технічні форуми, GitHub, StackOverflow	Низька затримка, простота інтеграції	Показати вигоду у продуктивності	«Миттєва верифікація — більше продуктивності»

Як результат було створено ринкову (маркетингову) програму, що включає в себе визначення ключових переваг концепції потенційного товару, опис моделі товару, визначення меж встановлення ціни, формування системи збуту та концепцію маркетингових комунікацій.

Висновки до розділу

В п’ятому розділі описано стратегії та підходи з розроблення стартап-проєкту, визначено наявність попиту, динаміку та рентабельність роботи ринку, як висновок було вказано що існує можливість ринкової комерціалізації проєкту. Розглянувши потенційні групи клієнтів, бар’єри входження, стан конкуренції та конкурентоспроможність проєкту було встановлено що проєкт є перспективним. Розглянуто та вибрано альтернативу впровадження стартап-проєкту та доведено доцільність подальшої імплементації проєкту.

ВИСНОВКИ

У ході виконання магістерської дисертації було розглянуто питання, пов’язані з підвищенням надійності та безпеки машинної авторизації шляхом удосконалення процесів генерації, верифікації, ротації та відкликання API-ключів у розподілених системах. На основі аналізу існуючих бібліотек і інженерних підходів сформульовано задачу розробки Java бібліотеки, що використовує розширені JWT із підтримкою підтвердження володіння (DPoP) та уникає залежності від централізованих баз даних. Для розробки програмного забезпечення використано відкриті (Java, Spring екосистема, Redis, Bloom-фільтри, стандартні криптографічні алгоритми), що дозволяють інтегрувати бібліотеку у широкий спектр сервісів.

Розроблений оптимізований підхід відкликання API-ключів базується на епохальному механізмі з використанням Bloom-фільтрів для скорочення часу перевірки. Створено програмне забезпечення (бібліотеку), яке забезпечує генерацію API-ключів на основі розширених JWT-токенів з вбудованими метаданими відкликання, їх криптографічну верифікацію, перевірку володіння API-ключем та безперервну ротацію ключів без простою системи. Забезпечено інтеграцію даної бібліотеки в середовище фреймворку Spring. Проведено маркетинговий аналіз потенційного стартап-проєкту: описано ідею, виконано технологічний аудит, оцінено ринкові можливості, сформовано ринкову стратегію та маркетингову програму.

Наукова новизна одержаних результатів:

– удосконалено інженерні рішення ефективного представлення API-ключів у вигляді розширених JWT-токенів, які підтримують підтверження володіння;

– удосконалено алгоритм відкликання API-ключів на основі епохального механізму та використання Bloom-фільтрів..

Практична значимість результатів полягає у можливості інтеграції розробленої Java бібліотеки у будь-яку систему машинної авторизації для безпечної обробки API-ключів в екосистемі фреймворку Spring. Вона забезпечує швидке відкликання,

безперервну ротацію та масштабовану взаємодію між сервісами, що підвищує рівень безпеки та надійності системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OWASP API Security Project | OWASP Foundation. *OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation*. URL: https://owasp.org/www-project-api-security (дата звернення: 10.11.2025).

2. *NIST Technical Series Publications*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf (дата звернення: 10.11.2025).

3. Manage API keys | Authentication | Google Cloud. *Google Cloud*. URL: https://cloud.google.com/docs/authentication/api-keys#best_practices (дата звернення: 10.11.2025).

4. *NIST Technical Series Publications*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf (дата звернення: 10.11.2025).

5. State of Secrets Sprawl Report 2024. *GitGuardian: Secrets Security and NHI Governance*. URL: https://www.gitguardian.com/state-of-secrets-sprawl-report-2024 (дата звернення: 10.11.2025).

6. Bayesian Differential Privacy for Linear Dynamical Systems. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/9447811/keywords#keywords (дата звернення: 10.11.2025).

7. RFC 9449: OAuth 2.0 Demonstrating Proof of Possession (DPoP). *» RFC Editor*. URL: https://www.rfc-editor.org/rfc/rfc9449 (дата звернення: 10.11.2025).

8. Recommendation for Key Management. *NIST Technical Series Publications*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf (дата звернення: 10.11.2025).

9. Understand static and dynamic secrets | Vault | HashiCorp Developer. *Understand static and dynamic secrets | Vault | HashiCorp Developer*. URL: https://developer.hashicorp.com/vault/tutorials/get-started/understand-static-dynamic-secrets (дата звернення: 10.11.2025).

10. Evaluating the Performance of Novel JWT Revocation Strategy | Acta Cybernetica. *Acta Cybernetica*. URL: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4195 (дата звернення: 10.11.2025).

11. Evaluating the Performance of a Novel JWT Revocation Strategy. *Acta Cybernetica*. URL: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4195/4027 (дата звернення: 10.11.2025).

12. Performance comparison of signed algorithms on JSON Web Token. *IOPscience | Scientific, Technical & Medical Journals*. URL: https://iopscience.iop.org/article/10.1088/1757-899X/550/1/012023 (дата звернення: 10.11.2025).

13. Schwake E. OWASP API Security Top 10 Explained - What is OWASP?. *API Security Solutions & Tools - API Protection Platform*. URL: https://salt.security/blog/owasp-api-security-top-10-explained (дата звернення: 10.11.2025).

14. Fast Revocation of Attribute-Based Credentials. URL: https://cs.ru.nl/~jhh/publications/epoch-revocation-journal.pdf (дата звернення: 10.11.2025).

15. CRSet: Private Non-Interactive Verifiable Credential Revocation. *arXiv.org*. URL: https://arxiv.org/abs/2501.17089 (дата звернення: 10.11.2025).

16. A blockchain-based certificate revocation management. URL: https://www.lrde.epita.fr/dload-new/papers/christian.21.cs.pdf (дата звернення: 10.11.2025).

17. Retiring and revoking grants - AWS Key Management Service. URL: https://docs.aws.amazon.com/kms/latest/developerguide/grant-delete.html (дата звернення: 10.11.2025).

18. HashiCorp Vault - Secrets Management, Policy-as-Code and Data Encryption - Digital Marketplace. *Digital Marketplace*.

URL: https://www.applytosupply.digitalmarketplace.service.gov.uk/g-cloud/services/516491187140869 (дата звернення: 10.11.2025).

19. API Security: Validating Auth and Access with Traffic Simulation Starts with Behavior | Speedscale. *Speedscale*. URL: https://speedscale.com/blog/api-security-validating-auth-and-access-with-traffic-simulation-starts-with-behavior/ (дата звернення: 10.11.2025).

20. Distributed Caching for High-Traffic APIs | Antler Digital. *Antler Digital*. URL: https://antler.digital/blog/distributed-caching-for-high-traffic-apis (дата звернення: 10.11.2025).

21. What Is Reactive Programming in Java: Examples, Frameworks | SaM Solutions. *SaM Solutions*. URL: https://sam-solutions.com/blog/java-reactive-programming/ (дата звернення: 10.11.2025).

22. What Is Reactive Programming in Java: Examples, Frameworks | SaM Solutions. *SaM Solutions*. URL: https://sam-solutions.com/blog/java-reactive-programming/ (дата звернення: 10.11.2025).

23. Robertson B. Efficient Cloud Key Management Strategies. *Thales Cloud Security Products*. URL: https://cpl.thalesgroup.com/blog/encryption/efficient-cloud-key-management-strategies (дата звернення: 10.11.2025).

24. Emergency Informatics: Using Computing to Improve Disaster Management. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/7469999 (дата звернення: 10.11.2025).

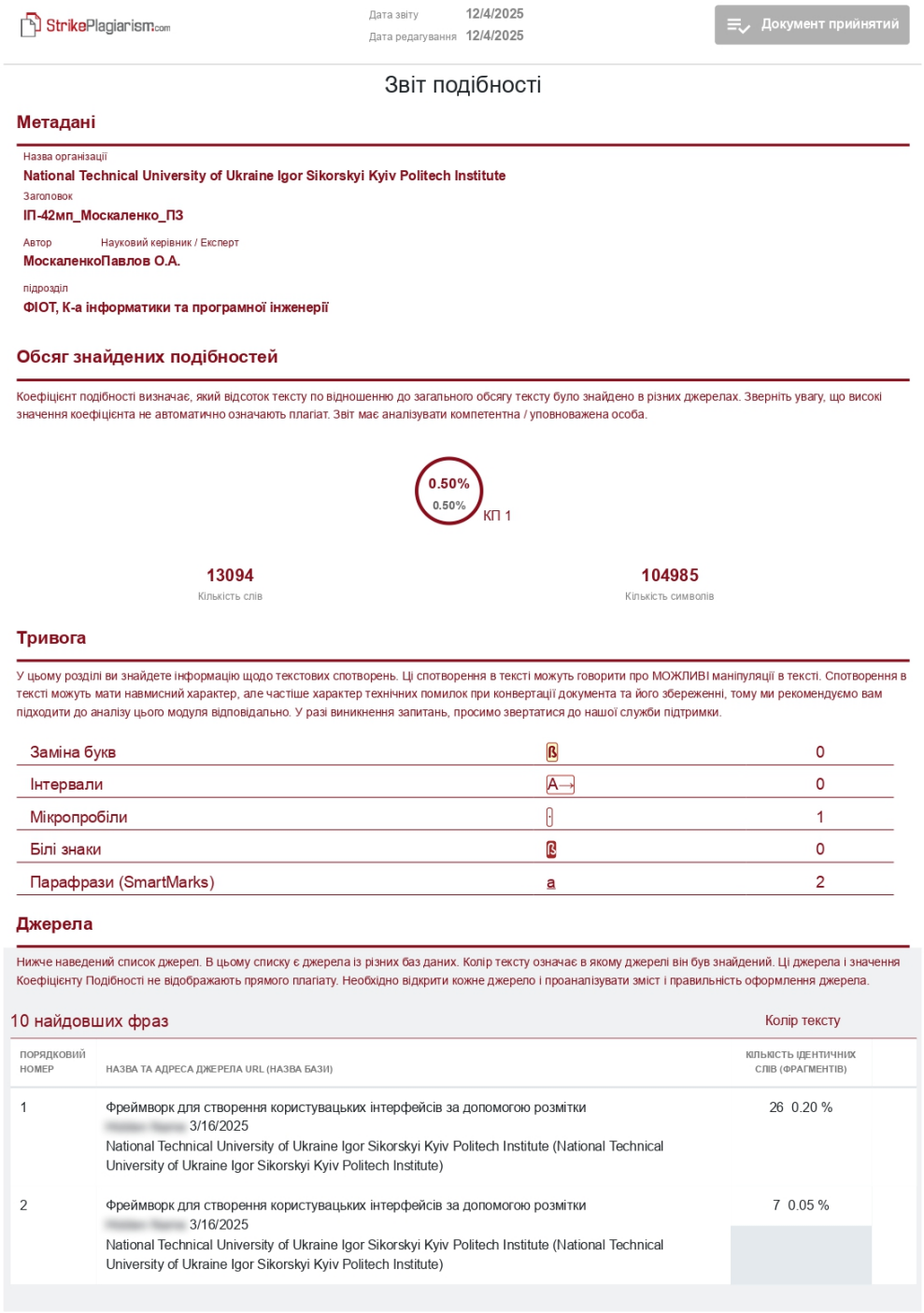
25. GitHub - skobow/apikey-authentication-spring-boot-starter: Spring boot starter to enable easy to use and configurable API key authentication for your Spring Boot project. *GitHub*. URL: https://github.com/skobow/apikey-authentication-spring-boot-starter (дата звернення: 10.11.2025).

26. GitHub - 42BV/api-key-authentication: Library to easily configure API Key authentication in (parts of) your Spring Boot Application. *GitHub*. URL: https://github.com/42BV/api-key-authentication (дата звернення: 10.11.2025).

27. GitHub - mihirdilip/aspnetcore-authentication-apikey. *GitHub*. URL: https://github.com/mihirdilip/aspnetcore-authentication-apikey (дата звернення: 10.11.2025).

ДОДАТОК А

Результати перевірки роботи на співпадіння



StrikePlagiarism.com

Дата звіту 12/4/2025
Дата редагування 12/4/2025

Документ прийнятий

Звіт подібності

Метадані

Назва організації
National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute
Заголовок
ІП-42мп_Москаленко_ПЗ
Автор Науковий керівник / Експерт
МоскаленкоПавлов О.А.
підрозділ
ФІОТ, К-а інформатики та програмної інженерії

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.50%
0.50%
КП 1

13094
Кількість слів

104985
Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	0
Інтервали	A⟶	0
Мікропробіли	·	1
Білі знаки	¶	0
Парафрази (SmartMarks)	a	2

Джерела

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Фреймворк для створення користувацьких інтерфейсів за допомогою розмітки [redacted] 3/16/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	26 0.20 %
2	Фреймворк для створення користувацьких інтерфейсів за допомогою розмітки [redacted] 3/16/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	7 0.05 %

Рисунок А.1 – Результат перевірки на співпадіння

3	https://dspace.znu.edu.ua/jspui/bitstream/12345/20020/1/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC_%D0%AE%D1%80%D1%8C%D1%94%D0%B2.pdf	6 0.05 %
4	https://ir.dpu.edu.ua/bitstreams/8a941db1-0476-45ce-8fe9-425287e80093/download	6 0.05 %
5	https://dspace.znu.edu.ua/jspui/bitstream/12345/20020/1/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC_%D0%AE%D1%80%D1%8C%D1%94%D0%B2.pdf	5 0.04 %
6	https://dspace.znu.edu.ua/jspui/bitstream/12345/20020/1/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC_%D0%AE%D1%80%D1%8C%D1%94%D0%B2.pdf	5 0.04 %
7	https://dspace.znu.edu.ua/jspui/bitstream/12345/20020/1/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC_%D0%AE%D1%80%D1%8C%D1%94%D0%B2.pdf	5 0.04 %
8	https://dspace.znu.edu.ua/jspui/bitstream/12345/20020/1/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC_%D0%AE%D1%80%D1%8C%D1%94%D0%B2.pdf	5 0.04 %

з домашньої бази даних (0.25 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Фреймворк для створення користувацьких інтерфейсів за допомогою розмітки [redacted] 3/16/2025 **National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)**	**33 (2) 0.25 %**

з програми обміну базами даних (0.00 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)

з Інтернету (0.24 %)

ПОРЯДКОВИЙ НОМЕР	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
2	https://dspace.znu.edu.ua/jspui/bitstream/12345/20020/1/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC_%D0%AE%D1%80%D1%8C%D1%94%D0%B2.pdf	26 (5) 0.20 %
3	https://ir.dpu.edu.ua/bitstreams/8a941db1-0476-45ce-8fe9-425287e80093/download	6 (1) 0.05 %

Список прийнятих фрагментів (немає прийнятих фрагментів)

ПОРЯДКОВИЙ НОМЕР	ЗМІСТ	КІЛЬКІСТЬ ОДНАКОВИХ СЛІВ (ФРАГМЕНТІВ)

Рисунок А.2 – Результат перевірки на співпадіння

ДОДАТОК Б

Лістинг коду

```
package org.kpi.api.key.service;

import io.jsonwebtoken.Claims;

import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class ApiKeyService {
    private final TokenVerificationService tokenVerificationService;

    public ApiKeyService(TokenVerificationService tokenVerificationService) {
        this.tokenVerificationService = tokenVerificationService;
    }

    public Set<String> extractScopes(String apiKey) {
        TokenVerificationService.TokenValidationResult result =
                tokenVerificationService.validateToken(apiKey);

        if (!result.isValid()) {
            return Set.of();
        }

        return extractScopesFromClaims(result.getClaims());
    }

    private Set<String> extractScopesFromClaims(Claims claims) {
        Object scopesObj = claims.get("scopes");
        if (scopesObj instanceof List) {
            return ((List<String>) scopesObj).stream().collect(Collectors.toSet());
        }
        return Set.of();
    }
}

package org.kpi.api.key.service;

import io.jsonwebtoken.Claims;
import org.kpi.api.key.config.DpopConfig;
import org.kpi.api.key.storage.RedisStore;

import java.math.BigInteger;
import java.net.URI;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.security.spec.RSAPublicKeySpec;
import java.time.Instant;
import java.util.Base64;
import java.util.Map;
import java.util.concurrent.TimeUnit;

public class DpopValidationService {
    private final TokenVerificationService tokenVerificationService;
    private final RedisStore redisStore;
```

```
    private final DpopConfig config;

    public DpopValidationService(TokenVerificationService tokenVerificationService,
                                 RedisStore redisStore,
                                 DpopConfig config) {
        this.tokenVerificationService = tokenVerificationService;
        this.redisStore = redisStore;
        this.config = config;
    }

    public DpopValidationResult validateDpop(String dpopProof, String accessToken,
String httpMethod, URI uri) {
        try {
            TokenVerificationService.TokenValidationResult tokenResult =
                    tokenVerificationService.validateToken(accessToken);
            if (!tokenResult.isValid()) {
                return DpopValidationResult.invalid("Invalid access token");
            }

            Claims tokenClaims = tokenResult.getClaims();
            PublicKey publicKey = extractPublicKeyFromToken(tokenClaims);
            if (publicKey == null) {
                return DpopValidationResult.invalid("No public key found in token");
            }

            String[] proofParts = dpopProof.split("\\.");
            if (proofParts.length != 2) {
                return DpopValidationResult.invalid("Invalid DPoP proof format");
            }

            String payloadBase64 = proofParts[0];
            String signatureBase64 = proofParts[1];

            byte[] payloadBytes = Base64.getUrlDecoder().decode(payloadBase64);
            String payload = new String(payloadBytes, StandardCharsets.UTF_8);

            String[] payloadParts = payload.split("\\|");
            if (payloadParts.length != 4) {
                return DpopValidationResult.invalid("Invalid payload format");
            }

            String proofMethod = payloadParts[0];
            String proofPath = payloadParts[1];
            String proofQuery = payloadParts[2];
            String timestampStr = payloadParts[3];

            if (!validateRequestDetails(proofMethod, proofPath, proofQuery,
httpMethod, uri)) {
                return DpopValidationResult.invalid("Request details mismatch");
            }

            long timestamp = Long.parseLong(timestampStr);
            String proofHash = calculateProofHash(payload);

            if (!validateTimestampAndReplay(timestamp, proofHash)) {
                return DpopValidationResult.invalid("Invalid timestamp or replay
detected");
            }
```

```
            byte[] signatureBytes = Base64.getUrlDecoder().decode(signatureBase64);
            if (!verifySignature(payloadBytes, signatureBytes, publicKey)) {
                return DpopValidationResult.invalid("Invalid signature");
            }

            recordProofUsage(proofHash);

            return DpopValidationResult.valid("DPoP validation successful");

        } catch (Exception e) {
            return DpopValidationResult.invalid("Error validating DPoP: " +
e.getMessage());
        }
    }

    private PublicKey extractPublicKeyFromToken(Claims claims) {
        try {
            Map<String, Object> popClaim = (Map<String, Object>) claims.get("pop");
            if (popClaim == null || !popClaim.containsKey("jwk")) {
                return null;
            }

            Map<String, Object> jwk = (Map<String, Object>) popClaim.get("jwk");
            return extractPublicKeyFromJwkMap(jwk);

        } catch (Exception e) {
            return null;
        }
    }

    private PublicKey extractPublicKeyFromJwkMap(Map<String, Object> jwk) {
        try {
            if (!"RSA".equals(jwk.get("kty"))) {
                return null;
            }

            String nBase64Url = (String) jwk.get("n");
            String eBase64Url = (String) jwk.get("e");

            if (nBase64Url == null || eBase64Url == null) {
                return null;
            }

            Base64.Decoder urlDecoder = Base64.getUrlDecoder();
            BigInteger modulus = new BigInteger(1, urlDecoder.decode(nBase64Url));
            BigInteger exponent = new BigInteger(1, urlDecoder.decode(eBase64Url));

            RSAPublicKeySpec keySpec = new RSAPublicKeySpec(modulus, exponent);
            KeyFactory keyFactory = KeyFactory.getInstance("RSA");
            return keyFactory.generatePublic(keySpec);

        } catch (Exception e) {
            return null;
        }
    }

    private boolean validateRequestDetails(String proofMethod, String proofPath,
String proofQuery,
                                           String httpMethod, URI uri) {
```

```
        if (!httpMethod.equalsIgnoreCase(proofMethod)) {
            return false;
        }

        if (!uri.getPath().equals(proofPath)) {
            return false;
        }

        String actualQuery = uri.getQuery();
        String expectedQuery = proofQuery.isEmpty() ? null : proofQuery;

        return !((actualQuery == null && expectedQuery != null) ||
                (actualQuery != null && !actualQuery.equals(expectedQuery)));
    }

    private boolean validateTimestampAndReplay(long timestamp, String proofHash) {
        long currentTime = Instant.now().getEpochSecond();

        if (timestamp < currentTime - config.getProofMaxAgeSeconds()) {
            return false;
        }

        String replayKey = config.getReplayPrefix() + "proof:" + proofHash;
        return !Boolean.TRUE.equals(redisStore.hasKey(replayKey));
    }

    private String calculateProofHash(String payload) throws Exception {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(payload.getBytes(StandardCharsets.UTF_8));
        return Base64.getUrlEncoder().withoutPadding().encodeToString(hash);
    }

    private boolean verifySignature(byte[] payload, byte[] signature, PublicKey
publicKey) {
        try {
            Signature sig = Signature.getInstance("SHA256withRSA");
            sig.initVerify(publicKey);
            sig.update(payload);
            return sig.verify(signature);
        } catch (Exception e) {
            return false;
        }
    }

    private void recordProofUsage(String proofHash) {
        String replayKey = config.getReplayPrefix() + "proof:" + proofHash;
        redisStore.set(replayKey, "1", config.getProofMaxAgeSeconds() * 2,
TimeUnit.SECONDS);
    }
}


package org.kpi.api.key.service;

import org.kpi.api.key.config.RevocationConfig;
import org.kpi.api.key.storage.BloomFilterOperations;
import org.kpi.api.key.storage.RedisStore;

import java.util.concurrent.TimeUnit;
```

```
public class EpochBloomFilterService {
    private final BloomFilterOperations bloomFilterOps;
    private final RedisStore redisStore;
    private final RevocationConfig config;

    public EpochBloomFilterService(BloomFilterOperations bloomFilterOps,
                                   RedisStore redisStore,
                                   RevocationConfig config) {
        this.bloomFilterOps = bloomFilterOps;
        this.redisStore = redisStore;
        this.config = config;
    }

    public boolean addToEpochFilter(long epoch, String jti) {
        String bloomFilterKey = config.getBloomFilterPrefix() + epoch;
        String jtiSetKey = config.getRevokedJtiPrefix() + epoch;

        ensureBloomFilterExists(bloomFilterKey);

        boolean bloomResult = bloomFilterOps.addToFilter(bloomFilterKey, jti);

        String jtiKey = jtiSetKey + ":" + jti;
        redisStore.set(jtiKey, "1",
                (config.getEpochDurationSeconds() * 2) + 300,
                TimeUnit.SECONDS);

        return bloomResult;
    }

    public boolean existsInEpochFilter(long epoch, String jti) {
        String bloomFilterKey = config.getBloomFilterPrefix() + epoch;

        if (!bloomFilterOps.filterExists(bloomFilterKey)) {
            return false;
        }

        if (!bloomFilterOps.existsInFilter(bloomFilterKey, jti)) {
            return false;
        }

        String jtiSetKey = config.getRevokedJtiPrefix() + epoch;
        String jtiKey = jtiSetKey + ":" + jti;
        return redisStore.get(jtiKey) != null;
    }

    public void cleanupEpochData(long epoch) {
        String bloomFilterKey = config.getBloomFilterPrefix() + epoch;
        bloomFilterOps.deleteFilter(bloomFilterKey);

        String jtiSetKey = config.getRevokedJtiPrefix() + epoch;
    }

    private void ensureBloomFilterExists(String bloomFilterKey) {
        if (!bloomFilterOps.filterExists(bloomFilterKey)) {
            bloomFilterOps.createFilter(bloomFilterKey,
                    config.getBloomFilterErrorRate(),
                    config.getBloomFilterCapacity());
        }
```

```
    }
}
```

```
package org.kpi.api.key.service;

import org.kpi.api.key.storage.KeyStorageException;
import org.kpi.api.key.storage.KeyStorageProvider;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;

public class KeyGenerationService {
    private final KeyStorageProvider keyStorageProvider;
    private final KeyIdStoreService keyIdStoreService;

    public KeyGenerationService(KeyStorageProvider keyStorageProvider,
KeyIdStoreService keyIdStoreService) {
        this.keyStorageProvider = keyStorageProvider;
        this.keyIdStoreService = keyIdStoreService;
    }

    public String generateAndStoreNewKey() throws Exception {
        try {
            String kid = keyIdStoreService.generateNewKeyId();

            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            keyGen.initialize(2048);
            KeyPair pair = keyGen.generateKeyPair();

            keyStorageProvider.storeKeyPair(kid, pair.getPrivate(),
pair.getPublic());
            keyIdStoreService.registerNewKey(kid);

            return kid;
        } catch (NoSuchAlgorithmException | KeyStorageException e) {
            throw new Exception("Key generation failed", e);
        }
    }

    public String rotateKeys() throws Exception {
        String newKid = generateAndStoreNewKey();
        return newKid;
    }

    public boolean testStorage() {
        try {
            return keyStorageProvider.testConnection();
        } catch (KeyStorageException e) {
            return false;
        }
    }
}
```

```
package org.kpi.api.key.service;
```

```
import org.kpi.api.key.config.KeyGenerationConfig;
import org.kpi.api.key.storage.RedisStore;

import java.util.UUID;

public class KeyIdStoreService {
    private final RedisStore redisStore;
    private final KeyGenerationConfig config;

    public KeyIdStoreService(RedisStore redisStore, KeyGenerationConfig config) {
        this.redisStore = redisStore;
        this.config = config;
    }

    public String getActiveKeyId() {
        return redisStore.get(config.getActiveKeyIdKey());
    }

    public String getPreviousKeyId() {
        return redisStore.get(config.getPreviousKeyIdKey());
    }

    public void registerNewKey(String newKeyId) {
        String currentActive = getActiveKeyId();
        if (currentActive != null) {
            redisStore.set(config.getPreviousKeyIdKey(), currentActive, 0, null);
        }
        redisStore.set(config.getActiveKeyIdKey(), newKeyId, 0, null);
    }

    public String generateNewKeyId() {
        return "rsa-" + UUID.randomUUID();
    }
}


package org.kpi.api.key.service;

import org.kpi.api.key.config.RevocationConfig;
import org.kpi.api.key.storage.RedisStore;

import java.time.Instant;
import java.util.concurrent.TimeUnit;

public class RevocationService {
    private final RedisStore redisStore;
    private final EpochBloomFilterService epochBloomFilterService;
    private final RevocationConfig config;

    public RevocationService(RedisStore redisStore, EpochBloomFilterService
epochBloomFilterService,
                             RevocationConfig config) {
        this.redisStore = redisStore;
        this.epochBloomFilterService = epochBloomFilterService;
        this.config = config;
    }

    public long getCurrentRevocationEpoch() {
        String revEpochStr = redisStore.get(config.getEpochKey());
```

```
        return (revEpochStr != null) ? Long.parseLong(revEpochStr) : 1;
    }

    public long getCurrentRevocationIndex(long epoch) {
        String indexKey = config.getIndexPrefix() + epoch;
        String indexStr = redisStore.get(indexKey);
        return (indexStr != null) ? Long.parseLong(indexStr) : 0;
    }

    public long incrementRevocationIndex(long epoch) {
        String indexKey = config.getIndexPrefix() + epoch;
        return redisStore.incrementAndExpire(indexKey,
                                             (config.getEpochDurationSeconds() * 2) +
300,
                                             TimeUnit.SECONDS);
    }

    public void initializeEpochIfNeeded() {
        String currentEpochStr = redisStore.get(config.getEpochKey());
        if (currentEpochStr == null) {
            redisStore.setIfAbsent(config.getEpochKey(), "1", 0, null);
            String epochStartKey = config.getEpochStartPrefix() + "1";
            redisStore.setIfAbsent(epochStartKey,
String.valueOf(Instant.now().getEpochSecond()),
                                   (config.getEpochDurationSeconds() * 2) + 300,
TimeUnit.SECONDS);
        }
    }

    public EpochRotationResult manualRotateEpoch() {
        try {
            long currentEpoch = getCurrentRevocationEpoch();
            long currentTime = Instant.now().getEpochSecond();

            String epochStartKey = config.getEpochStartPrefix() + currentEpoch;
            String startTimeStr = redisStore.get(epochStartKey);
            long epochStartTime = (startTimeStr != null) ?
Long.parseLong(startTimeStr) : currentTime;

            long epochAge = currentTime - epochStartTime;

            EpochRotationResult result = rotateEpoch(currentEpoch);
            result.setEpochAge(epochAge);
            result.setManualRotation(true);

            return result;
        } catch (Exception e) {
            return EpochRotationResult.failure("Manual rotation failed: " +
e.getMessage());
        }
    }

    public boolean shouldRotateEpoch() {
        long currentEpoch = getCurrentRevocationEpoch();
        String epochStartKey = config.getEpochStartPrefix() + currentEpoch;
        String startTimeStr = redisStore.get(epochStartKey);

        if (startTimeStr == null) {
            redisStore.set(epochStartKey,
```

```
String.valueOf(Instant.now().getEpochSecond()),
                    (config.getEpochDurationSeconds() * 2) + 300, TimeUnit.SECONDS);
            return false;
        }

        long epochStartTime = Long.parseLong(startTimeStr);
        long currentTime = Instant.now().getEpochSecond();

        return currentTime - epochStartTime >= config.getEpochDurationSeconds();
    }

    private EpochRotationResult rotateEpoch(long currentEpoch) {
        long newEpoch = currentEpoch + 1;
        long currentTime = Instant.now().getEpochSecond();

        redisStore.set(config.getEpochKey(), String.valueOf(newEpoch), 0, null);

        String newEpochStartKey = config.getEpochStartPrefix() + newEpoch;
        redisStore.set(newEpochStartKey, String.valueOf(currentTime),
                (config.getEpochDurationSeconds() * 2) + 300, TimeUnit.SECONDS);

        String newIndexKey = config.getIndexPrefix() + newEpoch;
        redisStore.set(newIndexKey, "0",
                (config.getEpochDurationSeconds() * 2) + 300, TimeUnit.SECONDS);

        CleanupResult cleanupResult = cleanupOldEpochs(newEpoch);

        return EpochRotationResult.success(currentEpoch, newEpoch,
                cleanupResult.getClearedEpoch(),
                cleanupResult.getItemsCleared());
    }

    private CleanupResult cleanupOldEpochs(long currentEpoch) {
        long epochToClean = currentEpoch - 2;

        if (epochToClean <= 0) {
            return new CleanupResult(-1, 0);
        }

        int itemsCleared = 0;

        try {
            String oldEpochStartKey = config.getEpochStartPrefix() + epochToClean;
            if (redisStore.get(oldEpochStartKey) != null) {
                redisStore.delete(oldEpochStartKey);
                itemsCleared++;
            }

            String oldIndexKey = config.getIndexPrefix() + epochToClean;
            if (redisStore.get(oldIndexKey) != null) {
                redisStore.delete(oldIndexKey);
                itemsCleared++;
            }

            epochBloomFilterService.cleanupEpochData(epochToClean);
            itemsCleared += 2;
        } catch (Exception e) {
        }
```

```
        return new CleanupResult(epochToClean, itemsCleared);
    }

    public boolean isEpochActive(long epoch) {
        long currentEpoch = getCurrentRevocationEpoch();
        return epoch >= (currentEpoch - 1) && epoch <= currentEpoch;
    }

}


package org.kpi.api.key.service;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.kpi.api.key.config.TokenGenerationConfig;
import org.kpi.api.key.storage.KeyStorageException;
import org.kpi.api.key.storage.KeyStorageProvider;

import java.security.PrivateKey;
import java.time.Instant;
import java.util.*;

public class TokenGenerationService {
    private final TokenGenerationConfig config;
    private final KeyIdStoreService keyIdStoreService;
    private final KeyStorageProvider keyStorageProvider;
    private final RevocationService revocationService;

    public TokenGenerationService(TokenGenerationConfig config,
                                  KeyIdStoreService keyIdStoreService,
                                  KeyStorageProvider keyStorageProvider,
                                  RevocationService revocationService) {
        this.config = config;
        this.keyIdStoreService = keyIdStoreService;
        this.keyStorageProvider = keyStorageProvider;
        this.revocationService = revocationService;
    }

    public String generateToken(String subject, String audience, List<String> scopes,
Map<String, Object> pop) {
        try {
            String keyId = keyIdStoreService.getActiveKeyId();
            if (keyId == null) {
                throw new RuntimeException("No active key available for token
generation");
            }

            PrivateKey privateKey = keyStorageProvider.loadPrivateKey(keyId);
            Claims claims = createClaimsWithRevocationInfo(subject, audience, scopes,
pop);

            return Jwts.builder()
                    .setClaims(claims)
                    .setHeaderParam("kid", keyId)
                    .signWith(SignatureAlgorithm.RS256, privateKey)
                    .compact();
        } catch (KeyStorageException e) {
```

```
            throw new RuntimeException("Failed to load private key", e);
        } catch (Exception e) {
            throw new RuntimeException("Failed to generate token", e);
        }
    }

    private Claims createClaimsWithRevocationInfo(String subject, String audience,
List<String> scopes, Map<String, Object> pop) {
        Instant now = Instant.now();
        long currentEpoch = revocationService.getCurrentRevocationEpoch();
        long currentRevIndex =
revocationService.getCurrentRevocationIndex(currentEpoch);

        return Jwts.claims()
                .subject(subject)
                .audience().single(audience)
                .issuer(config.getIssuer())
                .issuedAt(Date.from(now))

.expiration(Date.from(now.plusSeconds(config.getDefaultExpirationSeconds())))
                .id(UUID.randomUUID().toString())
                .add("scopes", scopes)
                .add("pop", pop)
                .add("rev_epoch", currentEpoch)
                .add("rev_index", currentRevIndex)
                .build();
    }
}


package org.kpi.api.key.service;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.util.Base64;

public class TokenRevocationService {
    private final RevocationService revocationService;
    private final EpochBloomFilterService epochBloomFilterService;

    public TokenRevocationService(RevocationService revocationService,
                                  EpochBloomFilterService epochBloomFilterService) {
        this.revocationService = revocationService;
        this.epochBloomFilterService = epochBloomFilterService;
    }

    public RevocationResult revokeToken(String token) {
        try {
            String jti = extractJti(token);
            if (jti == null) {
                return RevocationResult.failure("Invalid token: Could not extract
JTI");
            }

            long currentEpoch = revocationService.getCurrentRevocationEpoch();

            if (isTokenRevoked(jti)) {
                return RevocationResult.failure("Token already revoked");
            }
```

```
            long newIndex = revocationService.incrementRevocationIndex(currentEpoch);
            epochBloomFilterService.addToEpochFilter(currentEpoch, jti);

            return RevocationResult.success("Token revoked successfully");
        } catch (Exception e) {
            return RevocationResult.failure("Error revoking token: " +
e.getMessage());
        }
    }

    public boolean isTokenRevoked(String jti) {
        long currentEpoch = revocationService.getCurrentRevocationEpoch();

        if (epochBloomFilterService.existsInEpochFilter(currentEpoch, jti)) {
            return true;
        }

        long previousEpoch = currentEpoch - 1;
        if (previousEpoch > 0 && revocationService.isEpochActive(previousEpoch)) {
            return epochBloomFilterService.existsInEpochFilter(previousEpoch, jti);
        }

        return false;
    }

    public boolean isTokenRevokedOptimized(String jti, long tokenEpoch, long
tokenRevIndex) {
        if (!revocationService.isEpochActive(tokenEpoch)) {
            return false;
        }

        long currentRevIndex =
revocationService.getCurrentRevocationIndex(tokenEpoch);

        if (tokenRevIndex >= currentRevIndex) {
            return false;
        }

        return epochBloomFilterService.existsInEpochFilter(tokenEpoch, jti);
    }

    private String extractJti(String token) {
        try {
            String[] parts = token.split("\\.");
            if (parts.length != 3) {
                return null;
            }

            String payload = parts[1];
            String decodedPayload = new
String(Base64.getUrlDecoder().decode(payload));

            if (decodedPayload.contains("\"jti\"")) {
                int jtiStart = decodedPayload.indexOf("\"jti\"");
                int valueStart = decodedPayload.indexOf(":", jtiStart) + 1;
                int valueEnd = decodedPayload.indexOf(",", valueStart);
                if (valueEnd == -1) {
                    valueEnd = decodedPayload.indexOf("}", valueStart);
```

```
                }
                String jtiValue = decodedPayload.substring(valueStart,
valueEnd).trim();
                return jtiValue.startsWith("\"") && jtiValue.endsWith("\"")
                        ? jtiValue.substring(1, jtiValue.length() - 1)
                        : jtiValue;
            }
            return null;
        } catch (Exception e) {
            return null;
        }
    }

}



public class TokenVerificationService {
    private final TokenGenerationConfig config;
    private final KeyIdStoreService keyIdStoreService;
    private final KeyStorageProvider keyStorageProvider;
    private final TokenRevocationService tokenRevocationService;
    private final Map<String, PublicKey> keyCache = new HashMap<>();

    public TokenVerificationService(TokenGenerationConfig config,
                                    KeyIdStoreService keyIdStoreService,
                                    KeyStorageProvider keyStorageProvider,
                                    TokenRevocationService tokenRevocationService) {
        this.config = config;
        this.keyIdStoreService = keyIdStoreService;
        this.keyStorageProvider = keyStorageProvider;
        this.tokenRevocationService = tokenRevocationService;
    }

    public TokenValidationResult validateToken(String token) {
        try {
            String keyId = extractKeyIdFromToken(token);
            if (keyId == null) {
                return TokenValidationResult.invalid("Missing key ID in token
header");
            }

            PublicKey publicKey = getPublicKey(keyId);
            if (publicKey == null) {
                return TokenValidationResult.invalid("Invalid key ID: " + keyId);
            }

            Jws<Claims> jws = Jwts.parser()
                    .verifyWith(publicKey)
                    .build()
                    .parseSignedClaims(token);

            Claims claims = jws.getBody();

            String issuer = claims.getIssuer();
            if (!config.getIssuer().equals(issuer)) {
                return TokenValidationResult.invalid("Invalid token issuer");
            }
```

```
            if (claims.getSubject() == null || claims.getAudience() == null ||
                    claims.getIssuedAt() == null || claims.getExpiration() == null ||
                    claims.getId() == null) {
                return TokenValidationResult.invalid("Missing required claims");
            }

            @SuppressWarnings("unchecked")
            Map<String, Object> pop = claims.get("pop", Map.class);
            if (pop == null || !pop.containsKey("alg")) {
                return TokenValidationResult.invalid("Invalid pop object");
            }

            Long tokenRevEpoch = claims.get("rev_epoch", Long.class);
            Long tokenRevIndex = claims.get("rev_index", Long.class);

            if (tokenRevEpoch == null || tokenRevIndex == null) {
                return TokenValidationResult.invalid("Missing revocation epoch or
index");
            }

            String jti = claims.getId();
            if (tokenRevocationService.isTokenRevokedOptimized(jti, tokenRevEpoch,
tokenRevIndex)) {
                return TokenValidationResult.revoked("Token has been explicitly
revoked");
            }

            return TokenValidationResult.valid(claims);
        } catch (ExpiredJwtException e) {
            return TokenValidationResult.invalid("Token has expired");
        } catch (SignatureException e) {
            return TokenValidationResult.invalid("Invalid token signature");
        } catch (MalformedJwtException e) {
            return TokenValidationResult.invalid("Malformed token");
        } catch (Exception e) {
            return TokenValidationResult.invalid("Error validating token: " +
e.getMessage());
        }
    }

    private String extractKeyIdFromToken(String token) {
        try {
            String[] parts = token.split("\\.");
            if (parts.length != 3) {
                return null;
            }

            String headerJson = new String(Base64.getUrlDecoder().decode(parts[0]));
            if (headerJson.contains("\"kid\"")) {
                int kidStart = headerJson.indexOf("\"kid\"");
                int valueStart = headerJson.indexOf(":", kidStart) + 1;
                int valueEnd = headerJson.indexOf(",", valueStart);
                if (valueEnd == -1) {
                    valueEnd = headerJson.indexOf("}", valueStart);
                }
                String kidValue = headerJson.substring(valueStart, valueEnd).trim();
                return kidValue.startsWith("\"") && kidValue.endsWith("\"")
                        ? kidValue.substring(1, kidValue.length() - 1)
                        : kidValue;
```

```
            }
            return null;
        } catch (Exception e) {
            return null;
        }
    }

    private PublicKey getPublicKey(String keyId) {
        if (keyCache.containsKey(keyId)) {
            return keyCache.get(keyId);
        }

        try {
            String activeKeyId = keyIdStoreService.getActiveKeyId();
            String previousKeyId = keyIdStoreService.getPreviousKeyId();

            if (!keyId.equals(activeKeyId) && !keyId.equals(previousKeyId)) {
                return null;
            }

            PublicKey publicKey = keyStorageProvider.loadPublicKey(keyId);
            keyCache.put(keyId, publicKey);
            return publicKey;
        } catch (KeyStorageException e) {
            return null;
        } catch (Exception e) {
            return null;
        }
    }

}
```

```
package org.kpi.api.key.storage;

public interface BloomFilterOperations {
    boolean addToFilter(String filterKey, String item);
    boolean existsInFilter(String filterKey, String item);
    void createFilter(String filterKey, double errorRate, long capacity);
    void deleteFilter(String filterKey);
    boolean filterExists(String filterKey);
}
```

```
package org.kpi.api.key.storage;

import java.security.PrivateKey;
import java.security.PublicKey;

public interface KeyStorageProvider {
    void storeKeyPair(String keyId, PrivateKey privateKey, PublicKey publicKey)
throws KeyStorageException;
    PrivateKey loadPrivateKey(String keyId) throws KeyStorageException;
    PublicKey loadPublicKey(String keyId) throws KeyStorageException;
    boolean testConnection() throws KeyStorageException;
}
```

```
package org.kpi.api.key.storage;

import java.util.concurrent.TimeUnit;

public interface RedisStore {
    void set(String key, String value, long timeout, TimeUnit unit);
    Boolean hasKey(String key);
    String get(String key);
    Boolean delete(String key);

    Long increment(String key);
    Long incrementAndExpire(String key, long timeout, TimeUnit unit);
    void setIfAbsent(String key, String value, long timeout, TimeUnit unit);
}
```

```
package org.kpi.api.key.config;

public class TokenGenerationConfig {
    private final String issuer;
    private final String signingKey;
    private final long defaultExpirationSeconds;

    public TokenGenerationConfig(String issuer, String signingKey, long
defaultExpirationSeconds) {
        this.issuer = issuer;
        this.signingKey = signingKey;
        this.defaultExpirationSeconds = defaultExpirationSeconds;
    }

    public String getIssuer() { return issuer; }
    public String getSigningKey() { return signingKey; }
    public long getDefaultExpirationSeconds() { return defaultExpirationSeconds; }
}
```

```
package org.kpi.api.key.config;

public class KeyGenerationConfig {
    private final String keysBasePath;
    private final String activeKeyIdKey;
    private final String previousKeyIdKey;
    private final boolean rotationEnabled;
    private final String rotationCron;

    public KeyGenerationConfig(String keysBasePath, String activeKeyIdKey, String
previousKeyIdKey,
                               boolean rotationEnabled, String rotationCron) {
        this.keysBasePath = keysBasePath;
        this.activeKeyIdKey = activeKeyIdKey;
        this.previousKeyIdKey = previousKeyIdKey;
        this.rotationEnabled = rotationEnabled;
        this.rotationCron = rotationCron;
    }

    public String getKeysBasePath() { return keysBasePath; }
    public String getActiveKeyIdKey() { return activeKeyIdKey; }
    public String getPreviousKeyIdKey() { return previousKeyIdKey; }
    public boolean isRotationEnabled() { return rotationEnabled; }
```

```
    public String getRotationCron() { return rotationCron; }
}
```

```
package org.kpi.api.key.config;

public class RevocationConfig {
    private final String epochKey;
    private final String indexPrefix;
    private final String epochStartPrefix;
    private final long epochDurationSeconds;
    private final long epochCheckInterval;
    private final String bloomFilterPrefix;
    private final String revokedJtiPrefix;
    private final double bloomFilterErrorRate;
    private final long bloomFilterCapacity;

    public RevocationConfig(String epochKey, String indexPrefix, String
epochStartPrefix,
                            long epochDurationSeconds, long epochCheckInterval,
                            String bloomFilterPrefix, String revokedJtiPrefix,
                            double bloomFilterErrorRate, long bloomFilterCapacity) {
        this.epochKey = epochKey;
        this.indexPrefix = indexPrefix;
        this.epochStartPrefix = epochStartPrefix;
        this.epochDurationSeconds = epochDurationSeconds;
        this.epochCheckInterval = epochCheckInterval;
        this.bloomFilterPrefix = bloomFilterPrefix;
        this.revokedJtiPrefix = revokedJtiPrefix;
        this.bloomFilterErrorRate = bloomFilterErrorRate;
        this.bloomFilterCapacity = bloomFilterCapacity;
    }

    public String getEpochKey() { return epochKey; }
    public String getIndexPrefix() { return indexPrefix; }
    public String getEpochStartPrefix() { return epochStartPrefix; }
    public long getEpochDurationSeconds() { return epochDurationSeconds; }
    public long getEpochCheckInterval() { return epochCheckInterval; }
    public String getBloomFilterPrefix() { return bloomFilterPrefix; }
    public String getRevokedJtiPrefix() { return revokedJtiPrefix; }
    public double getBloomFilterErrorRate() { return bloomFilterErrorRate; }
    public long getBloomFilterCapacity() { return bloomFilterCapacity; }
}
```

```
package org.kpi.api.key.config;

public class DpopConfig {
    private final long proofMaxAgeSeconds;
    private final String replayPrefix;

    public DpopConfig(long proofMaxAgeSeconds, String replayPrefix) {
        this.proofMaxAgeSeconds = proofMaxAgeSeconds;
        this.replayPrefix = replayPrefix;
    }

    public long getProofMaxAgeSeconds() { return proofMaxAgeSeconds; }
```

```
    public String getReplayPrefix() { return replayPrefix; }
}


package org.kpi.api.key.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface RequirePermissions {
    String[] value();
    PermissionMode mode() default PermissionMode.ALL;
    String description() default "";
    boolean requireDpop() default false;
    enum PermissionMode {
        ALL, ANY
    }
}

package org.kpi.api.key.config;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.kpi.api.key.annotation.RequirePermissions;
import org.kpi.api.key.service.ApiKeyService;
import org.kpi.api.key.service.ApiRequestValidationService;
import org.kpi.api.key.service.DpopValidationService;
import org.kpi.api.key.service.EndpointScopeRegistry;
import org.kpi.api.key.service.TokenVerificationService;
import org.springframework.stereotype.Component;
import org.springframework.web.method.HandlerMethod;
import org.springframework.web.servlet.HandlerInterceptor;

import java.net.URI;
import java.util.Set;

@Component
public class PermissionInterceptor implements HandlerInterceptor {

    private final ApiKeyService apiKeyService;
    private final EndpointScopeRegistry endpointRegistry;
    private final DpopValidationService dpopValidationService;
    private final TokenVerificationService tokenVerificationService;

    public PermissionInterceptor(ApiKeyService apiKeyService,
                                 EndpointScopeRegistry endpointRegistry,
                                 DpopValidationService dpopValidationService,
                                 TokenVerificationService tokenVerificationService) {
        this.apiKeyService = apiKeyService;
        this.endpointRegistry = endpointRegistry;
        this.dpopValidationService = dpopValidationService;
        this.tokenVerificationService = tokenVerificationService;
    }

    @Override
```

```
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response,
                             Object handler) throws Exception {

        if (!(handler instanceof HandlerMethod)) {
            return true;
        }

        HandlerMethod handlerMethod = (HandlerMethod) handler;

        RequirePermissions annotation =
handlerMethod.getMethodAnnotation(RequirePermissions.class);

        if (annotation == null) {
            annotation =
handlerMethod.getBeanType().getAnnotation(RequirePermissions.class);
        }

        Set<String> requiredPermissions = null;
        RequirePermissions.PermissionMode mode =
RequirePermissions.PermissionMode.ALL;
        boolean requiresDpop = false;

        if (annotation != null) {
            requiredPermissions = Set.of(annotation.value());
            mode = annotation.mode();
            requiresDpop = annotation.requireDpop();
        } else {
            String httpMethod = request.getMethod();
            String requestPath = request.getRequestURI();
            Set<String> registryScopes =
endpointRegistry.getRequiredScopes(httpMethod, requestPath);

            if (!registryScopes.isEmpty()) {
                requiredPermissions = registryScopes;
                mode = RequirePermissions.PermissionMode.ALL;
                requiresDpop = endpointRegistry.requiresDpop(httpMethod,
requestPath);
            }
        }

        if (requiredPermissions == null || requiredPermissions.isEmpty()) {
            return true;
        }

        String authHeader = request.getHeader("Authorization");
        if (authHeader == null) {
            sendErrorResponse(response, HttpServletResponse.SC_UNAUTHORIZED,
                              "Authorization header required", null, null);
            return false;
        }

        try {
            String token = extractToken(authHeader);

            TokenVerificationService.TokenValidationResult tokenResult =
                tokenVerificationService.validateToken(token);

            if (!tokenResult.isValid()) {
```

```
                sendErrorResponse(response, HttpServletResponse.SC_UNAUTHORIZED,
                                  tokenResult.getMessage(), null, null);
                return false;
            }

            Set<String> userScopes = apiKeyService.extractScopes(token);

            boolean hasPermission = checkPermissions(userScopes, requiredPermissions,
mode);
            if (!hasPermission) {
                sendErrorResponse(response, HttpServletResponse.SC_FORBIDDEN,
                                  "Insufficient permissions", requiredPermissions,
userScopes);
                return false;
            }

            if (requiresDpop) {
                String dpopHeader = request.getHeader("X-PoP-Sig");
                if (dpopHeader == null) {
                    sendErrorResponse(response, HttpServletResponse.SC_BAD_REQUEST,
                                      "DPoP proof required for this endpoint", null,
null);
                    return false;
                }

                boolean dpopValid = validateDpopProof(dpopHeader, token, request);
                if (!dpopValid) {
                    sendErrorResponse(response, HttpServletResponse.SC_FORBIDDEN,
                                      "Invalid DPoP proof", null, null);
                    return false;
                }
            }

            return true;
        } catch (Exception e) {
            sendErrorResponse(response, HttpServletResponse.SC_UNAUTHORIZED,
                              "Token validation error: " + e.getMessage(), null, null);
            return false;
        }
    }

    private boolean validateDpopProof(String dpopProof, String token,
HttpServletRequest request) {
        try {
            String httpMethod = request.getMethod();
            URI uri = new URI(request.getRequestURL().toString());

            DpopValidationService.DpopValidationResult result =
                dpopValidationService.validateDpop(dpopProof, token, httpMethod,
uri);

            return result.isValid();
        } catch (Exception e) {
            return false;
        }
    }

    private String extractToken(String authHeader) {
        if (authHeader.toLowerCase().startsWith("bearer ")) {
```

```
            return authHeader.substring(7);
        }
        return authHeader;
    }

    private boolean checkPermissions(Set<String> userScopes, Set<String>
requiredPermissions,
                                     RequirePermissions.PermissionMode mode) {
        if (mode == RequirePermissions.PermissionMode.ALL) {
            return userScopes.containsAll(requiredPermissions);
        } else {
            return requiredPermissions.stream().anyMatch(userScopes::contains);
        }
    }

    private void sendErrorResponse(HttpServletResponse response, int status, String
message,
                                   Set<String> required, Set<String> provided) throws
Exception {
        response.setStatus(status);
        response.setContentType("application/json");

        String errorJson;
        if (required != null && provided != null) {
            errorJson = String.format(
                "{\"error\":\"%s\",\"required\":%s,\"provided\":%s}",
                message,
                required.toString().replace("'", "\""),
                provided.toString().replace("'", "\"")
            );
        } else {
            errorJson = String.format("{\"error\":\"%s\"}", message);
        }

        response.getWriter().write(errorJson);
    }
}


package org.kpi.api.key.config;

import org.kpi.api.key.integration.FileSystemKeyStorageProvider;
import org.kpi.api.key.integration.SpringBloomFilterOperations;
import org.kpi.api.key.integration.SpringRedisStore;
import org.kpi.api.key.service.ApiKeyService;
import org.kpi.api.key.service.ApiRequestValidationService;
import org.kpi.api.key.service.DpopValidationService;
import org.kpi.api.key.service.EndpointScopeRegistry;
import org.kpi.api.key.service.EpochBloomFilterService;
import org.kpi.api.key.service.KeyGenerationService;
import org.kpi.api.key.service.KeyIdStoreService;
import org.kpi.api.key.service.RevocationService;
import org.kpi.api.key.service.TokenGenerationService;
import org.kpi.api.key.service.TokenRevocationService;
import org.kpi.api.key.service.TokenVerificationService;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.data.redis.core.StringRedisTemplate;

@Configuration
@EnableConfigurationProperties(ApiKeyLibraryProperties.class)
public class ApiKeyLibraryAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public TokenGenerationConfig tokenGenerationConfig(ApiKeyLibraryProperties
properties) {
        return new TokenGenerationConfig(
                properties.getIssuer(),
                properties.getSigningKey(),
                properties.getDefaultExpirationSeconds()
        );
    }

    @Bean
    @ConditionalOnMissingBean
    public DpopConfig dpopConfig(ApiKeyLibraryProperties properties) {
        return new DpopConfig(
                properties.getDpop().getProofMaxAgeSeconds(),
                properties.getDpop().getReplayPrefix()
        );
    }

    @Bean
    @ConditionalOnMissingBean
    public SpringRedisStore springRedisStore(StringRedisTemplate redisTemplate) {
        return new SpringRedisStore(redisTemplate);
    }

    @Bean
    @ConditionalOnMissingBean
    public DpopValidationService dpopValidationService(
            TokenVerificationService tokenVerificationService,
            SpringRedisStore redisStore,
            DpopConfig dpopConfig) {
        return new DpopValidationService(tokenVerificationService, redisStore,
dpopConfig);
    }

    @Bean
    @ConditionalOnMissingBean
    public KeyGenerationConfig keyGenerationConfig(ApiKeyLibraryProperties
properties) {
        return new KeyGenerationConfig(
                properties.getKeys().getBasePath(),
                properties.getKeys().getActiveKeyIdKey(),
                properties.getKeys().getPreviousKeyIdKey(),
                properties.getKeys().isRotationEnabled(),
                properties.getKeys().getRotationCron()
        );
    }

    @Bean
    @ConditionalOnMissingBean
    public RevocationConfig revocationConfig(ApiKeyLibraryProperties properties) {
        return new RevocationConfig(
```

```
                properties.getRevocation().getEpochKey(),
                properties.getRevocation().getIndexPrefix(),
                properties.getRevocation().getEpochStartPrefix(),
                properties.getRevocation().getEpochDurationSeconds(),
                properties.getRevocation().getEpochCheckInterval(),
                properties.getRevocation().getBloomFilterPrefix(),
                properties.getRevocation().getRevokedJtiPrefix(),
                properties.getRevocation().getBloomFilterErrorRate(),
                properties.getRevocation().getBloomFilterCapacity()
        );
    }

    @Bean
    @ConditionalOnMissingBean
    public FileSystemKeyStorageProvider
fileSystemKeyStorageProvider(KeyGenerationConfig config) {
        return new FileSystemKeyStorageProvider(config.getKeysBasePath());
    }

    @Bean
    @ConditionalOnMissingBean
    public SpringBloomFilterOperations
springBloomFilterOperations(ApiKeyLibraryProperties properties) {
        return new SpringBloomFilterOperations(
                properties.getRedis().getHost(),
                properties.getRedis().getPort()
        );
    }

    @Bean
    @ConditionalOnMissingBean
    public KeyIdStoreService keyIdStoreService(SpringRedisStore redisStore,
KeyGenerationConfig config) {
        return new KeyIdStoreService(redisStore, config);
    }

    @Bean
    @ConditionalOnMissingBean
    public KeyGenerationService keyGenerationService(
            FileSystemKeyStorageProvider keyStorageProvider,
            KeyIdStoreService keyIdStoreService) {
        return new KeyGenerationService(keyStorageProvider, keyIdStoreService);
    }

    @Bean
    @ConditionalOnMissingBean
    public EpochBloomFilterService epochBloomFilterService(
            SpringBloomFilterOperations bloomFilterOps,
            SpringRedisStore redisStore,
            RevocationConfig config) {
        return new EpochBloomFilterService(bloomFilterOps, redisStore, config);
    }

    @Bean
    @ConditionalOnMissingBean
    public RevocationService revocationService(
            SpringRedisStore redisStore,
            EpochBloomFilterService epochBloomFilterService,
            RevocationConfig config) {
```

```
        return new RevocationService(redisStore, epochBloomFilterService, config);
    }

    @Bean
    @ConditionalOnMissingBean
    public TokenRevocationService tokenRevocationService(
            RevocationService revocationService,
            EpochBloomFilterService epochBloomFilterService) {
        return new TokenRevocationService(revocationService,
epochBloomFilterService);
    }

    @Bean
    @ConditionalOnMissingBean
    public TokenGenerationService tokenGenerationService(
            TokenGenerationConfig config,
            KeyIdStoreService keyIdStoreService,
            FileSystemKeyStorageProvider keyStorageProvider,
            RevocationService revocationService) {
        return new TokenGenerationService(config, keyIdStoreService,
keyStorageProvider, revocationService);
    }

    @Bean
    @ConditionalOnMissingBean
    public TokenVerificationService tokenVerificationService(
            TokenGenerationConfig config,
            KeyIdStoreService keyIdStoreService,
            FileSystemKeyStorageProvider keyStorageProvider,
            TokenRevocationService tokenRevocationService) {
        return new TokenVerificationService(config, keyIdStoreService,
keyStorageProvider, tokenRevocationService);
    }

    @Bean
    @ConditionalOnMissingBean
    public EndpointScopeRegistry endpointScopeRegistry() {
        return new EndpointScopeRegistry();
    }

    @Bean
    @ConditionalOnMissingBean
    public ApiRequestValidationService
apiRequestValidationService(EndpointScopeRegistry registry) {
        return new ApiRequestValidationService(registry);
    }

    @Bean
    @ConditionalOnMissingBean
    public ApiKeyService apiKeyService(TokenVerificationService
tokenVerificationService) {
        return new ApiKeyService(tokenVerificationService);
    }
}



package org.kpi.api.key.integration;
```

```
import org.kpi.api.key.storage.RedisStore;
import org.springframework.data.redis.core.StringRedisTemplate;

import java.util.concurrent.TimeUnit;

public class SpringRedisStore implements RedisStore {
    private final StringRedisTemplate redisTemplate;

    public SpringRedisStore(StringRedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    @Override
    public void set(String key, String value, long timeout, TimeUnit unit) {
        if (timeout > 0 && unit != null) {
            redisTemplate.opsForValue().set(key, value, timeout, unit);
        } else {
            redisTemplate.opsForValue().set(key, value);
        }
    }

    @Override
    public Boolean hasKey(String key) {
        return redisTemplate.hasKey(key);
    }

    @Override
    public String get(String key) {
        return redisTemplate.opsForValue().get(key);
    }

    @Override
    public Boolean delete(String key) {
        return redisTemplate.delete(key);
    }

    @Override
    public Long increment(String key) {
        return redisTemplate.opsForValue().increment(key);
    }

    @Override
    public Long incrementAndExpire(String key, long timeout, TimeUnit unit) {
        Long value = redisTemplate.opsForValue().increment(key);
        redisTemplate.expire(key, timeout, unit);
        return value;
    }

    @Override
    public void setIfAbsent(String key, String value, long timeout, TimeUnit unit) {
        redisTemplate.opsForValue().setIfAbsent(key, value, timeout, unit);
    }
}


package org.kpi.api.key.integration;

import org.kpi.api.key.storage.BloomFilterOperations;
import redis.clients.jedis.Jedis;
```

```
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
import redis.clients.jedis.bloom.RedisBloomProtocol;

public class SpringBloomFilterOperations implements BloomFilterOperations {
    private final JedisPool jedisPool;

    public SpringBloomFilterOperations(String redisHost, int redisPort) {
        JedisPoolConfig poolConfig = new JedisPoolConfig();
        poolConfig.setMaxTotal(10);
        poolConfig.setMaxIdle(5);
        this.jedisPool = new JedisPool(poolConfig, redisHost, redisPort);
    }

    @Override
    public boolean addToFilter(String filterKey, String item) {
        try (Jedis jedis = jedisPool.getResource()) {
            Long result = (Long) jedis.sendCommand(
                    RedisBloomProtocol.BloomFilterCommand.ADD,
                    filterKey,
                    item
            );
            return result == 1;
        } catch (Exception e) {
            return false;
        }
    }

    @Override
    public boolean existsInFilter(String filterKey, String item) {
        try (Jedis jedis = jedisPool.getResource()) {
            Long result = (Long) jedis.sendCommand(
                    RedisBloomProtocol.BloomFilterCommand.EXISTS,
                    filterKey,
                    item
            );
            return result == 1;
        } catch (Exception e) {
            return false;
        }
    }

    @Override
    public void createFilter(String filterKey, double errorRate, long capacity) {
        try (Jedis jedis = jedisPool.getResource()) {
            jedis.sendCommand(
                    RedisBloomProtocol.BloomFilterCommand.RESERVE,
                    filterKey,
                    String.valueOf(errorRate),
                    String.valueOf(capacity)
            );
        } catch (Exception e) {
        }
    }

    @Override
    public void deleteFilter(String filterKey) {
        try (Jedis jedis = jedisPool.getResource()) {
            jedis.del(filterKey);
```

```
        } catch (Exception e) {
        }
    }

    @Override
    public boolean filterExists(String filterKey) {
        try (Jedis jedis = jedisPool.getResource()) {
            return jedis.exists(filterKey);
        } catch (Exception e) {
            return false;
        }
    }
}


package org.kpi.api.key.integration;

import org.kpi.api.key.storage.KeyStorageException;
import org.kpi.api.key.storage.KeyStorageProvider;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;

public class FileSystemKeyStorageProvider implements KeyStorageProvider {
    private final String basePath;

    public FileSystemKeyStorageProvider(String basePath) {
        this.basePath = basePath;
    }

    @Override
    public void storeKeyPair(String keyId, PrivateKey privateKey, PublicKey
publicKey) throws KeyStorageException {
        try {
            Path keyDir = Paths.get(basePath);
            Files.createDirectories(keyDir);

            String privateKeyPem = "-----BEGIN PRIVATE KEY-----\n" +
                    Base64.getEncoder().encodeToString(privateKey.getEncoded()) +
                    "\n-----END PRIVATE KEY-----";
            Files.write(keyDir.resolve(keyId + ".private.pem"),
privateKeyPem.getBytes());

            String publicKeyPem = "-----BEGIN PUBLIC KEY-----\n" +
                    Base64.getEncoder().encodeToString(publicKey.getEncoded()) +
                    "\n-----END PUBLIC KEY-----";
            Files.write(keyDir.resolve(keyId + ".public.pem"),
publicKeyPem.getBytes());

        } catch (IOException e) {
            throw new KeyStorageException("Failed to store key pair", e);
```

```
        }
    }

    @Override
    public PrivateKey loadPrivateKey(String keyId) throws KeyStorageException {
        try {
            String privateKeyPath = basePath + "/" + keyId + ".private.pem";
            String pem = new String(Files.readAllBytes(Paths.get(privateKeyPath)));
            String privateKeyPEM = pem
                    .replace("-----BEGIN PRIVATE KEY-----", "")
                    .replace("-----END PRIVATE KEY-----", "")
                    .replaceAll("\\s", "");
            byte[] encoded = Base64.getDecoder().decode(privateKeyPEM);
            PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(encoded);
            KeyFactory kf = KeyFactory.getInstance("RSA");
            return kf.generatePrivate(keySpec);
        } catch (Exception e) {
            throw new KeyStorageException("Failed to load private key", e);
        }
    }

    @Override
    public PublicKey loadPublicKey(String keyId) throws KeyStorageException {
        try {
            String publicKeyPath = basePath + "/" + keyId + ".public.pem";
            String pem = new String(Files.readAllBytes(Paths.get(publicKeyPath)));
            String publicKeyPEM = pem
                    .replace("-----BEGIN PUBLIC KEY-----", "")
                    .replace("-----END PUBLIC KEY-----", "")
                    .replaceAll("\\s", "");
            byte[] encoded = Base64.getDecoder().decode(publicKeyPEM);
            X509EncodedKeySpec keySpec = new X509EncodedKeySpec(encoded);
            KeyFactory kf = KeyFactory.getInstance("RSA");
            return kf.generatePublic(keySpec);
        } catch (Exception e) {
            throw new KeyStorageException("Failed to load public key", e);
        }
    }

    @Override
    public boolean testConnection() throws KeyStorageException {
        try {
            Path keyDir = Paths.get(basePath);
            return Files.exists(keyDir) || Files.createDirectories(keyDir) != null;
        } catch (IOException e) {
            throw new KeyStorageException("Storage test failed", e);
        }
    }
}
```