Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Система управління API-ключами та доступом до сервісів

Виконав: студент IV курсу, групи СН-41
спеціальності 122 Комп’ютерні науки
(шифр і назва спеціальності)

(підпис) Волянюк О.Р. (прізвище та ініціали)

Керівник (підпис) Гром'як Р.С. (прізвище та ініціали)

Нормоконтроль (підпис) Шимчук Г.В. (прізвище та ініціали)

Завідувач кафедри (підпис) Боднарчук І.О. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«___» _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

Студенту Волянюку Олегу Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Система управління API-ключами та доступом до сервісів

Керівник роботи Гром'як Роман Сильвестрович, к.ф-м.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*14*» *травня* *2026* року № *4/9-239*

2. Термін подання студентом завершеної роботи 20 червня 2026р.

3. Вихідні дані до роботи Літературні та інтернет-джерела про API-ключі, автентифікацію, авторизацію та захист API. Документація з FastAPI, , SQLAlchemy, JWT, Pydantic і матеріали щодо керування доступом до сервісів.

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
1) Аналіз завдання та огляд предметної області
2) Реалізація системи управління арі-ключами та доступом до сервісів
3) Тестування роботи системи управління арі-ключами та доступом до сервісів
4) Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титульна сторінка. 2. Актуальність дослідження. 3. Мета, Об'єкт, Предмет дослідження. 4. Завдання дослідження. 5. API-ключі як механізм доступу до сервісів. 7. Архітектура розробленої системи. 8. Структура бази даних системи. 9. JWT-автентифікація та доступ на основі ролей. 10. Реалізація сервісів і scope-дозволів. 11. Створення та зберігання API-ключів. 12. Перевірка доступу до захищених сервісів. 13. Ротація, відкликання та обмеження кількості запитів. 14. Журнал аудиту та фіксація подій 15. Результати тестування системи. 16. Висновки. 17. Завершальний.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Мариненко Сергій Юрійович, к.т.н. доцент кафедри МТ		

7. Дата видачі завдання 29 січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	30.01.2026	*Виконано*
2.	Підбір джерел по темі кваліфікаційної роботи	31.01.2026-03.02.2026	*Виконано*
3.	Опрацювання джерел по темі кваліфікаційної роботи	04.02.2026-06.02.2026	*Виконано*
4.	Виконання дослідження щодо проєктування,	07.02.2026-11.02.2026	*Виконано*
	реалізації та тестування системи управління		
	API-ключами і доступом до сервісів		
5.	Оформлення розділу «Аналіз завдання та огляд	17.06.2026-23.05.2026	*Виконано*
	предметної області»		
6.	Оформлення розділу «Реалізація системи управління	24.05.2026-28.05.2026	*Виконано*
	арі-ключами та доступом до сервісів»		
6.	Оформлення розділу «Тестування роботи системи	29.05.2026-04.06.2026	*Виконано*
	управління арі-ключами та доступом до сервісів»		
7.	Виконання завдання до розділу «Безпека життєдіяльності»	05.06.2026-06.06.2026	*Виконано*
8.	Виконання завдання до підрозділу «Основи охорони	07.06.2026-08.06.2026	*Виконано*
	праці»		
9.	Оформлення кваліфікаційної роботи	09.06.2026-10.06.2026	*Виконано*
10.	Нормоконтроль	11.06.2026-13.06.2026	*Виконано*
11.	Перевірка на плагіат	14.06.2026	*Виконано*
12.	Попередній захист кваліфікаційної роботи	15.06.2026	*Виконано*
13.	Захист кваліфікаційної роботи	22.06.2026	

Студент _____ Волянюк О.Р.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гром'як Р.С.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Система управління API-ключами та доступом до сервісів // Кваліфікаційна робота освітнього рівня «Бакалавр» // Волянюк Олег Романович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СН-41 // Тернопіль, 2026 // С.98, рис. – 68, табл. – 0, кресл. – 17, додат. – 0, бібліогр. – 35.

Ключові слова: API Key, Access Management, Authentication, Authorization, JWT, FastAPI, Scope, Rate Limiting, Audit Log, SQLAlchemy.

У кваліфікаційній роботі бакалавра розглянуто питання розроблення системи управління API-ключами та доступом до сервісів. Метою дослідження було проєктування, реалізація та тестування програмного рішення для централізованого створення, перевірки, відкликання та ротації API-ключів. У роботі проаналізовано механізми автентифікації й авторизації, використання JWT-токенів, scope-дозволів, обмеження кількості запитів і журналювання подій. Практична частина передбачала створення серверної системи на основі FastAPI з використанням SQLAlchemy, рольового доступу для адміністратора, розробника й аудитора, а також механізму перевірки API-ключів для захищених сервісів. Результатом роботи стало функціональне рішення, яке дозволяє керувати сервісами, генерувати API-ключі, призначати дозволи, контролювати доступ і фіксувати важливі дії в журналі аудиту. Проведене тестування підтвердило коректність роботи основних функцій системи та доцільність її використання для централізованого керування доступом у програмних і корпоративних середовищах.

ANNOTATION

API Key Management and Service Access Control System // Qualification work of the educational level "Bachelor" // Volianiuk Oleh // Ternopil Ivan Pulyu National Technical University, Computer and Information Systems and Software Engineering Faculty, Computer Sciences Department, group SN-41 // Ternopil, 2026 // P. 98, fig. - 68, tabl. - 0, drawings - 17, annexes. – 0, references - 35.

Keywords: API Key, Access Management, Authentication, Authorization, JWT, FastAPI, Scope, Rate Limiting, Audit Log, SQLAlchemy.

The bachelor's qualification thesis examines the development of a system for managing API keys and access to services. The aim of the research was to design, implement, and test a software solution for the centralized creation, verification, revocation, and rotation of API keys. The paper analyzes authentication and authorization mechanisms, the use of JWT tokens, scope-based permissions, request rate limiting, and event logging.The practical part involved the creation of a server-side system based on FastAPI using SQLAlchemy, role-based access for an administrator, developer, and auditor, as well as an API key verification mechanism for protected services. The result of the work is a functional solution that enables service management, API key generation, permission assignment, access control, and the recording of important actions in an audit log.The conducted testing confirmed the correct operation of the system's main functions and the feasibility of using it for centralized access management in software and corporate environments.

ПЕРЕЛІК СКОРОЧЕНЬ

API (англ. Application Programming Interface) — програмний інтерфейс застосунку.

JWT (англ. JSON Web Token) — вебтокен у форматі JSON.

OAuth 2.0 (англ. Open Authorization 2.0) — відкритий протокол авторизації версії 2.0.

JSON (англ. JavaScript Object Notation) — текстовий формат обміну даними.

IAM (англ. Identity and Access Management) — керування ідентифікацією та доступом.

RBAC (англ. Role-Based Access Control) — рольова модель керування доступом.

ACL (англ. Access Control List) — список контролю доступу.

ABAC (англ. Attribute-Based Access Control) — атрибутна модель керування доступом.

ReBAC (англ. Relationship-Based Access Control) — модель керування доступом на основі зв’язків між сутностями.

SIEM (англ. Security Information and Event Management) — система керування подіями та інформацією безпеки.

AWS (англ. Amazon Web Services) — хмарна платформа Amazon Web Services.

ORM (англ. Object-Relational Mapping) — об’єктно-реляційне відображення.

SQL (англ. Structured Query Language) — структурована мова запитів.

CI/CD (англ. Continuous Integration / Continuous Delivery) — безперервна інтеграція та безперервна доставка.

CRUD (англ. Create, Read, Update, Delete) — базові операції створення, читання, оновлення та видалення даних.

ЗМІСТ

ВСТУП

Актуальність теми. В умовах активного розвитку вебсервісів, мікросервісної архітектури, хмарних платформ та інтеграції різних інформаційних систем особливої актуальності набуває питання безпечного керування доступом до API. Сучасні програмні системи дедалі частіше взаємодіють між собою через прикладні програмні інтерфейси, тому виникає потреба у надійних механізмах ідентифікації клієнтів, контролю прав доступу, обмеження кількості запитів і журналювання подій. Одним із поширених засобів такого контролю є API-ключі, які дозволяють пов’язати запит із конкретним користувачем, сервісом або клієнтським застосунком. Проте самі по собі API-ключі не забезпечують повноцінного захисту, якщо не поєднуються з перевіркою прав, обмеженням області дії, ротацією, відкликанням, аудитом та додатковими механізмами автентифікації. Саме тому розроблення централізованої системи управління API-ключами та доступом до сервісів є актуальним завданням для сучасних інформаційних систем.

Мета і задачі дослідження. Метою даної роботи є проєктування, реалізація та тестування системи управління API-ключами та доступом до сервісів, яка забезпечує централізоване створення, перевірку, відкликання, ротацію ключів, контроль прав доступу та журналювання подій.

Для досягнення поставленої мети було сформульовано такі основні задачі:

- провести аналіз механізму доступу до сервісів на основі API-ключів;
- дослідити сучасні підходи до захисту API;
- розглянути використання JWT-токенів, ролей та scope-дозволів;
- спроєктувати архітектуру системи;
- розробити структуру бази даних;
- реалізувати серверну частину системи на основі FastAPI та SQLAlchemy;
- реалізувати створення, перевірку, відкликання та ротацію API-ключів;

– забезпечити контроль кількості запитів і журналювання подій;

– провести тестування основних сценаріїв роботи системи.

Об’єкт дослідження. Об’єктом дослідження є процеси керування доступом до програмних сервісів через API в сучасних інформаційних системах.

Предмет дослідження. Предметом дослідження є методи, засоби та програмні механізми створення, перевірки, відкликання, ротації API-ключів, а також реалізація рольового доступу, scope-дозволів, журналювання подій і контролю кількості запитів.

Практичне значення одержаних результатів. Практичне значення дослідження полягає у розробці функціональної системи управління API-ключами та доступом до сервісів, яка може бути використана як основа для захисту внутрішніх або зовнішніх API. Реалізоване програмне рішення дозволяє централізовано створювати API-ключі, прив’язувати їх до конкретних сервісів і дозволів, перевіряти доступ під час кожного запиту, відкликати або проводити заміну ключів, а також фіксувати важливі дії в журналі аудиту. Такий підхід підвищує рівень безпеки програмної системи, спрощує адміністрування доступу і дає змогу контролювати використання сервісів. Отримані результати можуть бути використані для подальшого розвитку інформаційних системи, інтеграції з іншими механізмами автентифікації та авторизації, а також у навчальних цілях під час вивчення захисту API та побудови серверних застосунків.

РОЗДІЛ 1. АНАЛІЗ ЗАВДАННЯ ТА ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 API-ключі як механізм автентифікації в сучасних інформаційних системах

API-ключ у загальному розумінні – це рядковий ідентифікатор, який клієнт передає під час звернення до API для того, щоб система могла співвіднести запит із конкретним застосунком, проєктом або споживачем сервісу. У документації Google Cloud API key описується як ключ, що дозволяє асоціювати запит із проєктом для цілей квот і білінгу, а в документації Apigee наголошується, що API key, або consumer key, є рядковим значенням, яке клієнтський застосунок передає API-проксі й яке однозначно ідентифікує цей застосунок. Отже, вже на рівні базових визначень видно, що ключ виконує насамперед функцію ідентифікації джерела запиту та прив’язки виклику до певного контексту використання API [1].

Принцип роботи API-ключа достатньо простий, що і зробило цей механізм масовим. Спочатку система або шлюз API генерує ключ і прив’язує його до певної сутності: проєкту, клієнтського застосунку, набору API, продукту, тарифного плану чи політики використання. Далі клієнт надсилає цей ключ разом із кожним запитом, а серверна частина перевіряє, чи існує такий ключ, чи має він дозволений стан, чи пов’язаний із потрібним ресурсом, чи не скасований і чи не порушує задані обмеження. В Apigee ключ пов’язується з developer app та API product, після чого проксі перевіряє дійсність ключа, стан застосунку, активність розробника, доступність потрібного шляху та відсутність відкликання або прострочення. В AWS API Gateway окремі методи можна позначити як такі, що вимагають API key, а сам ключ додатково зв’язується з usage plan для контролю використання [2].

Саме простота впровадження є основною перевагою API-ключів. Вони не потребують складного діалогу між клієнтом і сервером, як у повноцінних токенних схемах, легко інтегруються в скрипти, серверні сервіси, CI/CD-процеси, партнерські інтеграції та API, що надають доступ до публічних або низькоризикових ресурсів [3]. Крім того, ключі добре поєднуються з бізнесовими сценаріями: вони дозволяють рахувати споживання API, застосовувати квоти, прив’язувати виклики до тарифних планів і виявляти зловживання. OWASP прямо зазначає, що API-ключі часто застосовуються для протидії “фермуванню” API, для зниження впливу відмови в обслуговуванні та для монетизації сервісів, коли клієнт отримує доступ відповідно до купленого плану [4].

Водночас саме простота є і головним обмеженням API-ключів [5]. На відміну від повноцінних механізмів автентифікації суб’єкта, типовий API-ключ часто не несе інформації про конкретного користувача, не підтверджує його особу в строгому сенсі й не підтримує складні сценарії делегування прав. У сучасній документації Google Cloud це сформульовано особливо чітко: стандартний API-ключ не автентифікує principal, а отже, без principal система не може застосувати перевірки IAM, щоб визначити, чи має той хто здійснює виклик право на потрібну операцію. Це означає, що стандартний API-ключ у багатьох системах є швидше засобом слабкої автентифікації або ідентифікації клієнтського застосунку, ніж повноцінною автентифікацією суб’єкта доступу [6].

Звідси випливає важлива термінологічна різниця між автентифікацією та авторизацією. Згідно з NIST SP 800-63B, автентифікація пов’язана з віддаленим підтвердженням того, що суб’єкт є саме тим, за кого себе видає, а OWASP у рекомендаціях з авторизації наголошує, що авторизація – це перевірка того, чи дозволена певна дія або сервіс для конкретної сутності, і що вона відрізняється від автентифікації. Коли в системі використовується лише API-ключ, то перевіряється здебільшого право певного клієнтського застосунку звертатися

до API, але не завжди підтверджується особа кінцевого користувача і не завжди реалізується гнучка модель прав на рівні окремих дій чи об’єктів [7,8].

Порівняння API-ключів з іншими засобами автентифікації чітко показує, чому ключ не можна ототожнювати з паролем, client secret або access token. У моделі OAuth 2.0 ідентифікатор клієнта сам по собі не є секретом і не повинен використовуватися як єдиний доказ автентичності клієнта. OAuth 2.0, згідно з RFC 6749, призначений для отримання обмеженого доступу до HTTP-сервісу від імені власника ресурсу або від імені самого клієнта без передачі клієнту облікових даних користувача [9-11]. Отже, якщо API-ключ зазвичай є статичним артефактом доступу, то в OAuth центральним елементом є тимчасовий токен, отриманий за формалізованим протоколом авторизації [12].

Особливо показовим є порівняння з JWT. RFC 7519 визначає JWT як компактний URL-safe формат передавання тверджень між сторонами, де набір claims може бути підписаний або захищений MAC чи шифруванням. На відміну від API-ключа, JWT, як правило, містить не лише ознаку ініціатора запиту, а й структуровані claims, необхідні для прийняття рішень щодо доступу. Стандарт описує зареєстровані claims, зокрема iss, aud, exp, nbf, iat і jti; claim exp визначає момент, після якого токен не повинен прийматися до обробки, aud вимагає, щоб споживач токена ідентифікував себе серед адресатів, а jti може застосовуватися для запобігання повторному використанню токена. Тому JWT здатний переносити контекст авторизації, часові межі чинності та інші атрибути, які звичайний API-ключ зазвичай не несе [13].

Для безпечного використання API-ключів критично важливе не лише саме існування ключа, а й спосіб його передавання та зберігання. OWASP вимагає використовувати лише HTTPS для REST-сервісів, оскільки це захищає облікові дані в каналі зв’язку. Документація Google Cloud додатково рекомендує не передавати ключ як query parameter, оскільки тоді він потрапляє в URL і може бути викрадений через журнали, сканування чи інші механізми спостереження; натомість варто використовувати HTTP headers. Також Google прямо радить не вбудовувати ключі у вихідний код, не зберігати їх у

репозиторіях, тримати їх поза деревом вихідних кодів, видаляти невикористані ключі та регулярно виконувати ротацію [14].

Окремим елементом надійного підходу є обмеження області дії ключа. У сучасних хмарних системах це реалізується не лише через факт існування ключа, а через звуження його допустимих сценаріїв використання [15]. У Google Cloud для цього передбачені API restrictions, що дозволяють використання ключа тільки з певним набором API, а також application restrictions, які обмежують ключ конкретними вебсайтами, IP-адресами, Android-застосунками або iOS bundle ID. Такий підхід зменшує наслідки компрометації. Навіть якщо ключ буде вкрадено, він не зможе бути довільно використаний у будь-якому сервісі та з будь-якого клієнта.

У підсумку API-ключ доцільно розглядати як базовий і зручний засіб контролю доступу на рівні застосунку або інтеграції, але не як універсальний замінник повноцінної автентифікації користувача та гнучкої авторизації. Для публічних, партнерських, демонстраційних або квотових API він може бути цілком достатнім, особливо якщо поєднується з TLS, обмеженнями джерела, ротацією та моніторингом. Для чутливих ресурсів, дій з високою вартістю або сценаріїв делегування прав API-ключ має використовуватися або як допоміжний ідентифікатор клієнта, або в комбінації з сильнішими механізмами – насамперед OAuth 2.0, токенами доступу та більш формалізованими моделями авторизації.

1.2 Аналіз існуючих підходів до управління доступом та захисту API

Захист API не зводиться до наявності ключа або токена. Фактично безпечна модель складається з кількох взаємопов’язаних рівнів: підтвердження походження запиту, визначення дозволених дій, перевірки доступу до кожного конкретного ресурсу, транспортного захисту, обмеження інтенсивності звернень і засобів спостереження за всіма значущими подіями. OWASP у рекомендаціях щодо авторизації підкреслює три базові принципи: мінімально

необхідні привілеї, заборона за замовчуванням і валідація дозволів під час кожного запиту. Це означає, що безпечний API має не “довіряти” вже автентифікованому ініціатору запиту автоматично, а систематично перевіряти, чи дозволена саме теперішня дія щодо саме теперішнього об’єкта.

Якщо перенести ці принципи на систему керування API-ключами, стає зрозуміло, що особливе значення має модель ролей користувачів самої системи. Із аналізу сучасних платформ можна зробити висновок, що мінімально життєздатна модель передбачає щонайменше розмежування між роллю перегляду, роллю адміністрування та роллю власника клієнтського застосунку або інтеграції. Google API Keys API на рівні IAM відрізняє ролі viewer та admin, а Apigee будує доступ навколо сутностей developer app, API product і політик валідації. Звідси випливає практичний висновок, що користувач, який лише переглядає ключі або читає журнали, не повинен мати права їх створювати, змінювати чи видаляти, а власник інтеграції не повинен автоматично отримувати необмежені права на всю інфраструктуру API.

Одним із базових підходів до опису доступу є ACL. У класичному визначенні ACL являє собою список ACE-записів, де кожен запис указує довірену сутність і задає, які саме права для неї дозволені, заборонені або підпадають під аудит. Microsoft також розрізняє DACL, яка визначає дозволи та заборони доступу, і SACL, яка задає, які спроби доступу повинні породжувати записи аудиту. Для API-підходу ACL привабливий тим, що дає дуже пряме, майже об’єктне відображення політики. Можна явно задати, хто має право викликати конкретний маршрут, метод або ресурс [16].

Однак у великій API-екосистемі ACL масштабується погано. Якщо ресурсів багато, якщо маршрути мають різні контексти доступу, якщо сутності доступу постійно додаються або змінюються, то кількість правил стрімко зростає, а їх централізоване супроводження стає складним [17]. Саме тому ACL доречний там, де об’єктів мало або політика доступу дуже локалізована, але в корпоративних API-платформах він часто потребує доповнення іншими моделями – передусім рольовою або основі атрибутів. Такий висновок логічно

випливає з властивостей ACL як списку індивідуальних правил і з вимоги OWASP не покладатися на фрагментарні або випадково пропущені перевірки доступу.

Протилежним за логікою підходом є RBAC – role-based access control. NIST визначає RBAC як модель, у якій доступ контролюється через ролі, а дозволені дії асоціюються не з конкретними суб'єктами, а саме з ролями. У межах NIST-підходу RBAC був стандартизований як ANSI/INCITS 359 і підтримує ієрархії ролей та багато-до-багатьох зв'язки між користувачами, ролями і дозволами. Практична цінність RBAC полягає в тому, що він відображає організаційні функції адміністратор, аудитор, розробник, оператор і значно знижує вартість адміністрування порівняно з індивідуальними ACL для кожного користувача [18].

Разом з тим RBAC має і власні обмеження. OWASP звертає увагу, що для прикладних систем дедалі частіше доцільно віддавати перевагу ABAC або ReBAC, оскільки RBAC погано працює з тонкими умовами на рівні окремих об'єктів, атрибутів середовища чи взаємозв'язків між сутностями. У контексті API це видно особливо добре. Роль "інтегратор" або "клієнт" ще не відповідає на питання, чи має право цей клієнт звернутися саме до цього ресурсу, саме в цей час, саме з цього джерела, саме в межах цього середовища або тарифного плану. Тому для системи API-ключів RBAC доцільно використовувати як основу адміністративного доступу до самої панелі керування, але не як єдину модель для всіх правил виконання запитів.

Сучасні API-платформи поєднують RBAC із прикладними політиками доступу. Це видно на прикладі API Keys API в Google Cloud, де IAM визначає, хто може створювати, оновлювати чи переглядати ключі, а окремі restrictions визначають, до яких API, сайтів, IP-адрес чи застосунків може бути прив'язаний сам ключ. Така комбінація адміністративного RBAC і прикладних обмежень ключа є фактично гібридною моделлю, де права на керування секретом відокремлюються від технічних меж використання цього секрету в робочих викликах.

Окреме місце серед підходів до доступу посідає OAuth 2.0. Згідно з RFC 6749, цей фреймворк дозволяє сторонньому застосунку отримати обмежений доступ до HTTP-сервісу або від імені власника ресурсу, або від власного імені. У моделі OAuth фігурують чотири ролі: resource owner, client, authorization server і resource server. Така архітектура принципово відрізняється від API-ключа. Клієнт не повинен володіти обліковими даними користувача, а отримує access token за формалізованою процедурою, у межах якої authorization server автентифікує власника ресурсу та видає токен із заданим контекстом доступу.

Для API-систем це має вирішальне значення, коли потрібне делегування прав. Якщо API-ключ зазвичай є довгоживучим і жорстко прив’язаним до конкретного інтегратора, то OAuth дозволяє обмежувати доступ у часі, обсязі та контексті, не розкриваючи користувацьких облікових даних клієнтському застосунку. Авторизаційний код у класичному потоці надає важливі переваги безпеки, оскільки кінцевий користувач автентифікується на authorization server, а токен не проходить через user-agent у відкритому вигляді як первинний артефакт авторизації. Саме тому OAuth став де-факто стандартом для user-centric API та інтеграцій із делегованими повноваженнями.

Подальший розвиток специфікацій OAuth ще сильніше посилив цей підхід. RFC 9700, який кодифікує актуальні кращі практики безпеки для OAuth 2.0, прямо зазначає, що implicit grant не слід використовувати, а resource owner password credentials grant взагалі не повинен застосовуватися, оскільки він небезпечно розкриває облікові дані власника ресурсу клієнту. Натомість рекомендується використовувати authorization code flow, а також PKCE як захист від перехоплення або ін’єкції коду авторизації. Це важливо і для проєктування системи API-ключів: там, де бізнес-вимоги виходять за межі простої ідентифікації застосунку, система має передбачати інтеграцію з токенною моделлю, а не намагатися розширити API-ключ до ролі, для якої він концептуально не призначений [19].

Ще одним широко вживаним елементом захисту API є JWT. Його сила полягає у компактності, самодостатності та можливості переносити

структуровані claims між компонентами розподіленої системи. Але ця сила перетворюється на ризик, якщо розробник сприймає JWT як “розумний API-ключ” і не будує повноцінну логіку валідації. OWASP наголошує, що перевіряти треба не лише криптографічну цілісність токена, а й змістовні claims. Стандарт RFC 7519 визначає часові обмеження через exp та nbf, цільову аудиторію через aud, видавця через iss, а ідентифікатор jti може допомагати в протидії повторному використанню. У проєктованій системі це означає, що JWT доцільно розглядати не як аналог ключа, а як окремий клас облікових даних із власними правилами зберігання, перевірки та відкликання.

Поряд із моделями автентифікації й авторизації критичне значення мають механізми обмеження доступу на рівні інтенсивності й контексту використання API. OWASP рекомендує для API-ключів вимагати ключ у кожному запиті, повертати 429 Too Many Requests, якщо звернення надходять надто часто, і відкликати ключ при порушенні правил використання. AWS API Gateway розвиває цю ідею через usage plans, де задаються throttling та quota limits, але водночас прямо попереджає, що такі ліміти мають характер best effort і не є жорсткою гарантією блокування чи контролю витрат. Це важливий практичний висновок: rate limiting і quota management необхідні, але не повинні сприйматися як єдиний захист від зловживань.

Нарешті, жодна модель доступу не може вважатися зрілою без аудиту та моніторингу. NIST у SP 800-92 підкреслює потребу в цілісній інфраструктурі log management та в підтримці ефективних процесів журналювання в масштабах організації. OWASP називає журналювання одним із найважливіших detective controls, радить використовувати уніфіковані формати логів, синхронізувати час на системах і централізувати їх у SIEM або log server. У Google API Keys API дії життєвого циклу ключа (створення, оновлення, видалення, читання key string) прямо відображаються в audit logs як Admin Activity або Data Access події. Для системи керування API-ключами це означає, що журналювання має фіксувати не лише факти виклику API, а й усі операції

над самими ключами, зміни політик, спроби читання секретних значень та ознаки аномального використання [20].

1.3 Огляд сучасних систем керування API-ключами та формування вимог до запропонованої системи

Сучасні системи керування API-ключами вже давно вийшли за межі простого сховища. У реальних продуктах вони поєднують генерацію ключів, прив’язування до політик, рольове адміністрування, обмеження сфери застосування, аудит життєвого циклу та механізми інтеграції з токенними схемами. Показовими прикладами є Google Cloud API Keys API, AWS API Gateway та Apigee. Хоч ці рішення різняться за цільовою аудиторією та глибиною можливостей, усі вони демонструють спільну тенденцію: API-ключ розглядається як частина ширшої системи керування доступом, а не як самодостатній елемент безпеки [21].

Google Cloud API Keys API репрезентує підхід, у якому ключ керується як окремий ресурс платформи. Серед сильних сторін цього рішення варто виділити програмне створення та адміністрування ключів, підтримку API restrictions і application restrictions, розмежування прав через IAM, а також наявність деталізованих audit logs для операцій створення, оновлення, видалення та читання рядка ключа. В документації також є вимога, що не слід створювати необмежені ключі, а невикористані ключі потрібно видаляти та періодично проводити ротацію. Слабкою стороною цього підходу є те, що стандартний ключ не автентифікує principal.[22].

AWS API Gateway демонструє дуже практичну, product-oriented модель. Тут ключі безпосередньо пов’язуються з usage plans, що задають квоти, throttling і доступ до окремих stage та method. Існує можливість вимагати ключ на рівні конкретного методу, імпортувати ключі, генерувати їх централізовано та використовувати заголовок X-API-KEY як джерело передавання. Сильними сторонами такого рішення є зручний контроль споживання API, хороша

інтеграція з gateway-рівнем і придатність для комерційного оприлюднення API. Проте сама AWS прямо попереджає, що usage plan quotas і throttling не є жорсткими обмеженнями, а API keys не слід використовувати як механізм автентифікації або авторизації. Це означає, що рішення добре підходить для керування споживанням API, але не може слугувати єдиним бар'єром доступу до чутливих операцій [23].

Apigee, своєю чергою, представляє зрілішу API management-парадигму, де ключ тісно пов'язується з developer app, API product і політикою перевірки. Перевага цього підходу полягає в тому, що доступ описується не лише логікою "ключ існує або не існує", а через зв'язок ключа з продуктом API, переліком дозволених проксі, маршрутів і станом самого застосунку чи розробника. Крім того, Apigee прямо допускає комбінування API key verification з OAuth token verification, що добре відповідає сучасним гібридним сценаріям. Водночас документація Apigee однозначно застерігає: безпека API-ключів обмежена, оскільки ключі легко витягуються з коду застосунку, а тому їх краще трактувати як унікальні ідентифікатори застосунків, а не як повноцінні security tokens [24].

Порівняння цих платформ дозволяє сформулювати важливий узагальнюючий висновок. Усі сучасні рішення, незалежно від рівня складності, намагаються вирішити одні й ті самі завдання: централізовано створювати й відкликати ключі, зв'язувати їх із політиками доступу, обмежувати джерела та сценарії використання, контролювати обсяг споживання API, а також забезпечувати журнали подій для подальшого аудиту. Різниця між продуктами полягає переважно у ступені формалізації політик, рівні інтеграції з інфраструктурою та підтримці складних сценаріїв авторизації. Саме це і має бути відправною точкою для формування вимог до власної системи [25].

Аналіз розглянутих рішень і нормативних джерел дозволяє сформулювати функціональні вимоги до розроблюваної системи. По-перше, система повинна забезпечувати повний життєвий цикл API-ключа: створення, призначення власнику, активацію, деактивацію, відкликання, видалення та

ротацію. По-друге, кожен ключ повинен мати чіткий контекст використання: прив’язку до конкретного клієнта або застосунку, переліку дозволених API чи маршрутів, середовища виконання та набору допустимих технічних обмежень, зокрема джерела виклику. По-третє, потрібне рольове адміністрування самої системи, де права на перегляд, створення, зміну та видалення ключів відокремлені між різними категоріями користувачів. По-четверте, система повинна підтримувати перевірку дозволів на кожному запиті, а також мати можливість інтегруватися з сильнішими механізмами доступу – зокрема OAuth 2.0 та токенами для сценаріїв, де API-ключа недостатньо. По-п’яте, необхідною є підсистема квотування, rate limiting та реакції на порушення політик використання, включно з автоматичним блокуванням або переведенням ключа в обмежений стан [26].

Не менш важливими є нефункціональні вимоги. Передусім система повинна забезпечувати конфіденційність ключів під час передавання і зберігання, не допускати витоку ключів через URL, вихідний код або незахищені канали, а також підтримувати безпечну ротацію та мінімізацію “поверхні атаки” за рахунок видалення невикористаних ключів. Другою вимогою є надійність і послідовність контролю доступу: політики повинні застосовуватися за принципом deny by default, а перевірка прав не може бути факультативною або прив’язаною лише до частини маршрутів. Третьою вимогою є ведення аудиту: усі адміністративні дії, спроби читання секретних значень, зміни прав, а також аномалії використання повинні потрапляти до централізованих логів у придатному для аналізу форматі. Четвертою вимогою є масштабованість. Система має коректно працювати за зростання кількості ключів, клієнтів і транзакцій без суттєвої деградації затримок у точці перевірки. П’ятою вимогою є можливість розширення – можливість доповнити систему новими типами обмежень, новими моделями політик або інтеграціями зі сторонніми шлюзами, SIEM та службами ідентифікації без повного перероблення архітектури [27].

З практичної точки зору для власної системи доцільно одразу відмовитися від хибного припущення, ніби API-ключ сам по собі розв’язує проблему захисту API. Натомість система має проектуватися як керований сервіс політик доступу, у якому ключ є лише носієм зв’язку між клієнтською інтеграцією та набором правил. Центральними сутностями такої архітектури повинні бути не тільки сам ключ, а й його власник, область застосування, стан, політики обмеження, історія змін, квоти, журнали подій та інтеграції з сильнішими механізмами автентифікації й авторизації. Саме такий підхід найбільшою мірою відповідає сучасним практикам, відображеним у нормативних документах і промислових платформах [28].

1.4 Висновок до першого розділу

У першому розділі було проведено огляд теоретичних засад використання API-ключів як одного з механізмів захисту програмних інтерфейсів. Встановлено, що API-ключі займають важливе, проте чітко обмежене місце серед засобів контролю доступу до API. Їх основними перевагами є простота використання, швидка інтеграція в програмні системи, можливість ідентифікації клієнтського застосунку, а також застосування для квотування, білінгу та базового обмеження доступу. Водночас з’ясовано, що у строгому безпековому розумінні типовий API-ключ не забезпечує повноцінної автентифікації користувача або сервісу та не може самостійно реалізувати гнучку модель авторизації для складних або чутливих сценаріїв доступу.

Також у першому розділі було визначено, що ефективне використання API-ключів потребує поєднання з іншими механізмами захисту. Зокрема, важливими є використання TLS-з’єднання, обмеження сфери дії ключа, контроль кількості запитів, журналювання дій, аудит доступу та, за потреби, інтеграція з токенними механізмами, такими як OAuth 2.0 або JWT. Це дає змогу розглядати API-ключ не як самостійний універсальний засіб безпеки, а як один із компонентів комплексної системи керування доступом.

Отже, у результаті аналізу було зроблено висновок, що для розроблення власної системи керування API-ключами недостатньо реалізувати лише їх генерацію та зберігання. Необхідно спроєктувати цілісну підсистему, яка поєднує рольове адміністрування, політики обмеження ключів, контроль запитів, централізований аудит, моніторинг та можливість подальшої інтеграції з надійнішими механізмами автентифікації й авторизації. Саме ці положення стали теоретичною та методологічною основою для подальшого проєктування архітектури, моделі даних і програмної реалізації системи керування API-ключами та доступом до сервісів.

РОЗДІЛ 2. РЕАЛІЗАЦІЯ СИСТЕМИ УПРАВЛІННЯ API-КЛЮЧАМИ ТА ДОСТУПОМ ДО СЕРВІСІВ

2.1 Архітектура системи

Архітектура розробленої системи побудована за модульним принципом і передбачає розділення логіки на окремі функціональні частини: автентифікацію користувачів, керування користувачами, керування сервісами, створення та обслуговування API-ключів, перевірку доступу до сервісів, обмеження кількості запитів і ведення журналу аудиту. Такий підхід дозволяє зробити систему зрозумілою, масштабованою та зручною для подальшого розширення [29]. На рисунку 2.1 показано, як взаємодія з системою може відбуватися двома основними способами.

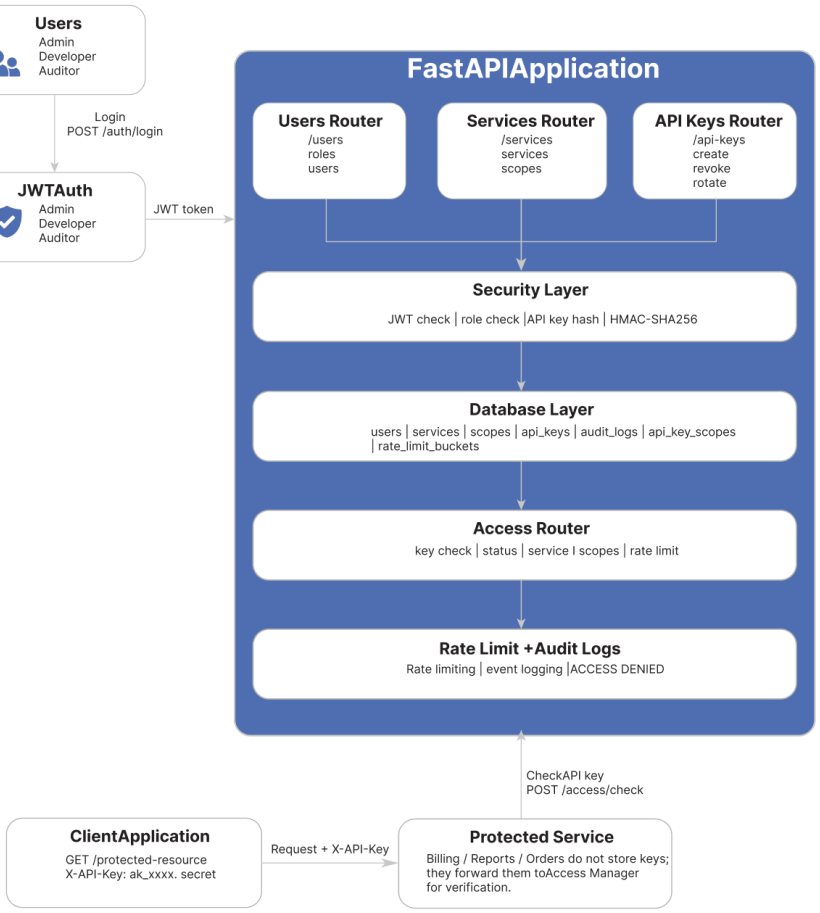

Рисунок 2.1 – Загальна архітектура системи управління API-ключами та доступом до сервісів

Перший спосіб – це робота користувачів системи через JWT-автентифікацію. До таких користувачів належать адміністратор, розробник та аудитор. Вони виконують вхід у систему через endpoint POST /auth/login, після чого отримують JWT-токен. Надалі цей токен використовується для доступу до захищених адміністративних функцій, наприклад створення сервісів, перегляду користувачів, створення API-ключів або перегляду журналу аудиту [30].

Другий спосіб взаємодії – це доступ клієнтського застосунку або окремого захищеного сервісу через API-ключ. У цьому випадку клієнтський застосунок передає API-ключ у заголовку X-API-Key, а захищений сервіс не перевіряє ключ самостійно, а передає його до Access Manager через endpoint POST /access/check. Такий підхід дозволяє винести логіку перевірки ключів в окрему централізовану систему.

Центральним компонентом архітектури є FastAPI Application. У межах цього застосунку реалізовано кілька напрямків, кожен із яких відповідає за окрему частину функціональності. Users Router використовується для роботи з користувачами, їхніми ролями та обліковими записами. Services Router відповідає за створення та перегляд сервісів, а також за налаштування scopes, які визначають конкретні дозволи доступу. API Keys Router реалізує функції створення, перегляду, відкликання та ротації API-ключів.

Окремим важливим компонентом є Access Router, який відповідає за перевірку доступу до сервісів. Саме через нього виконується перевірка API-ключа, його статусу, прив’язки до сервісу, наявності потрібних scopes та обмеження кількості запитів. Таким чином, Access Router є центральним елементом авторизації для зовнішніх клієнтів і захищених сервісів.

Між роутерами та базою даних розміщено Security Layer. Цей рівень відповідає за виконання основних перевірок безпеки. До нього належить перевірка JWT-токенів, перевірка ролей користувачів, хешування API-ключів, перевірка HMAC-SHA256 та контроль прав доступу. Завдяки цьому бізнес-логіка системи не працює напряму з відкритими паролями або відкритими API-

ключами. Паролі користувачів і API-ключі зберігаються в захищеному вигляді, що зменшує ризики у випадку компрометації бази даних [31].

Database Layer є рівнем збереження даних. У ньому містяться основні таблиці системи: users, services, scopes, api_keys, audit_logs, api_key_scopes і rate_limit_buckets. Таблиця users зберігає користувачів системи та їхні ролі. Таблиця services описує сервіси, до яких може надаватися доступ. Таблиця scopes визначає конкретні дозволи для кожного сервісу. Таблиця api_keys містить інформацію про створені API-ключі, їх статус, власника, сервіс, ліміт запитів і час використання. Проміжна таблиця api_key_scopes реалізує зв’язок між API-ключами та дозволами. Таблиця rate_limit_buckets використовується для підрахунку кількості запитів у межах певного часового вікна, а audit_logs зберігає історію важливих подій [32].

У нижній частині схеми на рисунку 2.1 показано взаємодію клієнтського застосунку із захищеним сервісом. Клієнт надсилає запит до protected service разом з API-ключем. Захищений сервіс, наприклад Billing, Reports або Orders, не зберігає ключі локально. Натомість він пересилає ключ до Access Manager для перевірки. Якщо ключ дійсний, активний, належить до потрібного сервісу і має необхідні scopes, Access Manager повертає позитивний результат, після чого protected service виконує запит клієнта.

Такий підхід має кілька переваг. По-перше, перевірка API-ключів централізується в одному місці, тому окремим сервісам не потрібно дублювати логіку авторизації. По-друге, усі дії з доступом фіксуються в журналі аудиту, що дозволяє відстежувати як успішні запити, так і відмови в доступі. По-третє, кожен API-ключ має чітко обмежену область дії: він прив’язаний до конкретного сервісу та має тільки ті scopes, які були надані під час створення.

У системі також передбачено обмеження кількості запитів. Якщо API-ключ перевищує встановлений ліміт за хвилину, запит блокується, а клієнт отримує помилку 429 Too Many Requests. Це дозволяє захистити сервіси від надмірного навантаження та зловживання ключами [33].

Важливою частиною архітектури є журналювання подій. Усі важливі дії, зокрема вхід користувача, створення сервісу, створення API-ключа, використання ключа, відкликання, ротація або відмова в доступі, записуються до audit_logs. Це підвищує прозорість роботи системи та дозволяє адміністратору аналізувати підозрілу активність.

2.2 Проєктування бази даних

Для реалізації системи управління API-ключами та доступом до сервісів було спроєктовано реляційну базу даних. Основним завданням бази даних є збереження інформації про користувачів системи, доступні сервіси, дозволи доступу, API-ключі, обмеження кількості запитів та журнал подій. Така структура дозволяє не просто зберігати ключі, а повноцінно контролювати, який користувач, до якого сервісу і з якими правами має доступ.

У проєкті для роботи з базою даних використано ORM-підхід на основі SQLAlchemy. Це дозволяє описувати таблиці бази даних у вигляді Python-класів, а зв’язки між таблицями – через відповідні поля та відношення. Підключення до бази даних реалізовано через змінну DATABASE_URL, що дає можливість використовувати SQLite під час локальної розробки та за потреби перейти на PostgreSQL або іншу сумісну базу даних.

На рисунку 2.2 показано, як у системі створюється базовий клас Base, від якого наслідуються всі моделі бази даних. Також створюється об’єкт engine, який відповідає за підключення до бази даних, та SessionLocal, що використовується для створення сесій роботи з БД. Завдяки цьому всі запити до бази виконуються централізовано через SQLAlchemy-сесію.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase, sessionmaker

from app.core.config import settings


class Base(DeclarativeBase):
    pass


connect_args = {}
if settings.DATABASE_URL.startswith("sqlite"):
    connect_args = {"check_same_thread": False}

engine = create_engine(settings.DATABASE_URL, connect_args=connect_args)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

Рисунок 2.2 – Налаштування підключення до бази даних

Однією з основних сутностей системи є користувач. Таблиця users зберігає дані про облікові записи, які можуть створювати API-ключі, керувати сервісами або переглядати журнал аудиту залежно від своєї ролі. У системі передбачено декілька ролей: адміністратор, розробник та аудитор. На рисунку 2.3 наведено модель користувача.

```
class User(Base):
    __tablename__ = "users"

    id: Mapped[str] = mapped_column(String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    email: Mapped[str] = mapped_column(String(255), unique=True, index=True, nullable=False)
    full_name: Mapped[str] = mapped_column(String(255), nullable=False)
    hashed_password: Mapped[str] = mapped_column(String(512), nullable=False)
    role: Mapped[UserRole] = mapped_column(
        Enum(UserRole, values_callable=enum_values),
        default=UserRole.DEVELOPER,
        nullable=False,
    )
    is_active: Mapped[bool] = mapped_column(Boolean, default=True, nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)
    updated_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)

    api_keys = relationship( argument: "ApiKey", back_populates="owner")
```

Рисунок 2.3 – Модель таблиці користувачів users

Поле id є унікальним ідентифікатором запису, email використовується для входу в систему, а hashed_password зберігає не сам пароль, а його хешоване значення. Це важливо з погляду безпеки, оскільки справжній пароль користувача не зберігається у відкритому вигляді. Поле role визначає рівень доступу користувача до функцій системи. Наприклад, адміністратор може створювати сервіси та користувачів, розробник може створювати API-ключі для роботи із сервісами, а аудитор може переглядати журнали дій.

Наступною важливою частиною бази даних є таблиці services та scopes. Таблиця services описує окремі сервіси, до яких може надаватися доступ через API-ключі. Наприклад, у системі можуть бути сервіси billing, reports, users-api тощо. Таблиця scopes відповідає за конкретні права доступу всередині сервісу, наприклад read:billing, write:billing, read:reports або export:reports. Як показано на рисунку 2.4, кожен сервіс може мати багато дозволів, тобто scopes.

```python
class Service(Base):
    __tablename__ = "services"

    id: Mapped[str] = mapped_column(String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    slug: Mapped[str] = mapped_column(String(80), unique=True, index=True, nullable=False)
    name: Mapped[str] = mapped_column(String(160), nullable=False)
    description: Mapped[str | None] = mapped_column(Text, nullable=True)
    is_active: Mapped[bool] = mapped_column(Boolean, default=True, nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)

    scopes = relationship( argument: "Scope", back_populates="service", cascade="all, delete-orphan")
    api_keys = relationship( argument: "ApiKey", back_populates="service")


class Scope(Base):
    __tablename__ = "scopes"
    __table_args__ = (UniqueConstraint( *columns: "service_id", "code", name="uq_scope_service_code"),)

    id: Mapped[str] = mapped_column(String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    service_id: Mapped[str] = mapped_column(ForeignKey( column: "services.id", ondelete="CASCADE"), nullable=False)
    code: Mapped[str] = mapped_column(String(120), nullable=False)
    description: Mapped[str | None] = mapped_column(Text, nullable=True)
    is_active: Mapped[bool] = mapped_column(Boolean, default=True, nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)

    service = relationship( argument: "Service", back_populates="scopes")
```

Рисунок 2.4 – Моделі таблиць services та scopes

Зв’язок між сервісом і дозволами реалізований за допомогою зовнішнього ключа service_id у таблиці scopes. Це означає, що кожен дозвіл належить до конкретного сервісу. Наприклад, scope read:billing належить до сервісу billing, а scope export:reports – до сервісу reports.

Для уникнення дублювання прав доступу використано обмеження унікальності UniqueConstraint("service_id", "code"). Завдяки цьому в межах одного сервісу не можна створити два однакові scope-коди. Наприклад, для сервісу billing не можна двічі створити дозвіл read:billing.

Центральною таблицею системи є api_keys, оскільки саме вона зберігає інформацію про API-ключі, їх власників, прив’язку до сервісів, статус, час створення, термін дії та кількість використань. Важливо, що у базі даних не зберігається відкритий API-ключ. Замість цього зберігається тільки його хеш у полі hashed_key та короткий префікс у полі key_prefix (див. рисунок 2.5).

```
api_key_scopes = Table(
    name: "api_key_scopes",
    Base.metadata,
    *args: Column( __name_pos: "api_key_id", ForeignKey( column: "api_keys.id", ondelete="CASCADE"), primary_key=True),
    Column( __name_pos: "scope_id", ForeignKey( column: "scopes.id", ondelete="CASCADE"), primary_key=True),
)

class ApiKey(Base):
    __tablename__ = "api_keys"

    id: Mapped[str] = mapped_column(String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    owner_id: Mapped[str] = mapped_column(ForeignKey( column: "users.id", ondelete="CASCADE"), index=True, nullable=False)
    service_id: Mapped[str] = mapped_column(ForeignKey( column: "services.id", ondelete="CASCADE"), index=True, nullable=False)
    name: Mapped[str] = mapped_column(String(160), nullable=False)
    key_prefix: Mapped[str] = mapped_column(String(32), unique=True, index=True, nullable=False)
    hashed_key: Mapped[str] = mapped_column(String(128), unique=True, index=True, nullable=False)
    status: Mapped[ApiKeyStatus] = mapped_column(Enum(ApiKeyStatus, values_callable=enum_values), nullable=False)
    rate_limit_per_minute: Mapped[int] = mapped_column(Integer, default=60, nullable=False)
    usage_count: Mapped[int] = mapped_column(Integer, default=0, nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)
    expires_at: Mapped[datetime | None] = mapped_column(DateTime(timezone=True), nullable=True)
    revoked_at: Mapped[datetime | None] = mapped_column(DateTime(timezone=True), nullable=True)
    last_used_at: Mapped[datetime | None] = mapped_column(DateTime(timezone=True), nullable=True)

    owner = relationship( argument: "User", back_populates="api_keys")
    service = relationship( argument: "Service", back_populates="api_keys")
    scopes = relationship( argument: "Scope", secondary=api_key_scopes)
```

Рисунок 2.5 – Модель таблиці api_keys та проміжної таблиці api_key_scopes

На рисунку 2.5 показано, що кожен API-ключ має власника, тобто користувача, який його створив або для якого він був створений. Це

реалізовано через поле owner_id, яке посилається на таблицю users. Також кожен API-ключ прив'язаний до певного сервісу через поле service_id. Такий підхід не дозволяє використовувати ключ одного сервісу для доступу до іншого сервісу.

Окремо реалізовано зв'язок між API-ключами та scopes. Оскільки один API-ключ може мати декілька дозволів, а один дозвіл може бути прив'язаний до багатьох ключів, між таблицями api_keys і scopes використовується проміжна таблиця api_key_scopes. Вона реалізує зв'язок типу «багато до багатьох». Наприклад, один ключ може мати одночасно дозволи read:billing і write:billing, а інший ключ – тільки read:billing.

Поле status дозволяє контролювати стан API-ключа. Ключ може бути активним, відкликаним або простроченим. Це дає можливість не видаляти ключ із бази даних, а змінювати його стан, зберігаючи історію його існування. Поля expires_at, revoked_at та last_used_at використовуються для контролю терміну дії, часу відкликання та останнього використання ключа.

Для обмеження кількості запитів у системі передбачено таблицю rate_limit_buckets. Вона зберігає кількість запитів для конкретного API-ключа в межах певного часового вікна. Це потрібно для захисту сервісів від надмірного навантаження або зловживання API-ключем (див. рисунок 2.6).

```python
class RateLimitBucket(Base):
    __tablename__ = "rate_limit_buckets"
    __table_args__ = (UniqueConstraint( *columns: "api_key_id", "window_start", name="uq_rate_limit_key_window"),)

    id: Mapped[str] = mapped_column(String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    api_key_id: Mapped[str] = mapped_column(ForeignKey( column: "api_keys.id", ondelete="CASCADE"), index=True, nullable=False)
    window_start: Mapped[datetime] = mapped_column(DateTime(timezone=True), index=True, nullable=False)
    requests_count: Mapped[int] = mapped_column(Integer, default=0, nullable=False)
```

Рисунок 2.6 – Модель таблиці rate_limit_buckets

Як видно з рисунка 2.6, таблиця rate_limit_buckets пов'язана з таблицею api_keys через поле api_key_id. Поле window_start визначає початок часового проміжку, для якого рахується кількість запитів, а requests_count зберігає їхню

кількість. Обмеження унікальності для api_key_id та window_start гарантує, що для одного API-ключа в межах одного часового вікна буде тільки один запис.

Також у базі даних передбачено таблицю audit_logs, яка використовується для журналювання важливих дій у системі. До таких дій належать створення користувача, вхід у систему, створення сервісу, створення API-ключа, використання ключа, відкликання ключа, ротація ключа та відмова в доступі. На рисунку 2.7 наведено структуру таблиці журналу аудиту.

```python
class AuditLog(Base):
    __tablename__ = "audit_logs"

    id: Mapped[str] = mapped_column(String(36), primary_key=True, default=lambda: str(uuid.uuid4()))
    actor_user_id: Mapped[str | None] = mapped_column(ForeignKey( column: "users.id", ondelete="SET NULL"), nullable=True)
    action: Mapped[AuditAction] = mapped_column(Enum(AuditAction, values_callable=enum_values), index=True, nullable=False)
    target_type: Mapped[str | None] = mapped_column(String(80), nullable=True)
    target_id: Mapped[str | None] = mapped_column(String(80), nullable=True)
    ip_address: Mapped[str | None] = mapped_column(String(80), nullable=True)
    details: Mapped[dict | None] = mapped_column(JSON, nullable=True)
    created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), index=True, nullable=False)
```

Рисунок 2.7 – Модель таблиці audit_logs

Поле actor_user_id вказує на користувача, який виконав дію. Поле action описує тип події, наприклад створення API-ключа або відмову в доступі. Поля target_type і target_id дозволяють визначити, над яким об’єктом виконувалася дія. Наприклад, якщо було відкликано API-ключ, то в журналі може зберігатися тип об’єкта api_key та його ідентифікатор. Поле details зберігає додаткові дані у форматі JSON, що робить журнал більш гнучким.

Загальна структура бази даних побудована таким чином, щоб забезпечити розмежування доступу між різними сервісами. Користувач створює API-ключ, API-ключ прив’язується до конкретного сервісу, а через scopes визначається, які саме дії дозволені. Якщо ключ не належить до потрібного сервісу або не має необхідного scope, система відмовляє у доступі. Завдяки цьому база даних підтримує не тільки збереження інформації, а й основну логіку авторизації в системі.

2.3 Реалізація сервісів і Scope-дозволів

У розробленій системі механізм автентифікації та авторизації є однією з ключових складових, оскільки саме він визначає, хто може працювати із системою та які дії дозволено виконувати. Автентифікація відповідає за перевірку особи користувача, тобто підтвердження його логіна і пароля. Авторизація, у свою чергу, визначає рівень доступу користувача або API-ключа до певних функцій системи.

У проєкті використано два основні механізми доступу. Перший механізм призначений для користувачів системи та базується на JWT-токенах. Він використовується для входу адміністратора, розробника або аудитора в систему. Другий механізм призначений для зовнішніх сервісів і працює через API-ключі, які передаються в HTTP-заголовку X-API-Key або у форматі Authorization: ApiKey <key>.

Для захисту паролів користувачів у системі не зберігається пароль у відкритому вигляді. Перед записом у базу даних пароль хешується за допомогою алгоритму PBKDF2-SHA256. Такий підхід дозволяє зменшити ризик компрометації облікових записів у випадку несанкціонованого доступу до бази даних.

Як показано на рисунку 2.8, для кожного пароля створюється випадкова сіль salt, після чого формується хеш. При повторному вході користувача пароль не розшифровується, а знову хешується із тією самою сіллю, після чого результат порівнюється зі збереженим значенням. Для порівняння використовується функція hmac.compare_digest, що дозволяє безпечніше порівнювати значення та зменшує ризик атак, пов’язаних із вимірюванням часу виконання.

```
PASSWORD_ALGORITHM = "pbkdf2_sha256"
PASSWORD_ITERATIONS = 260_000

def hash_password(password: str) -> str:
    salt = secrets.token_bytes(16)
    digest = hashlib.pbkdf2_hmac( hash_name: "sha256", password.encode("utf-8"), salt, PASSWORD_ITERATIONS)
    return "$".join(
        [
            PASSWORD_ALGORITHM,
            str(PASSWORD_ITERATIONS),
            base64.b64encode(salt).decode("ascii"),
            base64.b64encode(digest).decode("ascii"),
        ]
    )


def verify_password(password: str, password_hash: str) -> bool:
    try:
        algorithm, iterations_raw, salt_raw, digest_raw = password_hash.split( sep: "$",  maxsplit: 3)
        if algorithm != PASSWORD_ALGORITHM:
            return False
        iterations = int(iterations_raw)
        salt = base64.b64decode(salt_raw.encode("ascii"))
        expected = base64.b64decode(digest_raw.encode("ascii"))
    except (ValueError, TypeError):
        return False

    actual = hashlib.pbkdf2_hmac( hash_name: "sha256", password.encode("utf-8"), salt, iterations)
    return hmac.compare_digest(actual, expected)
```

Рисунок 2.8 – Реалізація хешування та перевірки пароля користувача

Після успішної перевірки логіна і пароля система створює JWT-токен. У токен записується ідентифікатор користувача, час створення, час завершення дії токена та додаткові дані, наприклад роль користувача. Завдяки цьому клієнт може виконувати подальші запити до захищених endpoint-ів без повторного введення пароля.

На рисунку 2.9 наведено функцію створення access token.

```python
def create_access_token(subject: str, claims: dict[str, Any] | None = None) -> str:
    now = datetime.now(timezone.utc)
    payload = {
        "sub": subject,
        "iat": int(now.timestamp()),
        "exp": int((now + timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)).timestamp()),
    }
    if claims:
        payload.update(claims)

    header = {"alg": "HS256", "typ": "JWT"}
    signing_input = ".".join(
        [
            _b64url_encode(json.dumps(header, separators=(",", ":")).encode("utf-8")),
            _b64url_encode(json.dumps(payload, separators=(",", ":")).encode("utf-8")),
        ]
    )
    signature = hmac.new(settings.JWT_SECRET.encode("utf-8"), signing_input.encode("ascii"), hashlib.sha256)
    return f"{signing_input}.{_b64url_encode(signature.digest())}"
```

Рисунок 2.9 – Створення JWT-токена для автентифікованого користувача

Поле sub містить ідентифікатор користувача, iat – час створення токена, а exp – час завершення його дії. Токен підписується за допомогою секретного ключа JWT_SECRET, який зберігається у конфігурації системи. Це дозволяє серверу надалі перевіряти, чи був токен створений саме цією системою і чи не був він змінений сторонньою особою.

Для перевірки токена реалізовано окрему функцію декодування. Вона перевіряє структуру токена, його підпис і термін дії. Якщо токен некоректний, має неправильний підпис або вже прострочений, система повертає помилку автентифікації.

Як видно з рисунка 2.10, перевірка токена складається з кількох етапів. Спочатку токен розділяється на заголовок, корисне навантаження та підпис. Далі система повторно формує очікуваний підпис і порівнює його з підписом, який прийшов у запиті. Якщо підписи не збігаються, доступ забороняється. Також перевіряється поле exp, яке відповідає за час завершення дії токена. Це не дозволяє використовувати застарілі токени після завершення їхнього терміну дії.

```python
def decode_access_token(token: str) -> dict[str, Any]:
    try:
        header_raw, payload_raw, signature_raw = token.split( sep: ".",  maxsplit: 2)
    except ValueError as exc:
        raise UnauthorizedError("Invalid access token.") from exc

    signing_input = f"{header_raw}.{payload_raw}"
    expected_signature = hmac.new(
        settings.JWT_SECRET.encode("utf-8"),
        signing_input.encode("ascii"),
        hashlib.sha256,
    ).digest()
    supplied_signature = _b64url_decode(signature_raw)
    if not hmac.compare_digest(expected_signature, supplied_signature):
        raise UnauthorizedError("Invalid access token signature.")

    try:
        payload = json.loads(_b64url_decode(payload_raw))
    except json.JSONDecodeError as exc:
        raise UnauthorizedError("Invalid access token payload.") from exc

    exp = payload.get("exp")
    if not isinstance(exp, int) or exp < int(datetime.now(timezone.utc).timestamp()):
        raise UnauthorizedError("Access token expired.")

    return payload
```

Рисунок 2.10 – Перевірка JWT-токена та його терміну дії

Сам процес входу користувача реалізовано через endpoint /auth/login. Користувач надсилає email і пароль, після чого система перевіряє облікові дані. Якщо вони правильні, користувач отримує JWT-токен, який надалі використовується для доступу до захищених ресурсів.

На рисунку 2.11 показано, що після успішного входу користувача система не тільки повертає токен, а й записує подію входу до журналу аудиту. Це важливо для контролю дій у системі, оскільки адміністратор може переглянути, коли користувач входив у систему. У разі неправильного email або пароля повертається помилка 401 Unauthorized.

```python
@router.post( path: "/login", response_model=TokenResponse)
def login(payload: LoginRequest, request: Request, db: Session = Depends(get_db)):
    user = auth_service.authenticate_user(db, str(payload.email), payload.password)
    if not user:
        raise UnauthorizedError("Invalid email or password.")

    token = create_access_token(subject=user.id, claims={"role": user.role.value})
    record_audit(
        db,
        action=AuditAction.USER_LOGIN,
        actor_user_id=user.id,
        target_type="user",
        target_id=user.id,
        ip_address=request.client.host if request.client else None,
    )
    db.commit()
    return TokenResponse(access_token=token, expires_in_minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)
```

Рисунок 2.11 – Endpoint авторизації користувача /auth/login

Для отримання поточного користувача з HTTP-запиту використовується dependency-функція get_current_user. Вона читає заголовок Authorization, перевіряє, що використовується схема Bearer, декодує токен і знаходить користувача в базі даних (див. рисунок 2.12).

```python
def get_current_user(
    authorization: Annotated[str | None, Header(alias="Authorization")] = None,
    db: Session = Depends(get_db),
) -> User:
    if not authorization:
        raise UnauthorizedError()

    scheme, _, token = authorization.partition(" ")
    if scheme.lower() != "bearer" or not token:
        raise UnauthorizedError("Expected Authorization: Bearer <token>.")

    payload = decode_access_token(token)
    subject = payload.get("sub")
    if not isinstance(subject, str):
        raise UnauthorizedError("Invalid token subject.")

    user = get_user_by_id(db, subject)
    if not user.is_active:
        raise UnauthorizedError("User is inactive.")
    return user
```

Рисунок 2.12 – Отримання поточного користувача через JWT-токен

Як показано на рисунку 2.12, якщо заголовок Authorization відсутній або має неправильний формат, система повертає помилку доступу. Якщо токен коректний, із нього береться поле sub, яке містить ідентифікатор користувача. Після цього користувач шукається в базі даних. Додатково перевіряється поле is_active, щоб заблокований або неактивний користувач не міг працювати із системою.

Окрім самої автентифікації, у системі реалізовано авторизацію на основі ролей. Для цього використовується функція require_roles, яка дозволяє обмежувати доступ до певних endpoint-ів тільки для користувачів із потрібними ролями. У системі передбачено ролі admin, developer та auditor. На рисунку 2.13 показано механізм перевірки ролей.

```
def require_roles(*roles: UserRole) -> Callable:
    def dependency(current_user: User = Depends(get_current_user)) -> User:
        if current_user.role not in roles:
            raise ForbiddenError("Insufficient role.")
        return current_user

    return dependency
```

Рисунок 2.13 – Перевірка ролі користувача

Якщо користувач має одну з дозволених ролей, запит продовжує виконуватися. Якщо роль користувача не відповідає вимогам endpoint-а, система повертає помилку 403 Forbidden. Наприклад, створення сервісів доступне тільки адміністратору, а перегляд журналу аудиту може бути дозволений адміністратору або аудитору.

Приклад використання рольової авторизації наведено в endpoint-і створення сервісу. Для виконання цієї операції користувач повинен мати роль адміністратора.

Як видно з рисунка 2.14, endpoint створення сервісу використовує залежність require_roles(UserRole.ADMIN). Це означає, що навіть якщо користувач успішно увійшов у систему, але не є адміністратором, він не зможе

створити новий сервіс. Такий підхід дозволяє розділити права між різними типами користувачів.

```python
@router.post( path: "", response_model=ServiceRead, status_code=status.HTTP_201_CREATED)
def post_service(
    payload: ServiceCreate,
    request: Request,
    db: Session = Depends(get_db),
    current_user: User = Depends(require_roles(UserRole.ADMIN)),
):
    service = create_service(db, payload)
    record_audit(
        db,
        action=AuditAction.SERVICE_CREATED,
        actor_user_id=current_user.id,
        target_type="service",
        target_id=service.id,
        details={"slug": service.slug},
        ip_address=request.client.host if request.client else None,
    )
    db.commit()
    db.refresh(service)
    return service
```

Рисунок 2.14 – Обмеження доступу до створення сервісу за роллю адміністратора

Окрема частина авторизації стосується API-ключів. Вона потрібна для зовнішніх сервісів, які не здійснюють вхід за допомогою email і пароль, але повинні мати доступ до певного API. API-ключ може бути переданий у заголовку X-API-Key або у заголовку Authorization зі схемою ApiKey (див. рисунок 2.15).

```python
def extract_api_key_value(x_api_key: str | None, authorization: str | None) -> str:
    if x_api_key:
        return x_api_key.strip()

    if authorization:
        scheme, _, token = authorization.partition(" ")
        if scheme.lower() == "apikey" and token:
            return token.strip()

    raise UnauthorizedError("Expected X-API-Key header or Authorization: ApiKey <key>.")
```

Рисунок 2.15 – Отримання API-ключа з HTTP-заголовків

На рисунку 2.15 показано, що система підтримує два способи передачі API-ключа. Основним способом є заголовок X-API-Key, однак також передбачено варіант Authorization: ApiKey <key>. Якщо ключ не передано, система повертає помилку 401 Unauthorized. Це означає, що запит не може бути оброблений, поки клієнт не передасть коректні дані для автентифікації.

Для зручної перевірки доступу API-ключа реалізовано dependency-функцію api_key_guard. Вона використовується безпосередньо в захищених endpoint-ах і дозволяє вказати, до якого сервісу та з якими scope-дозволами повинен мати доступ ключ (див. рисунок 2.16).

```
def api_key_guard(service_slug: str, required_scopes: list[str] | None = None) -> Callable:
    def dependency(
        request: Request,
        x_api_key: Annotated[str | None, Header(alias="X-API-Key")] = None,
        authorization: Annotated[str | None, Header(alias="Authorization")] = None,
        db: Session = Depends(get_db),
    ) -> ApiKeyPrincipal:
        plain_key = extract_api_key_value(x_api_key, authorization)
        client_ip = request.client.host if request.client else None
        return verify_api_key_access(
            db=db,
            plain_api_key=plain_key,
            service_slug=service_slug,
            required_scopes=required_scopes or [],
            ip_address=client_ip,
        )

    return dependency
```

Рисунок 2.16 – Dependency-функція для захисту endpoint-а за API-ключем

Як показано на рисунку 2.16, api_key_guard приймає назву сервісу та список обов’язкових дозволів. Після цього функція отримує API-ключ із заголовків і передає його на перевірку. Якщо ключ дійсний, активний, належить до потрібного сервісу і має необхідні scopes, запит допускається до виконання. Якщо хоча б одна умова не виконується, система повертає відповідну помилку доступу.

Основна логіка перевірки API-ключа реалізована у функції verify_api_key_access. Вона перевіряє наявність ключа в базі, його статус, термін дії, відповідність сервісу, наявність потрібних scopes та обмеження кількості запитів. На рисунку 2.17 наведено першу частину перевірки API-ключа.

```
def verify_api_key_access(
    db: Session,
    plain_api_key: str,
    service_slug: str,
    required_scopes: list[str] | None = None,
    ip_address: str | None = None,
) -> ApiKeyPrincipal:
    hashed_key = hash_api_key(plain_api_key)
    api_key = db.scalar(
        select(ApiKey)
        .options( *options: selectinload(ApiKey.scopes), selectinload(ApiKey.service))
        .where(ApiKey.hashed_key == hashed_key)
    )

    if not api_key or not hmac.compare_digest(api_key.hashed_key, hashed_key):
        record_audit(
            db,
            action=AuditAction.ACCESS_DENIED,
            target_type="api_key",
            details={"reason": "invalid_api_key", "service_slug": service_slug},
            ip_address=ip_address,
        )
        db.commit()
        raise UnauthorizedError("Invalid API key.")

    now = datetime.now(timezone.utc)
    if api_key.status != ApiKeyStatus.ACTIVE:
        record_audit(
            db,
            action=AuditAction.ACCESS_DENIED,
            actor_user_id=api_key.owner_id,
            target_type="api_key",
            target_id=api_key.id,
            details={"reason": f"key_{api_key.status}", "service_slug": service_slug},
            ip_address=ip_address,
        )
        db.commit()
        raise UnauthorizedError("API key is not active.")
```

Рисунок 2.17 – Перевірка існування та статусу API-ключа

Спочатку відкритий ключ хешується, після чого система шукає відповідний хеш у базі даних. Це важливо, тому що сам API-ключ у відкритому вигляді не зберігається. Якщо ключ не знайдено або він не є активним, доступ забороняється, а подія записується в журнал аудиту.

Після перевірки існування ключа система перевіряє, чи не завершився термін його дії, чи відповідає він потрібному сервісу та чи має необхідні scope-дозволи (див. рисунок 2.18).

```
def deny(reason: str, error: Exception, missing_scopes: list[str] | None = None):
    record_audit(
        db,
        action=AuditAction.ACCESS_DENIED,
        actor_user_id=api_key.owner_id,
        target_type="api_key",
        target_id=api_key.id,
        details={
            "reason": reason,
            "service_slug": service_slug,
            "missing_scopes": missing_scopes,
        },
        ip_address=ip_address,
    )
    db.commit()
    raise error

if api_key.expires_at and api_key.expires_at <= now:
    api_key.status = ApiKeyStatus.EXPIRED
    deny( reason: "expired", UnauthorizedError("API key expired."))

if not api_key.service or not api_key.service.is_active or api_key.service.slug != service_slug:
    deny( reason: "service_mismatch", ForbiddenError("API key is not allowed for this service."))

granted_scopes = {scope.code for scope in api_key.scopes if scope.is_active}
required = set(required_scopes or [])

if "*" not in granted_scopes and not required.issubset(granted_scopes):
    deny(
        reason: "missing_scopes",
        ForbiddenError("API key is missing required scopes."),
        sorted(required - granted_scopes),
    )
```

Рисунок 2.18 – Перевірка терміну дії, сервісу та scope-дозволів API-ключа

Як видно з рисунка 2.18, система послідовно перевіряє кілька умов. Якщо API-ключ прострочений, його статус змінюється на expired, після чого запит відхиляється. Якщо ключ був створений для іншого сервісу, система повертає помилку 403 Forbidden. Наприклад, ключ для сервісу billing не може бути використаний для доступу до сервісу reports.

Окремо перевіряються scope-дозволи. Якщо endpoint вимагає дозвіл write:billing, а ключ має тільки read:billing, доступ буде заборонено. Такий підхід дозволяє реалізувати гнучке розмежування прав не тільки між сервісами, а й між окремими діями в межах одного сервісу.

У разі успішного проходження всіх перевірок система оновлює інформацію про використання API-ключа, збільшує лічильник запитів, записує подію до журналу аудиту та повертає об’єкт користувацького доступу (див. рисунок 2.19).

```
check_rate_limit(db, api_key, now=now)
api_key.last_used_at = now
api_key.usage_count += 1
record_audit(
    db,
    action=AuditAction.API_KEY_USED,
    actor_user_id=api_key.owner_id,
    target_type="api_key",
    target_id=api_key.id,
    details={"service_slug": service_slug, "required_scopes": sorted(required)},
    ip_address=ip_address,
)
db.commit()

return ApiKeyPrincipal(
    api_key_id=api_key.id,
    owner_id=api_key.owner_id,
    service_id=api_key.service_id,
    service_slug=api_key.service.slug,
    granted_scopes=sorted(granted_scopes),
)
```

Рисунок 2.19 – Успішна авторизація API-ключа та запис події використання

На рисунку 2.19 показано завершальний етап авторизації API-ключа. Перед наданням доступу система перевіряє rate limit, тобто обмеження кількості запитів за певний проміжок часу. Після цього оновлюються поля last_used_at і usage_count, що дозволяє відстежувати активність ключа. Також створюється запис аудиту з дією API_KEY_USED.

Для демонстрації роботи механізму авторизації в системі реалізовано endpoint /access/check. Він дозволяє перевірити, чи має переданий API-ключ доступ до вказаного сервісу та потрібних scopes (див. рисунок 2.20).

```python
@router.post( path: "/access/check", response_model=AccessCheckResponse)
def check_access(
    payload: AccessCheckRequest,
    request: Request,
    x_api_key: Annotated[str | None, Header(alias="X-API-Key")] = None,
    authorization: Annotated[str | None, Header(alias="Authorization")] = None,
    db: Session = Depends(get_db),
):
    plain_key = extract_api_key_value(x_api_key, authorization)
    principal = verify_api_key_access(
        db=db,
        plain_api_key=plain_key,
        service_slug=payload.service_slug,
        required_scopes=payload.required_scopes,
        ip_address=request.client.host if request.client else None,
    )
    return AccessCheckResponse(allowed=True, **principal.__dict__)
```

Рисунок 2.20 – Endpoint перевірки доступу /access/check

Як показано на рисунку 2.20, endpoint /access/check приймає назву сервісу та список потрібних дозволів. Якщо ключ проходить перевірку, система повертає відповідь allowed: true, а також інформацію про API-ключ, власника, сервіс і надані scopes. Якщо перевірка не пройдена, система повертає помилку 401 або 403 залежно від причини відмови.

Також у системі реалізовано приклад захищеного endpoint-а, який доступний тільки для API-ключів сервісу billing зі scope-дозволом read:billing. На рисунку 2.21 наведено приклад реального використання механізму авторизації через API-ключ.

```python
@router.get("/demo/billing-reports")
def demo_billing_reports(
    principal: ApiKeyPrincipal = Depends(api_key_guard( service_slug: "billing",  required_scopes: ["read:billing"])),
):
    return {
        "service": "billing",
        "message": "Protected billing report data.",
        "api_key_id": principal.api_key_id,
        "owner_id": principal.owner_id,
    }
```

Рисунок 2.21 – Приклад захищеного endpoint-а для сервісу billing

Endpoint /demo/billing-reports буде доступний тільки в тому випадку, якщо клієнт передасть валідний API-ключ, створений для сервісу billing, і цей ключ матиме scope read:billing. Якщо ключ не передано, буде повернено помилку 401 Unauthorized. Якщо ключ дійсний, але не має потрібного дозволу або належить до іншого сервісу, система поверне 403 Forbidden.

Таким чином, у системі реалізовано багаторівневий механізм автентифікації та авторизації. Для користувачів використовується JWT-автентифікація з перевіркою доступу на основі ролей, а для зовнішніх сервісів – API-ключі зі scope-дозволами. Такий підхід дозволяє окремо контролювати права адміністраторів, розробників, аудиторів та клієнтських сервісів. Крім того, кожна важлива дія записується до журналу аудиту, що підвищує прозорість роботи системи та дозволяє відстежувати спроби несанкціонованого доступу.

2.4 Реалізація управління API-ключами

У розробленій системі API-ключі використовуються як основний механізм доступу зовнішніх клієнтів або сервісів до захищених ресурсів. На відміну від звичайної автентифікації користувача за допомогою логіна і пароля, API-ключ дозволяє надати доступ не людині напряму, а певному клієнтському застосунку, мікросервісу або інтеграції. Саме тому в системі передбачено створення, перегляд, відкликання та ротацію API-ключів.

Кожен API-ключ у системі має власника, назву, сервіс, набір дозволів, статус, ліміт запитів, час створення та додаткові службові поля. Важливо, що відкритий ключ не зберігається в базі даних. Після створення він показується користувачу лише один раз, а в базу записується тільки його хеш. Це підвищує рівень безпеки, оскільки навіть у разі доступу до бази даних неможливо отримати справжні значення API-ключів.

Для опису структури запитів і відповідей, пов'язаних з API-ключами, у системі використано Pydantic-схеми. Вони визначають, які дані потрібно передати під час створення ключа та які дані повертаються у відповіді (див. рисунок 2.22).

```python
class ApiKeyCreate(BaseModel):
    name: str = Field(min_length=2, max_length=160)
    service_id: str
    scope_ids: list[str] = Field(min_length=1)
    owner_id: str | None = None
    expires_at: datetime | None = None
    rate_limit_per_minute: int = Field(default=60, ge=1, le=100_000)


class ApiKeyCreated(BaseModel):
    api_key: ApiKeyRead
    plain_key: str = Field(description="Shown once. Store it securely; it cannot be recovered.")


class ApiKeyRotateResponse(BaseModel):
    old_key_id: str
    new_api_key: ApiKeyRead
    plain_key: str
```

Рисунок 2.22 – Схеми створення та повернення API-ключа

Як показано на рисунку 2.22, під час створення API-ключа необхідно вказати його назву, сервіс, список дозволів scope_ids, а також за потреби власника, термін дії та ліміт запитів за хвилину. У відповіді система повертає об'єкт створеного ключа та поле plain_key. Саме plain_key є відкритим значенням API-ключа, яке користувач повинен зберегти одразу після створення, тому що надалі воно не може бути відновлене з бази даних.

Генерація API-ключа реалізована окремою функцією. Ключ складається з короткого префікса та секретної частини. Префікс може використовуватися для зручної ідентифікації ключа в системі, а секретна частина є основним значенням, яке передається клієнтом під час запитів. На рисунку 2.23 показано, що система створює API-ключ у форматі ak_<prefix>.<secret>.

```
def generate_api_key() -> tuple[str, str]:
    prefix = f"ak_{secrets.token_hex(4)}"
    secret = secrets.token_urlsafe(32)
    return prefix, f"{prefix}.{secret}"


def hash_api_key(plain_api_key: str) -> str:
    return hmac.new(
        settings.API_KEY_PEPPER.encode("utf-8"),
        plain_api_key.encode("utf-8"),
        hashlib.sha256,
    ).hexdigest()
```

Рисунок 2.23 – Генерація та хешування API-ключа

Для секретної частини використовується випадкове значення, згенероване через модуль secrets. Після створення відкритий ключ хешується за допомогою HMAC-SHA256 із використанням додаткового секретного значення API_KEY_PEPPER. У базі даних зберігається саме результат функції hash_api_key, а не сам ключ.

Основна логіка створення API-ключа розміщена у сервісному шарі. Перед створенням ключа система перевіряє, чи існує власник ключа, чи активний він, чи існує обраний сервіс та чи належать вибрані scopes саме до цього сервісу.

Як видно з рисунка 2.24, перед створенням ключа система не дозволяє прив’язати його до неіснуючого користувача або неактивного сервісу. Також перевіряється, чи всі передані scope_ids належать саме до вказаного сервісу. Це важливо, тому що ключ для сервісу billing не повинен отримати дозвіл, який належить до сервісу reports.

```python
def create_api_key(
    db: Session,
    owner_id: str,
    service_id: str,
    name: str,
    scope_ids: list[str],
    expires_at: datetime | None,
    rate_limit_per_minute: int,
) -> tuple[ApiKey, str]:
    owner = db.get(User, owner_id)
    if not owner or not owner.is_active:
        raise NotFoundError("Owner user not found or inactive.")

    service = db.get(Service, service_id)
    if not service or not service.is_active:
        raise NotFoundError("Service not found or inactive.")
```

Рисунок 2.24 – Перевірка власника, сервісу та дозволів перед створенням API-ключа

Після проходження перевірок система генерує відкритий API-ключ, створює його хеш і зберігає в базі даних новий запис. До запису також прив’язуються вибрані scopes, які визначають права доступу цього ключа (див. рисунок 2.25).

```python
    prefix, plain_key = generate_api_key()
    api_key = ApiKey(
        owner_id=owner_id,
        service_id=service_id,
        name=name.strip(),
        key_prefix=prefix,
        hashed_key=hash_api_key(plain_key),
        expires_at=expires_at,
        rate_limit_per_minute=rate_limit_per_minute,
        scopes=scopes,
    )
    db.add(api_key)
    db.flush()
    db.refresh(api_key)
    return api_key, plain_key
```

Рисунок 2.25 – Створення запису API-ключа в базі даних

На рисунку 2.25 показано створення об’єкта ApiKey. У поле key_prefix записується короткий префікс ключа, а в поле hashed_key – хешоване значення відкритого ключа. Поле rate_limit_per_minute визначає кількість дозволених запитів за хвилину. Поле scopes встановлює список дозволів, які буде мати цей ключ. Після збереження система повертає і сам об’єкт ключа, і відкритий plain_key, який надається користувачу лише один раз.

Для створення API-ключа з боку HTTP API реалізовано endpoint POST /api-keys. Доступ до нього мають користувачі з ролями адміністратора або розробника. При цьому розробник може створювати ключі лише для себе, а адміністратор може створювати ключі також для інших користувачів (див. рисунок 2.26).

```python
@router.post( path: "", response_model=ApiKeyCreated, status_code=status.HTTP_201_CREATED)
def create_api_key(
    payload: ApiKeyCreate,
    request: Request,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user),
):
    if current_user.role not in {UserRole.ADMIN, UserRole.DEVELOPER}:
        raise ForbiddenError("Only admins and developers can create API keys.")

    owner_id = payload.owner_id or current_user.id
    if owner_id != current_user.id and not _is_admin(current_user):
        raise ForbiddenError("Only admins can create keys for other users.")

    api_key, plain_key = api_key_service.create_api_key(
        db=db,
        owner_id=owner_id,
        service_id=payload.service_id,
        name=payload.name,
        scope_ids=payload.scope_ids,
        expires_at=payload.expires_at,
        rate_limit_per_minute=payload.rate_limit_per_minute,
    )
```

Рисунок 2.26 – Endpoint створення API-ключа

Як показано на рисунку 2.26, створення API-ключів обмежене за ролями. Якщо користувач не є адміністратором або розробником, система повертає помилку доступу. Якщо користувач намагається створити ключ для іншого власника, але не має ролі адміністратора, така дія також блокується. Це дозволяє уникнути ситуацій, коли звичайний користувач створює ключі від імені інших користувачів.

Після успішного створення API-ключа система записує відповідну подію до журналу аудиту. У журналі зберігається інформація про те, хто створив ключ, для якого сервісу він був створений і кому належить. На рисунку 2.27 показано, що після створення ключа викликається функція record_audit. Завдяки цьому в системі зберігається історія створення API-ключів. Це важливо для адміністрування та безпеки, оскільки дозволяє перевірити, хто саме створив ключ, коли це сталося та до якого сервісу ключ був прив'язаний.

```
record_audit(
    db,
    action=AuditAction.API_KEY_CREATED,
    actor_user_id=current_user.id,
    target_type="api_key",
    target_id=api_key.id,
    details={"key_prefix": api_key.key_prefix, "service_id": api_key.service_id, "owner_id": api_key.owner_id},
    ip_address=request.client.host if request.client else None,
)
db.commit()
db.refresh(api_key)
return ApiKeyCreated(api_key=api_key, plain_key=plain_key)
```

Рисунок 2.27 – Запис події створення API-ключа до журналу аудиту

Для перегляду створених API-ключів реалізовано endpoint GET /api-keys. Його поведінка залежить від ролі користувача. Адміністратор і аудитор можуть бачити всі ключі, а розробник – тільки власні ключі (див. рисунок 2.28).

```
@router.get( path: "", response_model=list[ApiKeyRead])
def list_api_keys(db: Session = Depends(get_db), current_user: User = Depends(get_current_user)):
    if current_user.role in {UserRole.ADMIN, UserRole.AUDITOR}:
        return api_key_service.list_api_keys(db)
    return api_key_service.list_api_keys(db, owner_id=current_user.id)
```

Рисунок 2.28 – Endpoint перегляду API-ключів

Як видно з рисунка 2.28, система розмежовує доступ до списку API-ключів. Це потрібно для того, щоб користувач із роллю розробника не міг переглядати ключі інших користувачів. Водночас адміністратор і аудитор мають ширший доступ, оскільки їхні ролі пов’язані з керуванням системою та контролем безпеки.

У сервісному шарі для отримання API-ключів використовується окрема функція. Вона може повертати всі ключі або тільки ключі конкретного власника. На рисунку 2.29 показано, що список ключів сортується за датою створення у зворотному порядку. Якщо передано owner_id, система повертає тільки ключі конкретного користувача. Такий підхід дозволяє використовувати одну функцію як для адміністратора, так і для звичайного розробника.

```python
def list_api_keys(db: Session, owner_id: str | None = None) -> list[ApiKey]:
    stmt = _api_key_query().order_by(ApiKey.created_at.desc())
    if owner_id:
        stmt = stmt.where(ApiKey.owner_id == owner_id)
    return list(db.scalars(stmt))
```

Рисунок 2.29 – Отримання списку API-ключів із бази даних

Важливою функцією управління API-ключами є їх відкликання. Відкликання використовується тоді, коли ключ більше не повинен мати доступ до сервісу. При цьому ключ не видаляється з бази даних, а його статус змінюється на revoked (див. рисунок 2.30).

```python
def revoke_api_key(db: Session, api_key_id: str, actor_user_id: str, actor_is_admin: bool) -> ApiKey:
    api_key = get_api_key(db, api_key_id)
    if not actor_is_admin and api_key.owner_id != actor_user_id:
        raise ForbiddenError("You can revoke only your own API keys.")

    api_key.status = ApiKeyStatus.REVOKED
    api_key.revoked_at = datetime.now(timezone.utc)
    db.flush()
    return api_key
```

Рисунок 2.30 – Логіка відкликання API-ключа

Як показано на рисунку 2.30, користувач може відкликати тільки власний API-ключ, якщо він не є адміністратором. Адміністратор має право відкликати будь-який ключ. Після відкликання статус ключа змінюється на REVOKED, а в поле revoked_at записується час відкликання. Це дозволяє зберігати історію ключа та не втрачати інформацію про його попереднє використання.

HTTP endpoint для відкликання ключа реалізовано через маршрут POST /api-keys/{api_key_id}/revoke. Після зміни статусу система також створює запис у журналі аудиту (див. рисунок 2.31).

```python
@router.post( path: "/{api_key_id}/revoke", response_model=ApiKeyRead)
def revoke_api_key(
    api_key_id: str,
    request: Request,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user),
):
    api_key = api_key_service.revoke_api_key(
        db=db,
        api_key_id=api_key_id,
        actor_user_id=current_user.id,
        actor_is_admin=_is_admin(current_user),
    )
    record_audit(
        db,
        action=AuditAction.API_KEY_REVOKED,
        actor_user_id=current_user.id,
        target_type="api_key",
        target_id=api_key.id,
        details={"key_prefix": api_key.key_prefix},
        ip_address=request.client.host if request.client else None,
    )
    db.commit()
    db.refresh(api_key)
    return api_key
```

Рисунок 2.31 – Endpoint відкликання API-ключа

На рисунку 2.31 наведено endpoint, який відповідає за відкликання ключа. Після виконання цієї операції ключ більше не може бути використаний для доступу до сервісів. Якщо клієнт спробує виконати запит із відкликаним ключем, система поверне помилку автентифікації, оскільки статус ключа вже не є активним.

Ще однією важливою функцією є ротація API-ключа. Ротація використовується тоді, коли потрібно замінити старий ключ на новий без зміни його прав доступу. Наприклад, це може бути потрібно у випадку підозри на компрометацію ключа або в межах регулярної політики безпеки. Як видно з рисунка 2.32, під час ротації старий ключ не видаляється, а відкликається. Після цього створюється новий ключ із тим самим власником, сервісом, scopes, терміном дії та лімітом запитів. Такий підхід дозволяє безпечно замінити ключ і водночас зберегти історію старого ключа.

```python
def rotate_api_key(db: Session, api_key_id: str, actor_user_id: str, actor_is_admin: bool) -> tuple[ApiKey, ApiKey, str]:
    old_key = get_api_key(db, api_key_id)
    if not actor_is_admin and old_key.owner_id != actor_user_id:
        raise ForbiddenError("You can rotate only your own API keys.")

    old_key.status = ApiKeyStatus.REVOKED
    old_key.revoked_at = datetime.now(timezone.utc)
    db.flush()

    new_key, plain_key = create_api_key(
        db=db,
        owner_id=old_key.owner_id,
        service_id=old_key.service_id,
        name=f"{old_key.name} rotated",
        scope_ids=[scope.id for scope in old_key.scopes],
        expires_at=old_key.expires_at,
        rate_limit_per_minute=old_key.rate_limit_per_minute,
    )
    return old_key, new_key, plain_key
```

Рисунок 2.32 – Логіка ротації API-ключа

Endpoint для ротації ключа реалізовано через маршрут POST /api-keys/{api_key_id}/rotate. У відповіді система повертає ідентифікатор старого ключа, інформацію про новий ключ і новий plain_key.

На рисунку 2.33 показано, що після ротації ключа система записує подію API_KEY_ROTATED до журналу аудиту. У деталях події зберігаються префікси старого та нового ключа. Це дозволяє адміністратору відстежувати, які ключі були замінені та коли саме це сталося.

```python
@router.post( path: "/{api_key_id}/rotate", response_model=ApiKeyRotateResponse)
def rotate_api_key(
    api_key_id: str,
    request: Request,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user),
):
    old_key, new_key, plain_key = api_key_service.rotate_api_key(
        db=db,
        api_key_id=api_key_id,
        actor_user_id=current_user.id,
        actor_is_admin=_is_admin(current_user),
    )
    record_audit(
        db,
        action=AuditAction.API_KEY_ROTATED,
        actor_user_id=current_user.id,
        target_type="api_key",
        target_id=old_key.id,
        details={"old_prefix": old_key.key_prefix, "new_prefix": new_key.key_prefix},
        ip_address=request.client.host if request.client else None,
    )
    db.commit()
    db.refresh(new_key)
    return ApiKeyRotateResponse(old_key_id=old_key.id, new_api_key=new_key, plain_key=plain_key)
```

Рисунок 2.33 – Endpoint ротації API-ключа

Для перевірки коректності роботи управління API-ключами в системі передбачено автоматизовані тести. Вони перевіряють створення ключа, успішний доступ, відмову при відсутності потрібного scope, блокування відкликаного ключа та спрацювання rate limit.

Як показано на рисунку 2.34, тест створює API-ключ для сервісу billing із дозволом read:billing, після чого перевіряє доступ через endpoint /access/check. Якщо ключ створено правильно і він має потрібний scope, система повертає успішну відповідь.

```python
def test_create_api_key_and_check_access(client):
    token = bootstrap_admin(client)
    service_id, read_scope_id, _ = create_billing_service(client, token)

    response = client.post(
        "/api-keys",
        headers={"Authorization": f"Bearer {token}"},
        json={
            "name": "CI key",
            "service_id": service_id,
            "scope_ids": [read_scope_id],
            "rate_limit_per_minute": 10,
        },
    )
    assert response.status_code == 201, response.text
    plain_key = response.json()["plain_key"]
    api_key_id = response.json()["api_key"]["id"]

    response = client.post(
        "/access/check",
        headers={"X-API-Key": plain_key},
        json={"service_slug": "billing", "required_scopes": ["read:billing"]},
    )
    assert response.status_code == 200, response.text
    assert response.json()["allowed"] is True
    assert response.json()["api_key_id"] == api_key_id
```

Рисунок 2.34 – Тест створення API-ключа та перевірки доступу

Отже, у системі реалізовано повний життєвий цикл управління API-ключами: створення, збереження хешу, прив’язка до сервісу, призначення scopes, перегляд, відкликання та ротація. Така реалізація дозволяє безпечно контролювати доступ до сервісів, обмежувати права клієнтів, вести аудит дій і швидко блокувати або замінювати ключі у разі потреби.

2.5 Реалізація контролю доступу до сервісів

Контроль доступу до сервісів є основною функціональною частиною системи управління API-ключами. Якщо попередні механізми відповідають за створення користувачів, сервісів, дозволів та API-ключів, то контроль доступу визначає, чи може конкретний ключ виконати запит до певного сервісу. Саме цей етап дозволяє перевірити, чи ключ є дійсним, чи не був він відкликаний, чи

не завершився його термін дії, чи належить він до потрібного сервісу та чи має необхідні scope-дозволи.

У системі контроль доступу реалізовано через окремий endpoint /access/check та dependency-функцію api_key_guard. Endpoint /access/check може використовуватися зовнішніми сервісами для централізованої перевірки API-ключа. Тобто окремий сервіс не зберігає ключі самостійно, а передає отриманий ключ до API Key Access Manager, який приймає рішення про надання або заборону доступу.

Для опису запиту та відповіді перевірки доступу використано Pydantic-схеми AccessCheckRequest та AccessCheckResponse. Вони визначають, які саме дані передаються під час перевірки доступу до сервісу.

Як показано на рисунку 2.35, клієнт передає в запиті поле service_slug, яке визначає сервіс, до якого потрібно отримати доступ. Поле required_scopes містить список необхідних дозволів. Якщо перевірка проходить успішно, система повертає allowed: true, ідентифікатор API-ключа, власника ключа, назву сервісу та список дозволів, які має ключ.

```python
class AccessCheckRequest(BaseModel):
    service_slug: str = Field(min_length=2, max_length=80)
    required_scopes: list[str] = Field(default_factory=list)


class AccessCheckResponse(BaseModel):
    allowed: bool
    api_key_id: str
    owner_id: str
    service_slug: str
    granted_scopes: list[str]
```

Рисунок 2.35 – Схеми запиту та відповіді для перевірки доступу

Сам endpoint перевірки доступу отримує API-ключ із HTTP-заголовків, після чого передає його до сервісної функції перевірки.

На рисунку 2.36 наведено реалізацію endpoint-а /access/check. Спочатку система отримує API-ключ із заголовка X-API-Key або Authorization. Потім викликається функція verify_api_key_access, яка виконує повну перевірку ключа. Якщо ключ дійсний і має потрібні права, endpoint повертає позитивну відповідь. Якщо перевірка не проходить, система повертає помилку доступу.

```python
@router.post( path: "/access/check", response_model=AccessCheckResponse)
def check_access(
    payload: AccessCheckRequest,
    request: Request,
    x_api_key: Annotated[str | None, Header(alias="X-API-Key")] = None,
    authorization: Annotated[str | None, Header(alias="Authorization")] = None,
    db: Session = Depends(get_db),
):
    plain_key = extract_api_key_value(x_api_key, authorization)
    principal = verify_api_key_access(
        db=db,
        plain_api_key=plain_key,
        service_slug=payload.service_slug,
        required_scopes=payload.required_scopes,
        ip_address=request.client.host if request.client else None,
    )
    return AccessCheckResponse(allowed=True, **principal.__dict__)
```

Рисунок 2.36 – Endpoint /access/check для перевірки доступу до сервісу

Перед виконанням перевірки система повинна отримати значення API-ключа з HTTP-запиту. Для цього реалізовано функцію extract_api_key_value (див. рисунок 2.37).

```python
def extract_api_key_value(x_api_key: str | None, authorization: str | None) -> str:
    if x_api_key:
        return x_api_key.strip()

    if authorization:
        scheme, _, token = authorization.partition(" ")
        if scheme.lower() == "apikey" and token:
            return token.strip()

    raise UnauthorizedError("Expected X-API-Key header or Authorization: ApiKey <key>.")
```

Рисунок 2.37 – Отримання API-ключа з HTTP-заголовків

Як показано на рисунку 2.37, система підтримує два способи передачі API-ключа. Основним способом є заголовок X-API-Key, у якому передається відкритий ключ. Також підтримується формат Authorization: ApiKey <key>. Якщо клієнт не передав ключ жодним із цих способів, система повертає помилку 401 Unauthorized. Це означає, що запит не може бути оброблений без автентифікаційних даних.

Для зручного захисту окремих endpoint-ів у коді реалізовано dependency-функцію api_key_guard. Вона дозволяє прямо в маршруті вказати, до якого сервісу та з якими правами повинен мати доступ API-ключ.

На рисунку 2.38 показано універсальний механізм захисту endpoint-ів. Наприклад, якщо endpoint повинен бути доступний лише для сервісу billing із правом read:billing, у маршруті достатньо використати api_key_guard("billing", ["read:billing"]). Після цього FastAPI автоматично виконає перевірку ключа перед запуском основної логіки endpoint-а.

```
def api_key_guard(service_slug: str, required_scopes: list[str] | None = None) -> Callable:
    def dependency(
        request: Request,
        x_api_key: Annotated[str | None, Header(alias="X-API-Key")] = None,
        authorization: Annotated[str | None, Header(alias="Authorization")] = None,
        db: Session = Depends(get_db),
    ) -> ApiKeyPrincipal:
        plain_key = extract_api_key_value(x_api_key, authorization)
        client_ip = request.client.host if request.client else None
        return verify_api_key_access(
            db=db,
            plain_api_key=plain_key,
            service_slug=service_slug,
            required_scopes=required_scopes or [],
            ip_address=client_ip,
        )

    return dependency
```

Рисунок 2.38 – Dependency-функція для контролю доступу до endpoint-а

Основна перевірка доступу виконується у функції verify_api_key_access. Спочатку система хешує переданий відкритий API-ключ і шукає його в базі

даних. Як видно з рисунка 2.39, система не шукає відкритий API-ключ напряму, оскільки в базі даних зберігається тільки його хеш. Переданий ключ хешується повторно, після чого отримане значення порівнюється із записом у базі. Якщо ключ не знайдено, виконується запис події ACCESS_DENIED до журналу аудиту, а клієнт отримує помилку 401 Unauthorized.

```python
def verify_api_key_access(
    db: Session,
    plain_api_key: str,
    service_slug: str,
    required_scopes: list[str] | None = None,
    ip_address: str | None = None,
) -> ApiKeyPrincipal:
    hashed_key = hash_api_key(plain_api_key)
    api_key = db.scalar(
        select(ApiKey)
        .options( *options: selectinload(ApiKey.scopes), selectinload(ApiKey.service))
        .where(ApiKey.hashed_key == hashed_key)
    )

    if not api_key or not hmac.compare_digest(api_key.hashed_key, hashed_key):
        record_audit(
            db,
            action=AuditAction.ACCESS_DENIED,
            target_type="api_key",
            details={"reason": "invalid_api_key", "service_slug": service_slug},
            ip_address=ip_address,
        )
        db.commit()
        raise UnauthorizedError("Invalid API key.")
```

Рисунок 2.39 – Пошук і перевірка API-ключа в базі даних

Після перевірки існування ключа система перевіряє його статус. Це потрібно для того, щоб заблоковані або відкликані ключі не могли використовуватися для доступу до сервісів.

На рисунку 2.40 показано перевірку статусу API-ключа. Якщо ключ має статус, відмінний від ACTIVE, система забороняє доступ. Наприклад, якщо ключ було відкликано через механізм revoke або замінено під час ротації, він

більше не може використовуватися. При цьому подія відмови також записується до журналу аудиту.

```
now = datetime.now(timezone.utc)
if api_key.status != ApiKeyStatus.ACTIVE:
    record_audit(
        db,
        action=AuditAction.ACCESS_DENIED,
        actor_user_id=api_key.owner_id,
        target_type="api_key",
        target_id=api_key.id,
        details={"reason": f"key_{api_key.status}", "service_slug": service_slug},
        ip_address=ip_address,
    )
    db.commit()
    raise UnauthorizedError("API key is not active.")
```

Рисунок 2.40 – Перевірка активності API-ключа

Наступним етапом є перевірка терміну дії ключа. Це дозволяє створювати тимчасові API-ключі, які автоматично втрачають доступ після визначеної дати (див. рисунок 2.41).

```
    if api_key.expires_at and api_key.expires_at <= now:
        api_key.status = ApiKeyStatus.EXPIRED
        record_audit(
            db,
            action=AuditAction.ACCESS_DENIED,
            actor_user_id=api_key.owner_id,
            target_type="api_key",
            target_id=api_key.id,
            details={"reason": "expired", "service_slug": service_slug},
            ip_address=ip_address,
        )
        db.commit()
        raise UnauthorizedError("API key expired.")
```

Рисунок 2.41 – Перевірка терміну дії API-ключа

Як показано на рисунку 2.41, якщо поле expires_at заповнене і вказана дата вже минула, система змінює статус ключа на EXPIRED та відхиляє запит. Це дозволяє автоматично блокувати прострочені ключі без ручного втручання адміністратора.

Однією з головних перевірок є відповідність API-ключа сервісу. Кожен ключ у системі прив’язаний до конкретного сервісу, тому його не можна використовувати для доступу до інших сервісів.

На рисунку 2.42 наведено перевірку сервісу.

```
if not api_key.service or not api_key.service.is_active or api_key.service.slug != service_slug:
    record_audit(
        db,
        action=AuditAction.ACCESS_DENIED,
        actor_user_id=api_key.owner_id,
        target_type="api_key",
        target_id=api_key.id,
        details={"reason": "service_mismatch", "service_slug": service_slug},
        ip_address=ip_address,
    )
    db.commit()
    raise ForbiddenError("API key is not allowed for this service.")
```

Рисунок 2.42 – Перевірка відповідності API-ключа сервісу

Якщо API-ключ створений для сервісу billing, а клієнт намагається використати його для сервісу reports, система повертає помилку 403 Forbidden. Це важливо для ізоляції сервісів між собою. Навіть якщо ключ є дійсним, він не дає доступу до всіх сервісів системи, а працює лише в межах того сервісу, для якого був створений.

Після перевірки сервісу система перевіряє scopes. Scopes визначають конкретні дії, які дозволено виконувати в межах сервісу. Наприклад, ключ може мати право тільки на читання даних, але не мати права на їх зміну.

Як видно з рисунка 2.43, система формує список активних scopes, які має API-ключ, і порівнює їх зі списком дозволів, потрібних для виконання запиту. Якщо ключ має всі потрібні scopes, перевірка проходить успішно.

```
granted_scopes = {scope.code for scope in api_key.scopes if scope.is_active}
required = set(required_scopes or [])
if "*" not in granted_scopes and not required.issubset(granted_scopes):
    missing = sorted(required - granted_scopes)
    record_audit(
        db,
        action=AuditAction.ACCESS_DENIED,
        actor_user_id=api_key.owner_id,
        target_type="api_key",
        target_id=api_key.id,
        details={"reason": "missing_scopes", "missing_scopes": missing, "service_slug": service_slug},
        ip_address=ip_address,
    )
    db.commit()
    raise ForbiddenError("API key is missing required scopes.")
```

Рисунок 2.43 – Перевірка scope-дозволів API-ключа

Якщо хоча б одного дозволу бракує, запит відхиляється з помилкою 403 Forbidden. Наприклад, ключ із дозволом read:billing може переглядати дані білінгу, але не може виконувати операції, які потребують write:billing.

Також у системі передбачено спеціальний scope *. Якщо ключ має такий дозвіл, він може проходити перевірку без перелічення кожного окремого scope. Такий механізм може використовуватися для службових або адміністративних інтеграцій, однак його потрібно застосовувати обережно.

Ще одним елементом контролю доступу є обмеження кількості запитів. Для цього перед наданням доступу викликається функція check_rate_limit.

На рисунку 2.44 показано реалізацію rate limiting. Для кожного API-ключа система створює часовий інтервал тривалістю одна хвилина та рахує кількість запитів у межах цього інтервалу. Якщо кількість запитів перевищує значення rate_limit_per_minute, система повертає помилку 429 Too Many Requests. Це захищає сервіси від надмірного навантаження та обмежує можливість зловживання API-ключем.

```python
def check_rate_limit(db: Session, api_key: ApiKey, now: datetime | None = None) -> None:
    if api_key.rate_limit_per_minute <= 0:
        return

    now = now or datetime.now(timezone.utc)
    window_start = now.replace(second=0, microsecond=0)
    bucket = db.scalar(
        select(RateLimitBucket).where(
            *whereclause: RateLimitBucket.api_key_id == api_key.id,
            RateLimitBucket.window_start == window_start,
        )
    )

    if not bucket:
        bucket = RateLimitBucket(api_key_id=api_key.id, window_start=window_start, requests_count=1)
        db.add(bucket)
        db.flush()
        return

    if bucket.requests_count >= api_key.rate_limit_per_minute:
        retry_after = 60 - now.second
        raise RateLimitError(retry_after_seconds=max(retry_after, 1))

    bucket.requests_count += 1
    db.flush()
```

Рисунок 2.44 – Реалізація обмеження кількості запитів для API-ключа

Якщо всі перевірки пройдено успішно, система оновлює службову інформацію про ключ, записує подію успішного використання до журналу аудиту та повертає об’єкт ApiKeyPrincipal.

На рисунку 2.45 показано завершальний етап контролю доступу. Якщо API-ключ є дійсним, належить до потрібного сервісу, має необхідні scopes і не перевищив ліміт запитів, система дозволяє доступ. Додатково оновлюється поле last_used_at, збільшується usage_count і створюється запис API_KEY_USED у журналі аудиту. Це дозволяє відстежувати, які ключі реально використовуються.

```
        check_rate_limit(db, api_key, now=now)
        api_key.last_used_at = now
        api_key.usage_count += 1
        record_audit(
            db,
            action=AuditAction.API_KEY_USED,
            actor_user_id=api_key.owner_id,
            target_type="api_key",
            target_id=api_key.id,
            details={"service_slug": service_slug, "required_scopes": sorted(required)},
            ip_address=ip_address,
        )
        db.commit()

        return ApiKeyPrincipal(
            api_key_id=api_key.id,
            owner_id=api_key.owner_id,
            service_id=api_key.service_id,
            service_slug=api_key.service.slug,
            granted_scopes=sorted(granted_scopes),
        )
```

Рисунок 2.45 – Успішне надання доступу до сервісу

Практичне застосування контролю доступу можна побачити на прикладі захищеного endpoint-а /demo/billing-reports. Цей endpoint доступний тільки для ключів, які належать до сервісу billing і мають дозвіл read:billing.

Як показано на рисунку 2.46, у параметрах api_key_guard явно зазначено сервіс billing і необхідний дозвіл read:billing. Це означає, що доступ до endpoint-а отримає тільки той клієнт, який передав валідний API-ключ саме для цього сервісу. Якщо клієнт передасть ключ для іншого сервісу або ключ без потрібного scope, запит буде відхилено.

```
@router.get("/demo/billing-reports")
def demo_billing_reports(
    principal: ApiKeyPrincipal = Depends(api_key_guard( service_slug: "billing",  required_scopes: ["read:billing"])),
):
    return {
        "service": "billing",
        "message": "Protected billing report data.",
        "api_key_id": principal.api_key_id,
        "owner_id": principal.owner_id,
    }
```

Рисунок 2.46 – Приклад захищеного endpoint-а сервісу billing

Для демонстрації різниці доступів можна розглянути кілька сценаріїв. Перший сценарій – успішний доступ, коли ключ має потрібний scope для потрібного сервісу.

На рисунку 2.47 показано запит, у якому використовується API-ключ із дозволом read:billing. Оскільки ключ належить до сервісу billing і має потрібний scope, система повертає успішну відповідь з HTTP-кодом 200.

```
curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
-H "X-API-Key: $BILLING_READ_KEY" \
-H "Content-Type: application/json" \
-d '{"service_slug":"billing","required_scopes":["read:billing"]}'
```

Рисунок 2.47 – Приклад успішної перевірки доступу до сервісу billing

Другий сценарій – відмова через відсутність необхідного scope. Наприклад, якщо ключ має тільки read:billing, але клієнт намагається виконати дію, яка потребує write:billing, доступ буде заборонено (див. рисунок 2.7).

```
curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
-H "X-API-Key: $BILLING_READ_KEY" \
-H "Content-Type: application/json" \
-d '{"service_slug":"billing","required_scopes":["write:billing"]}'
```

Рисунок 2.48 – Приклад відмови через відсутність необхідного scope

Як видно з рисунка 2.48, у запиті вимагається scope write:billing, однак ключ має лише право читання. У такому випадку система повертає HTTP-код 403 Forbidden. Це означає, що клієнт автентифікований, але не має достатніх прав для виконання цієї дії.

Третій сценарій – спроба використання ключа для іншого сервісу. Наприклад, ключ, створений для сервісу billing, не повинен працювати із сервісом reports.

На рисунку 2.49 показано ситуацію, коли API-ключ використовується не для свого сервісу. Навіть якщо ключ є валідним, система перевіряє поле

service_slug і порівнює його з сервісом, до якого прив'язаний ключ. Якщо вони не збігаються, доступ забороняється з HTTP-кодом 403 Forbidden.

```
curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
-H "X-API-Key: $BILLING_READ_KEY" \
-H "Content-Type: application/json" \
-d '{"service_slug":"reports","required_scopes":["read:reports"]}'
```

Рисунок 2.49 – Приклад відмови через невідповідність сервісу

Для перевірки коректності контролю доступу в системі реалізовано автоматизовані тести. Один із тестів перевіряє ситуацію, коли API-ключ не має потрібного scope.

Як показано на рисунку 2.50, тест створює ключ із правом read:billing, а потім намагається перевірити доступ із вимогою write:billing. Очікуваним результатом є відповідь 403, що підтверджує правильну роботу механізму scope-авторизації.

```python
def test_missing_scope_is_forbidden(client):
    token = bootstrap_admin(client)
    service_id, read_scope_id, _ = create_billing_service(client, token)

    response = client.post(
        "/api-keys",
        headers={"Authorization": f"Bearer {token}"},
        json={"name": "Read only", "service_id": service_id, "scope_ids": [read_scope_id]},
    )
    assert response.status_code == 201, response.text
    plain_key = response.json()["plain_key"]

    response = client.post(
        "/access/check",
        headers={"X-API-Key": plain_key},
        json={"service_slug": "billing", "required_scopes": ["write:billing"]},
    )
    assert response.status_code == 403
```

Рисунок 2.50 – Тест відмови в доступі через відсутній scope

Окремо перевіряється сценарій, коли API-ключ було відкликано. Такий ключ більше не повинен проходити перевірку доступу.

На рисунку 2.51 показано, що після відкликання ключа система більше не дозволяє використовувати його для доступу до сервісу. У цьому випадку очікуваним результатом є HTTP-код 401 Unauthorized, оскільки ключ уже не є активним.

```python
def test_revoked_key_is_rejected(client):
    token = bootstrap_admin(client)
    service_id, read_scope_id, _ = create_billing_service(client, token)

    response = client.post(
        "/api-keys",
        headers={"Authorization": f"Bearer {token}"},
        json={"name": "Temporary", "service_id": service_id, "scope_ids": [read_scope_id]},
    )
    assert response.status_code == 201, response.text
    body = response.json()

    response = client.post(
        f"/api-keys/{body['api_key']['id']}/revoke",
        headers={"Authorization": f"Bearer {token}"},
    )
    assert response.status_code == 200, response.text

    response = client.post(
        "/access/check",
        headers={"X-API-Key": body["plain_key"]},
        json={"service_slug": "billing", "required_scopes": ["read:billing"]},
    )
    assert response.status_code == 401
```

Рисунок 2.51 – Тест блокування відкликаного API-ключа

Також перевіряється робота обмеження кількості запитів. Якщо API-ключ перевищує встановлений ліміт, система повинна повернути помилку 429.

Як видно з рисунка 2.52, для ключа встановлюється ліміт один запит за хвилину. Перший запит проходить успішно, а другий у межах того самого часового вікна завершується помилкою 429 Too Many Requests. Це підтверджує, що система контролює не тільки права доступу, а й інтенсивність використання API-ключа.

```python
def test_rate_limit_is_enforced(client):
    token = bootstrap_admin(client)
    service_id, read_scope_id, _ = create_billing_service(client, token)

    response = client.post(
        "/api-keys",
        headers={"Authorization": f"Bearer {token}"},
        json={
            "name": "Rate limited",
            "service_id": service_id,
            "scope_ids": [read_scope_id],
            "rate_limit_per_minute": 1,
        },
    )
    assert response.status_code == 201, response.text
    plain_key = response.json()["plain_key"]

    payload = {"service_slug": "billing", "required_scopes": ["read:billing"]}
    assert client.post("/access/check", headers={"X-API-Key": plain_key}, json=payload).status_code == 200
    response = client.post("/access/check", headers={"X-API-Key": plain_key}, json=payload)
    assert response.status_code == 429
```

Рисунок 2.52 – Тест спрацювання обмеження кількості запитів

Отже, реалізований механізм контролю доступу до сервісів забезпечує комплексну перевірку API-ключів. Система перевіряє наявність ключа, його хеш, статус, термін дії, відповідність сервісу, наявність потрібних scopes та rate limit. Завдяки цьому можна створювати різні сценарії доступу: ключ лише для читання, ключ із правом запису, ключ для окремого сервісу або ключ із тимчасовим терміном дії. Такий підхід дозволяє безпечно керувати доступом до кількох сервісів і чітко розмежовувати права між клієнтами.

2.6 Висновок до другого розділу

У другому розділі було розглянуто реалізацію системи управління API-ключами та доступом до сервісів на основі FastAPI-застосунку. Описано архітектуру системи, яка включає модулі автентифікації, керування користувачами, сервісами, API-ключами, перевірки доступу, rate limiting та журналювання подій.

Було спроєктовано базу даних із таблицями користувачів, сервісів, scope-дозволів, API-ключів, журналу аудиту та обмеження кількості запитів. Така

структура дозволяє прив’язувати API-ключі до конкретних сервісів і надавати їм окремі права доступу. Завдяки цьому ключ одного сервісу не може використовуватися для іншого, а дії клієнта контролюються через scopes.

У системі реалізовано JWT-автентифікацію для користувачів і модель доступу і модель доступу на основі ролей з ролями адміністратора, розробника та аудитора. Для зовнішніх клієнтів використовується механізм API-ключів, які передаються через заголовок X-API-Key. Відкритий ключ показується лише один раз під час створення, а в базі даних зберігається тільки його хеш.

Також реалізовано створення, перегляд, відкликання та ротацію API-ключів. Контроль доступу виконується через перевірку ключа, його статусу, терміну дії, відповідності сервісу, наявності потрібних scope-дозволів і дотримання ліміту запитів. У разі помилки система повертає відповідні HTTP-коди: 401, 403 або 429.

Усі важливі дії записуються до журналу аудиту, що дозволяє відстежувати використання ключів, їх зміну та спроби несанкціонованого доступу. Отже, реалізована система забезпечує централізоване керування API-ключами, розмежування доступу між сервісами та контроль безпечного використання захищених ресурсів.

РОЗДІЛ 3. ТЕСТУВАННЯ РОБОТИ СИСТЕМИ УПРАВЛІННЯ API-КЛЮЧАМИ ТА ДОСТУПОМ ДО СЕРВІСІВ

3.1 Підготовка та запуск тестового середовища

Для перевірки працездатності розробленої системи було проведено тестування основних функціональних можливостей: запуску серверної частини, автентифікації адміністратора, перегляду сервісів, створення API-ключів, перевірки доступу до сервісів, обмеження кількості запитів, ротації, відкликання ключів та перегляду журналу аудиту.

Тестування виконувалося у локальному середовищі в межах проєкту api-key-access-manager. На початковому етапі було активовано віртуальне середовище Python та виконано ініціалізацію бази даних із початковими тестовими даними (див. рисунок 3.1).

```
(.venv) api-key-access-manager > python -m app.cli seed
Database schema is ready.
Seed data is ready.
Admin login: admin@example.com / Admin12345!
Developer login: developer@example.com / Developer12345!
```

Рисунок 3.1 – Створення початкових тестових даних у системі

На рисунку 3.1 показано результат виконання команди ініціалізації. У терміналі відображено повідомлення Database schema is ready, що підтверджує створення структури бази даних. Також повідомлення Seed data is ready підтверджує успішне додавання початкових даних. У результаті було створено тестові облікові записи адміністратора та розробника, які надалі використовуються для перевірки роботи системи.

Після підготовки бази даних було запущено серверну частину системи за допомогою Uvicorn.

На рисунку 3.2 показано, що FastAPI-застосунок успішно запущено на локальній адресі http://127.0.0.1:8000. У терміналі відображено повідомлення про запуск процесу сервера, очікування старту застосунку та завершення ініціалізації. Це підтверджує, що серверна частина системи готова приймати HTTP-запити.

```
(.venv) api-key-access-manager > uvicorn app.main:app --reload --host 127.0.0.1 --port 8000
INFO:     Will watch for changes in these directories: ['/Users/[redacted]/PycharmProjects/api-key-access-manager']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [11159] using WatchFiles
INFO:     Started server process [11161]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Рисунок 3.2 – Запуск серверної частини системи через Uvicorn

Для додаткової перевірки було виконано запит до службового endpoint-а /health.

На рисунку 3.3 показано відповідь системи зі статусом ok та середовищем виконання dev. Це свідчить про те, що сервер працює коректно, а застосунок доступний для подальшого тестування.

```
(.venv) api-key-access-manager > curl -s http://127.0.0.1:8000/health | python -m json.tool
{
    "status": "ok",
    "environment": "dev"
}
```

Рисунок 3.3 – Перевірка працездатності серверної частини через endpoint /health

Отже, на першому етапі було підтверджено, що тестове середовище підготовлене, база даних ініціалізована, початкові користувачі створені, а серверна частина системи успішно запущена.

3.2 Перевірка автентифікації, сервісів і створення API-ключа

Після запуску системи було перевірено механізм автентифікації адміністратора. Для цього виконано вхід у систему з використанням тестового облікового запису адміністратора. Після успішного входу система повернула JWT-токен, який надалі використовувався для доступу до захищених адміністративних endpoint-ів.

На рисунку 3.4 показано результат перевірки поточного користувача після авторизації. У відповіді системи відображено email admin@example.com, повне ім’я System Admin, роль admin та активний статус користувача. Це підтверджує, що механізм JWT-автентифікації працює коректно, а отриманий токен дозволяє виконувати запити від імені адміністратора.

```
(.venv) api-key-access-manager > export TOKEN=$(curl -s -X POST http://127.0.0.1:8000/auth/login \
  -H 'Content-Type: application/json' \
  -d '{"email":"admin@example.com","password":"Admin12345!"}' \
  | python -c 'import sys,json; print(json.load(sys.stdin)["access_token"])')
```

```
(.venv) api-key-access-manager > curl -s http://127.0.0.1:8000/auth/me \
  -H "Authorization: Bearer $TOKEN" \
  | python -m json.tool
{
    "id": "bff9ec3e-4446-4512-950b-5424f2b0d9bc",
    "email": "admin@example.com",
    "full_name": "System Admin",
    "role": "admin",
    "is_active": true
}
```

Рисунок 3.4 – Перевірка автентифікації адміністратора через JWT-токен

Наступним етапом було перевірено перегляд доступних сервісів і scope-дозволів. У системі після ініціалізації створено демонстраційні сервіси, зокрема billing, який має дозволи read:billing та write:billing.

На рисунку 3.5 показано відповідь системи зі списком сервісів. Для сервісу billing відображено його ідентифікатор, назву, опис, активний статус і

список дозволів. Наявність scopes read:billing та write:billing підтверджує, що система підтримує розмежування прав доступу всередині окремого сервісу.

```
(.venv) api-key-access-manager > curl -s http://127.0.0.1:8000/services \
  -H "Authorization: Bearer $TOKEN" \
  | python -m json.tool
[
    {
        "id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
        "slug": "billing",
        "name": "Billing Service",
        "description": "Demo billing service protected by API keys.",
        "is_active": true,
        "created_at": "2026-06-05T11:40:38.200999",
        "scopes": [
            {
                "id": "539fc421-e65c-40d3-b83a-0304bcdd4ff9",
                "service_id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
                "code": "read:billing",
                "description": "Read billing data.",
                "is_active": true,
                "created_at": "2026-06-05T11:40:38.202219"
            },
            {
                "id": "17c7ba4d-b9a3-4cc2-8b8a-a1269e42d97b",
                "service_id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
                "code": "write:billing",
                "description": "Write billing data.",
                "is_active": true,
                "created_at": "2026-06-05T11:40:38.202222"
            }
        ]
    },
```

Рисунок 3.5 – Перегляд сервісів і scope-дозволів

Після перегляду сервісів було отримано ідентифікатор сервісу billing та ідентифікатор scope-дозволу read:billing. Ці значення були використані для створення API-ключа, який повинен мати доступ лише до читання даних сервісу billing.

На рисунку 3.6 показано процес формування запиту на створення API-ключа. У запиті використовується JWT-токен адміністратора, ідентифікатор сервісу billing, scope read:billing та ліміт запитів 60 за хвилину.

```
(.venv) api-key-access-manager > python -m json.tool /tmp/api-key-response.json
{
    "api_key": {
        "id": "683e232c-4134-46cd-b2ba-e746fd7bb7df",
        "owner_id": "bff9ec3e-4446-4512-950b-5424f2b0d9bc",
        "service_id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
        "name": "Billing reader test",
        "key_prefix": "ak_3a8d8fc0",
        "status": "active",
        "rate_limit_per_minute": 60,
        "usage_count": 0,
        "created_at": "2026-06-05T20:49:01.890183",
        "expires_at": null,
        "revoked_at": null,
        "last_used_at": null,
        "scopes": [
            {
                "id": "539fc421-e65c-40d3-b83a-0304bcdd4ff9",
                "service_id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
                "code": "read:billing",
                "description": "Read billing data.",
                "is_active": true,
                "created_at": "2026-06-05T11:40:38.202219"
            }
        ]
    },
    "plain_key": "ak_3a8d8fc0.JfqE3Df4Gy2Pio5xQg6mmJlWgm7lBXba7EEswPQKyw4"
}
```

Рисунок 3.6 – Формування запиту на створення API-ключа для сервісу billing

Це демонструє, що створення API-ключа виконується авторизованим користувачем і прив’язується до конкретного сервісу та конкретного дозволу.

На рисунку 3.7 показано відповідь системи після створення API-ключа. У відповіді міститься об’єкт api_key, де зазначено ідентифікатор ключа, власника, сервіс, назву ключа, префікс, статус active, ліміт запитів, кількість використань та список наданих scopes. Також у відповіді присутнє поле plain_key, яке містить відкритий API-ключ.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["read:billing"]}'
{"allowed":true,"api_key_id":"be1da9b1-e683-4e58-87e8-bffc67be7a28","owner_id":"bff9ec3e-4446-4512-950b-5424f2b0d
HTTP 200

(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["write:billing"]}'
{"detail":"API key is missing required scopes."}
HTTP 403

(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"reports","required_scopes":["read:reports"]}'
{"detail":"API key is not allowed for this service."}
HTTP 403
```

Рисунок 3.7 – Результат створення API-ключа

Важливо, що відкритий API-ключ показується лише один раз під час створення. Надалі в базі даних зберігається не сам ключ, а його хешоване значення. Це підвищує безпеку системи, оскільки навіть у разі доступу до бази даних неможливо отримати справжнє значення ключа.

Отже, у межах цього підрозділу було підтверджено, що адміністратор може пройти автентифікацію, переглянути доступні сервіси та створити API-ключ із конкретним scope-дозволом.

3.3 Тестування різних сценаріїв доступу до сервісів

Після створення API-ключа було перевірено основні сценарії контролю доступу. Метою цього етапу було підтвердити, що система дозволяє доступ тільки тоді, коли ключ є активним, належить до потрібного сервісу та має необхідний scope-дозвіл.

Першим було протестовано позитивний сценарій. Створений API-ключ належить до сервісу billing і має дозвіл read:billing, тому система повинна дозволити доступ.

На рисунку 3.8 показано успішну відповідь системи під час перевірки доступу. У відповіді зазначено allowed: true, ідентифікатор API-ключа, власника, сервіс billing і наданий scope read:billing. HTTP-код 200 підтверджує, що доступ дозволено.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' http://127.0.0.1:8000/demo/billing-reports
{"detail":"Expected X-API-Key header or Authorization: ApiKey <key>."}
HTTP 401

(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' http://127.0.0.1:8000/demo/billing-reports \
  -H "X-API-Key: $API_KEY"
{"service":"billing","message":"Protected billing report data.","api_key_id":"be1da9b1-e683-4e58-87e8-bffc67be7a2
HTTP 200
```

Рисунок 3.8 – Успішна перевірка доступу до сервісу billing

Другим було перевірено сценарій, коли API-ключ не має потрібного scope-дозволу. Для цього ключ із правом read:billing було використано для дії, яка потребує write:billing.

На рисунку 3.9 показано, що система повертає повідомлення API key is missing required scopes та HTTP-код 403. Це означає, що ключ є дійсним, але не має достатніх прав для виконання запитаної дії. Такий результат підтверджує коректну роботу scope-авторизації.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["write:billing"]}'
{"detail":"API key is missing required scopes."}
HTTP 403
```

Рисунок 3.9 – Відмова в доступі через відсутність scope-дозволу write:billing

Третім було перевірено сценарій використання API-ключа для іншого сервісу. Ключ було створено для сервісу billing, однак у запиті вказано сервіс reports.

На рисунку 3.10 показано, що система повертає повідомлення API key is not allowed for this service та HTTP-код 403. Це підтверджує, що API-ключ не

може використовуватися поза межами сервісу, до якого він був прив’язаний під час створення.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"reports","required_scopes":["read:reports"]}'
{"detail":"API key is not allowed for this service."}
HTTP 403
```

Рисунок 3.10 – Відмова в доступі через невідповідність сервісу

Додатково було перевірено роботу захищеного endpoint-a /demo/billing-reports. Спочатку запит було виконано без API-ключа, після чого – з валідним API-ключем.

На рисунку 3.11 показано два результати. У першому випадку система повертає HTTP-код 401 і повідомлення про відсутність заголовка X-API-Key або Authorization: ApiKey <key>. У другому випадку, коли передано валідний API-ключ, система повертає дані захищеного ресурсу та HTTP-код 200. Це підтверджує, що endpoint захищений і доступний лише для клієнтів із коректним API-ключем.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' http://127.0.0.1:8000/demo/billing-reports
{"detail":"Expected X-API-Key header or Authorization: ApiKey <key>."}
HTTP 401

(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' http://127.0.0.1:8000/demo/billing-reports \
  -H "X-API-Key: $API_KEY"
{"service":"billing","message":"Protected billing report data.","api_key_id":"be1da9b1-e683-4e58-87e8-bffc67be7a2
HTTP 200
```

Рисунок 3.11 – Перевірка доступу до захищеного endpoint-a без ключа та з валідним API-ключем

Отже, тестування різних сценаріїв доступу показало, що система правильно обробляє як дозволені, так і заборонені запити. Доступ надається тільки за наявності правильного сервісу та потрібного scope-дозволу.

3.4 Перевірка обмеження запитів, ротації та відкликання API-ключа

На наступному етапі було перевірено додаткові механізми контролю API-ключів: обмеження кількості запитів, ротацію та відкликання. Ці функції потрібні для керування життєвим циклом ключів і зменшення ризиків у випадку їхнього надмірного використання або компрометації.

Для перевірки rate limiting було створено окремий API-ключ із лімітом один запит за хвилину. Після цього було виконано два однакові запити підряд.

На рисунку 3.12 показано, що перший запит із новим API-ключем завершується успішно з HTTP-кодом 200. Другий запит, виконаний у межах тієї самої хвилини, повертає повідомлення Rate limit exceeded та HTTP-код 429. Це підтверджує, що система контролює кількість запитів для API-ключа та блокує перевищення встановленого ліміту.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $RATE_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["read:billing"]}'
{"allowed":true,"api_key_id":"cbd6547d-e9ad-4ee2-bc82-d020e7a38d0a","owner_id":"bff9ec3e-4446-4512-950b-5424f2b0c
HTTP 200

(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $RATE_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["read:billing"]}'
{"detail":"Rate limit exceeded."}
HTTP 429
```

Рисунок 3.12 – Перевірка спрацювання rate limiting для API-ключа

Далі було перевірено механізм ротації API-ключа. Ротація використовується для заміни старого ключа на новий із тими самими правами доступу. Після виконання ротації старий ключ повинен стати неактивним, а новий – працювати.

На рисунку 3.13 показано результат перевірки старого та нового ключа після ротації. Під час використання старого ключа система повертає повідомлення API key is not active та HTTP-код 401. Під час використання

нового ключа система повертає allowed: true та HTTP-код 200. Це підтверджує, що ротація виконана коректно: старий ключ заблоковано, а новий ключ отримав ті самі права доступу.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["read:billing"]}'
{"detail":"API key is not active."}
HTTP 401

(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $NEW_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["read:billing"]}'
{"allowed":true,"api_key_id":"3c49f08a-d33f-4b31-9010-7623b52a8976","owner_id":"bff9ec3e-4446-4512-950b-5424f2b0c
HTTP 200
```

Рисунок 3.13 – Перевірка ротації API-ключа

Після ротації було перевірено механізм відкликання API-ключа. Відкликання використовується тоді, коли ключ більше не повинен мати доступ до сервісу.

На рисунку 3.14 показано відповідь системи після відкликання ключа. У полі status зазначено значення revoked, а поле revoked_at містить час відкликання. Це підтверджує, що система не видаляє ключ із бази даних, а змінює його статус, зберігаючи історію його існування.

```
(.venv) api-key-access-manager > curl -s -X POST http://127.0.0.1:8000/api-keys/$NEW_API_KEY_ID/revoke \
  -H "Authorization: Bearer $TOKEN" \
  | python -m json.tool
{
    "id": "3c49f08a-d33f-4b31-9010-7623b52a8976",
    "owner_id": "bff9ec3e-4446-4512-950b-5424f2b0d9bc",
    "service_id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
    "name": "Billing reader test rotated",
    "key_prefix": "ak_3707f677",
    "status": "revoked",
    "rate_limit_per_minute": 60,
    "usage_count": 1,
    "created_at": "2026-06-05T20:53:09.284045",
    "expires_at": null,
    "revoked_at": "2026-06-05T20:58:27.522091",
    "last_used_at": "2026-06-05T20:58:07.187594",
    "scopes": [
        {
            "id": "539fc421-e65c-40d3-b83a-0304bcdd4ff9",
            "service_id": "b6ea19a6-2e53-482b-b765-7fea0a314f93",
            "code": "read:billing",
            "description": "Read billing data.",
            "is_active": true,
            "created_at": "2026-06-05T11:40:38.202219"
        }
    ]
}
```

Рисунок 3.14 – Відкликання API-ключа

Після відкликання було виконано повторну перевірку доступу з використанням цього ключа.

На рисунку 3.15 показано, що після відкликання ключа система повертає повідомлення API key is not active та HTTP-код 401. Це підтверджує, що відкликаний ключ більше не може використовуватися для доступу до сервісу.

```
(.venv) api-key-access-manager > curl -s -w '\nHTTP %{http_code}\n' -X POST http://127.0.0.1:8000/access/check \
  -H "X-API-Key: $NEW_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"service_slug":"billing","required_scopes":["read:billing"]}'
{"detail":"API key is not active."}
HTTP 401
```

Рисунок 3.15 – Відмова в доступі після відкликання API-ключа

Таким чином, у межах цього підрозділу було підтверджено, що система підтримує повний життєвий цикл API-ключа: обмеження кількості запитів, ротацію та відкликання.

3.5 Перевірка журналу аудиту

Завершальним етапом тестування була перевірка журналу аудиту. Журнал аудиту використовується для фіксації важливих подій у системі: входу користувачів, створення API-ключів, використання ключів, відмов у доступі, ротації та відкликання. На рисунку 3.16 показано частину журналу аудиту.

```
(.venv) api-key-access-manager > curl -s http://127.0.0.1:8000/audit-logs \
  -H "Authorization: Bearer $TOKEN" \
  | python -m json.tool
[
    {
        "id": "6b7f11f7-ed86-4279-9ce9-1945627b6e53",
        "actor_user_id": "bff9ec3e-4446-4512-950b-5424f2b0d9bc",
        "action": "access_denied",
        "target_type": "api_key",
        "target_id": "3c49f08a-d33f-4b31-9010-7623b52a8976",
        "ip_address": "127.0.0.1",
        "details": {
            "reason": "key_revoked",
            "service_slug": "billing"
        },
        "created_at": "2026-06-05T20:58:41.484344"
    },
    {
        "id": "8fd47e63-e84a-4454-a041-8b5279aabaa4",
        "actor_user_id": "bff9ec3e-4446-4512-950b-5424f2b0d9bc",
        "action": "api_key_revoked",
        "target_type": "api_key",
        "target_id": "3c49f08a-d33f-4b31-9010-7623b52a8976",
        "ip_address": "127.0.0.1",
        "details": {
            "key_prefix": "ak_3707f677"
        },
        "created_at": "2026-06-05T20:58:27.523522"
    },
```

Рисунок 3.16 – Перегляд журналу аудиту системи

У ньому зафіксовано події api_key_revoked та access_denied. Для кожної події зберігається ідентифікатор користувача, тип об’єкта, ідентифікатор API-ключа, IP-адреса, додаткові деталі та час створення запису.

Запис api_key_revoked підтверджує факт відкликання ключа, а запис access_denied показує спробу використання вже відкликаного ключа. У полі details зазначено причину відмови, зокрема key_revoked, а також сервіс billing, до якого здійснювалася спроба доступу.

Наявність таких записів підтверджує, що система фіксує не лише успішні дії, а й відмови в доступі. Це важливо для контролю безпеки, оскільки адміністратор може переглядати історію використання ключів, аналізувати підозрілі запити та визначати причини блокування доступу.

У результаті тестування було підтверджено працездатність основних функцій системи. Серверна частина успішно запускається, адміністратор проходить автентифікацію, сервіси та scopes коректно відображаються, API-ключі створюються і прив’язуються до конкретних сервісів. Механізм контролю доступу правильно дозволяє запити з валідними правами та блокує запити без потрібного scope, для іншого сервісу або з неактивним ключем. Також підтверджено роботу rate limiting, ротації, відкликання API-ключів і журналу аудиту.

3.6 Висновок до третього розділу

У третьому розділі було проведено тестування роботи системи управління API-ключами та доступом до сервісів. Було перевірено повний процес роботи системи: підготовку тестового середовища, запуск серверної частини, авторизацію адміністратора, перегляд доступних сервісів і scope-дозволів, створення API-ключа та його використання для доступу до захищених ресурсів. Під час тестування підтверджено, що FastAPI-застосунок коректно запускається, відповідає на запити через endpoint /health і забезпечує роботу основних модулів системи.

Також було перевірено механізм автентифікації адміністратора за допомогою JWT-токена. Після входу в систему адміністратор отримує токен доступу, який використовується для виконання захищених адміністративних

дій. Було підтверджено можливість перегляду сервісів, зокрема сервісу billing, а також його scope-дозволів read:billing і write:billing. Це показало, що система правильно зберігає сервіси та дозволяє використовувати їх для подальшого створення API-ключів.

У процесі тестування було створено API-ключ для сервісу billing із дозволом read:billing. Система повернула відкритий ключ лише один раз під час створення, а також надала службову інформацію про ключ: його ідентифікатор, власника, сервіс, статус, ліміт запитів і список дозволів. Це підтвердило правильність реалізації механізму створення API-ключів і їх прив’язки до конкретного сервісу та scope-дозволу.

Було перевірено декілька сценаріїв контролю доступу. Успішний запит із валідним ключем і дозволом read:billing повернув HTTP-код 200, що підтверджує можливість доступу до дозволеного сервісу. Запит із вимогою write:billing для ключа, який має лише read:billing, завершився помилкою 403, що підтверджує коректну перевірку scope-дозволів. Також спроба використати ключ сервісу billing для доступу до сервісу reports завершилася помилкою 403, що показує правильну ізоляцію доступу між різними сервісами.

Додатково було протестовано захищений endpoint /demo/billing-reports. Без API-ключа система повернула помилку 401, а з валідним ключем надала доступ до ресурсу з HTTP-кодом 200. Це підтвердило, що endpoint захищений і доступний лише для клієнтів, які передають коректний API-ключ.

У розділі також було перевірено додаткові механізми безпеки: обмеження кількості запитів, ротацію та відкликання API-ключа. Під час перевірки rate limiting перший запит із ключем було виконано успішно, а повторний запит у межах встановленого обмеження завершився помилкою 429. Після ротації старий ключ став неактивним, а новий ключ отримав можливість доступу до сервісу. Після відкликання нового ключа система також заблокувала його використання та повернула помилку 401.

Завершальним етапом було перевірено журнал аудиту. У ньому були зафіксовані події відкликання ключа та відмови в доступі, зокрема із

зазначенням причини, ідентифікатора ключа, IP-адреси та часу створення запису. Це підтвердило, що система зберігає історію важливих дій і дозволяє контролювати використання API-ключів.

Отже, проведене тестування підтвердило працездатність розробленої системи. Система коректно виконує автентифікацію користувачів, створює API-ключі, перевіряє доступ до сервісів, розмежовує права за scopes, блокує неактивні або відкликані ключі, обмежує кількість запитів і фіксує важливі події в журналі аудиту. Усі отримані результати підтверджують, що реалізована система забезпечує контрольований і безпечний доступ до захищених сервісів.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Фактори ризику і можливі порушення здоров’я користувачів комп’ютерної мережі

Велика кількість людей проводить багато часу за комп’ютером, що збільшує ризики та можливі негативні наслідки для здоров’я. Довге сидіння за комп’ютером призводить до різних проблем із здоров’ям. Найпоширенішими причин погіршення стану здоров’я є порушення постави та біль у спині [34]. Неправильна постава, яка зумовлена внаслідок погано організованого робочого місця або неправильно підібраного комп’ютерного стільця, це може спричинити хронічні болі у спині, шиї та плечах. Також поширена проблема із зап’ястям та кистями, наприклад синдром зап’ястного каналу, який виникає через повторюванні рухи, наприклад використання миші чи клавіатури [34]. Ще одним серйозним ризиком, пов’язаним із тривалим використанням комп’ютера, є проблеми зі здоров’ям очей. Проведення надто тривалого часу перед екраном може призвести до таких симптомів, як погіршенням зору або напруження очей, яке характеризується сухістю, свербінням, подразненням і втомою очей. Робота за комп'ютером часто пов'язана з високим рівнем стресу, особливо в умовах коли терміни стислі, а вимоги високі. Це може призвести до вигоряння, депресії та інших психологічних проблем. Крім того, ізоляція, яка часто супроводжує роботу вдома або в невеликих кабінетах, може вплинути на соціальне та емоційне самопочуття.

Також важливо розглянути вплив на якість сну та загальне фізичне здоров'я. Багато користувачів комп'ютерів скаржаться на порушення сну, що може бути пов'язано з надмірним впливом на світлові екрани перед сном. Це світло пригнічує вироблення мелатоніну, гормону сну, заважаючи людям засинати та підтримувати здоровий цикл сну. Використання спеціальних програм або окулярів для блокування синього світла є важливим для тих, хто регулярно працює з комп'ютерами ввечері. Фізична неактивність, яка часто

супроводжує довготривалу роботу за комп'ютером, також є фактором ризику для ряду хронічних захворювань, серцево-судинні захворювання та діабет другого типу. Недостатня фізична активність призводить до уповільнення метаболізму, що може спричинити набір ваги і зниження загальної фізичної форми. Це підвищує ризик розвитку серцевих захворювань.

Залежність від Інтернету та соціальних медіа є ще однією проблемою, з якою стикаються користувачі комп'ютерних мереж. Ця залежність може призвести до зниження продуктивності, соціальної ізоляції та погіршення психічного здоров'я. Важливо встановлювати межі та регулярно проводити час без використання електронних пристроїв.

Для зменшення цих ризиків існують різні стратегії. По-перше, важливо правильно організувати робоче місце. Ергономічні крісла, належно розташовані монітори та клавіатури, підставка для ніг, регулярні перерви для руху та розтяжки можуть значно знизити ризик мускул скелетних проблем. Крім того, використання спеціальних окулярів або застосування програм, що знижують синє світло від екранів, можуть допомогти зменшити втому очей.

Що стосується психологічного здоров'я, важливо забезпечити баланс між роботою та особистим життям, включаючи регулярні перерви, хобі, соціальну взаємодію та вправи на свіжому повітрі. Також корисною може бути практика медитації для зниження рівня стресу.

4.2 Забезпечення захисту працівників суб’єкта господарювання від іонізуючого випромінювання

Працівники, які виконують роботи з радіоактивними речовинами, повинні перебувати під постійним медичним наглядом, використовувати засоби індивідуального захисту від радіації та прилади індивідуального дозиметричного контролю (універсальні радіометри) для своєчасного виявлення і вимірювання рівня випромінювання [35].

Захищаючись від зовнішнього іонізуючого опромінювання при роботах із закритими джерелами випромінювання, тобто такими, які виключають можливість потрапляння радіоактивних речовин у навколишнє середовище, перш за все необхідно не допустити переопромінення працівників.

Основним способами захисту від цього є:

- зменшення активності джерела, з яким контактують працівники під час конкретного технологічного процесу – досягається шляхом використання речовин із меншою активністю;
- зменшення часу контакту з джерелом випромінювання – досягається шляхом вдосконалення організації робіт і технологічного виробничого процесу та проведення попередніх тренінгів працівників;
- збільшення відстані між людиною і джерелом – використовується, як правило, при контакті з точковим джерелом випромінювання шляхом використання дистанційних універсальних маніпуляторів та інших автоматизованих пристроїв;
- розташування між людиною і джерелом захисного екрану (стаціонарного, пересувного, розбірного, настільного тощо), тобто пристрою, який зменшує інтенсивність випромінювання до безпечного рівня [35].

Для виготовлення екранів, а також для захисту працівників в стаціонарних спорудах, використовується бетон, чавун, сталь, алюміній, скло, свинець та інші матеріали. Від дії рентгенівських променів застосовують екрани зі сталевого листа товщиною 0,5-1 мм або алюмінію товщиною 3 мм, спеціальної гуми. Оглядові вікна виконують з плексигласу товщиною 30 мм або з покритого оловом скла товщиною 9 мм.

Для захисту шкіри від забруднень радіоактивними речовинами та запобігання їх попаданню всередину організму, захисту від альфа і бета-випромінювання передусім застосовуються засоби індивідуального захисту (ЗІЗ) від радіації.

Отже, засоби захисту від радіації використовуються у тих випадках, коли інші заходи недостатньо ефективні: при переході через зони збільшеної

інтенсивності випромінювання, при ремонтних та налагоджувальних роботах у аварійних ситуаціях, під час короткочасного контролю та при зміні інтенсивності опромінення.

З урахуванням зазначеного прогнозу на території області може виникнути складна радіаційна обстановка наслідки якої вимагатимуть від органів виконавчої влади, органів місцевого самоврядування, суб'єктів господарювання, на які покладено виконання завдань щодо захисту населення і територій від надзвичайних ситуацій, оперативного реагування та дій [35].

Місцеві органи виконавчої влади, органи місцевого самоврядування, суб'єкти господарювання здійснюють для забезпечення захисту людей від впливу іонізуючих випромінювань наступні заходи:

- приймають згідно з законодавством України рішення щодо застосування на підвідомчій території заходів втручання у разі радіаційних аварій;
- організовують проведення в установленому порядку щорічні обстеження з метою оцінки стану захисту людини від впливу іонізуючих випромінювань та ведення екологічного паспорта підвідомчої території;
- здійснюють організаційне керівництво системою обліку та контролю доз опромінення населення на підвідомчій території;
- організовують контроль за виконанням заходів щодо захисту людини від впливу радіонуклідів, що містяться у будівельних матеріалах;
- затверджують відповідні плани щодо захисту населення від радіаційних аварій та їх наслідків;
- забезпечують постійну готовність засобів оповіщення населення на підвідомчій території про виникнення радіаційної аварії;
- організовують контроль за виконанням заходів щодо захисту населення від радіаційних аварій та їх наслідків;
- забезпечують населення, в місцях його проживання, інформацією щодо рівнів опромінення людини та заходів захисту від впливу іонізуючих випромінювань, що виконуються на підвідомчій території;

- розроблюють та впроваджують програми захисту людей від впливу іонізуючих випромінювання;
- здійснюють оповіщення населення у разі виникнення радіаційної аварії та інформування про рятувальні та профілактичні заходи у зв’язку з цим.

Для виконання вищезазначених заходів залучаються органи управління, сили і засоби обласної територіальної та функціональних підсистем єдиної державної системи цивільного захисту (далі – ЄДС ЦЗ), порядок дій яких визначено Планом реагування на надзвичайні ситуації, пов'язаних з викидом радіоактивних речовин.

Режими захисту робітників і службовців на суб'єктах господарювання вводяться в дію рішенням керівників об'єктів. Незалежно від місця розміщення суб'єкту господарювання (в населеному пункті або за його межами) на його території вводиться в дію свій режим захисту з урахуванням рівнів радіації, виміряних на об'єкті, і реального ступеню захисту працівників і службовців.

При виникненні комунальної радіаційної аварії окрім термінових робіт щодо стабілізації радіаційного стану (включаючи відновлення контролю над джерелом) місцеві органи виконавчої влади, органи місцевого самоврядування, суб'єкти господарювання одночасно здійснюють заходи, спрямовані на:

- зведення до мінімуму кількості осіб з населення, які зазнають аварійного опромінення;
- запобігання чи зниження індивідуальних і колективних доз опромінення населення;
- запобігання чи зниження рівнів радіоактивного забруднення продуктів харчування, питної води, сільськогосподарської сировини і сільгоспугідь, об'єктів довкілля (повітря, води, ґрунту, рослин тощо), а також будівель і споруд.

Для населення, робітників та службовців суб'єктів господарювання, які можуть потрапити в зону випадіння радіоактивних опадів, доцільно завчасно, виходячи з конкретних місцевих умов, розрахувати варіанти режимів радіаційного захисту [35].

З урахуванням вищезазначеного, режими радіаційного захисту вводяться в дію місцевими органами виконавчої влади, органами місцевого самоврядування, суб'єктами господарювання з метою захисту людей від впливу іонізуючого випромінювання у разі загрози або виникнення надзвичайних ситуацій, пов'язаних з радіаційними аваріями.

4.3 Висновок до четвертого розділу

У четвертому розділі було розглянуто фактори ризику та можливі порушення здоров’я користувачів комп’ютерної мережі, а також питання забезпечення захисту працівників суб’єкта господарювання від іонізуючого випромінювання. Перший підрозділ присвячений аналізу негативного впливу тривалої роботи за комп’ютером на фізичний і психологічний стан людини. Було визначено, що основними ризиками для користувачів є порушення постави, біль у спині, шиї та плечах, перенапруження очей, синдром зап’ястного каналу, зниження фізичної активності, порушення сну, стрес, емоційне виснаження та залежність від Інтернету чи соціальних мереж. Особливу увагу було приділено необхідності правильної організації робочого місця, використанню ергономічних меблів, раціональному розміщенню монітора, клавіатури та миші, а також регулярним перервам для руху й відпочинку.

Другий підрозділ охоплює питання захисту працівників від впливу іонізуючого випромінювання. Було розглянуто основні способи радіаційного захисту, серед яких зменшення активності джерела випромінювання, скорочення часу контакту з ним, збільшення відстані між працівником і джерелом, а також використання захисних екранів і засобів індивідуального захисту. Також було підкреслено важливість медичного нагляду, дозиметричного контролю, своєчасного оповіщення населення та працівників, а також організації заходів цивільного захисту у разі виникнення радіаційної аварії.

Таким чином, четвертий розділ підкреслює важливість комплексного підходу до забезпечення безпеки працівників і користувачів комп’ютерних мереж. Дотримання ергономічних вимог, профілактика фізичного та психологічного перевантаження, правильна організація режиму праці й відпочинку, а також виконання заходів радіаційного захисту дозволяють знизити ризики для здоров’я, підвищити рівень безпеки праці та створити більш комфортні й безпечні умови діяльності на підприємстві.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було реалізовано та протестовано систему управління API-ключами та доступом до сервісів. У роботі послідовно проаналізовано теоретичні засади використання API-ключів як механізму контролю доступу до програмних інтерфейсів, спроєктовано архітектуру системи на основі FastAPI-застосунку, реалізовано основні функціональні модулі та проведено практичне тестування працездатності розробленого рішення.

У першому розділі було досліджено роль API-ключів у сучасних системах захисту API. Встановлено, що API-ключі є простим і зручним механізмом ідентифікації клієнтських застосунків, який може використовуватися для квотування, білінгу, базового обмеження доступу та контролю використання сервісів. Водночас було визначено, що API-ключ сам по собі не є повноцінним засобом автентифікації та авторизації, оскільки не забезпечує гнучкого керування правами доступу в складних або надійних сценаріях. Тому ефективне використання API-ключів потребує поєднання з додатковими механізмами безпеки, зокрема TLS-з’єднанням, обмеженням сфери дії ключа, контролем кількості запитів, журналюванням, аудитом доступу та, за потреби, інтеграцією з OAuth 2.0 або JWT. Це дозволило обґрунтувати необхідність створення не просто механізму генерації ключів, а цілісної системи керування доступом до сервісів.

У другому розділі було виконано проєктування та реалізацію системи управління API-ключами. Розроблена система побудована на основі FastAPI та включає модулі автентифікації користувачів, керування користувачами, сервісами, scope-дозволами, API-ключами, перевірки доступу, rate limiting і журналювання подій. Було спроєктовано базу даних, яка містить таблиці користувачів, сервісів, дозволів, API-ключів, журналу аудиту та обмеження кількості запитів. Така структура забезпечує прив’язку API-ключа до конкретного сервісу, надання йому визначених scope-дозволів і контроль його

використання. У системі реалізовано JWT-автентифікацію для адміністративних користувачів, рольову модель доступу, створення, перегляд, відкликання та ротацію API-ключів. Особливу увагу приділено безпечному зберіганню ключів: відкритий ключ відображається лише один раз під час створення, а в базі даних зберігається тільки його хеш. Це підвищує рівень захисту системи та зменшує ризик компрометації ключів.

У третьому розділі було проведено практичне тестування розробленої системи. Було перевірено запуск серверної частини, роботу endpoint /health, авторизацію адміністратора за допомогою JWT-токена, перегляд доступних сервісів і scope-дозволів, створення API-ключа та його використання для доступу до захищених ресурсів. Під час тестування підтверджено, що система коректно створює API-ключі, прив’язує їх до відповідних сервісів і дозволів, а також виконує перевірку доступу за статусом ключа, терміном дії, відповідністю сервісу, наявністю потрібних scope-дозволів і дотриманням ліміту запитів. Успішні запити з валідним ключем повертали HTTP-код 200, тоді як спроби доступу без ключа, з недостатніми правами або до іншого сервісу завершувалися відповідними помилками 401, 403 або 429. Також було протестовано ротацію та відкликання API-ключів, після чого старі або неактивні ключі втрачали можливість доступу до захищених ресурсів.

Окремо було підтверджено правильність роботи журналу аудиту. У ньому фіксувалися важливі події системи, зокрема створення, відкликання та спроби використання API-ключів, а також відмови в доступі із зазначенням причини, ідентифікатора ключа, IP-адреси та часу створення запису. Це забезпечує можливість подальшого аналізу дій користувачів і клієнтських застосунків, виявлення підозрілої активності та контролю безпечного використання сервісів.

У результаті виконання кваліфікаційної роботи було успішно реалізовано та перевірено на практиці систему управління API-ключами та доступом до сервісів. Розроблене рішення забезпечує централізоване створення, зберігання, ротацію та відкликання API-ключів, розмежування прав доступу за сервісами і

scope-дозволами, обмеження кількості запитів, захист адміністративних дій за допомогою JWT-автентифікації та ведення журналу аудиту. Проведене тестування підтвердило, що система коректно реагує на дозволені й заборонені сценарії доступу, блокує неактивні або відкликані ключі та забезпечує контрольоване використання захищених ресурсів. Запропоноване рішення відповідає сучасним вимогам до керування доступом в API-орієнтованих програмних системах і може бути використане як основа для подальшого розвитку, масштабування та інтеграції з іншими механізмами автентифікації й авторизації.

ПЕРЕЛІК ДЖЕРЕЛ

1. Google Cloud. (n.d.). API keys overview; Best practices for managing API keys; Manage API keys; Access control with IAM; API keys audit logging. Google Cloud Documentation. https://docs.cloud.google.com/api-keys/docs/overview.

2. Amazon Web Services. (n.d.). Usage plans and API keys for REST APIs in API Gateway; Set up API keys for REST APIs in API Gateway; Create and configure API keys and usage plans with CloudFormation. AWS Documentation. https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html.

3. Kharchenko, A., Bodnarchuk, I., & Yatcyshyn, V. (2014). The method for comparative evaluation of software architecture with accounting of trade-offs. American Journal of Information Systems, 2(1), 20-25.

4. OWASP Foundation. (n.d.). REST security cheat sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html.

5. Bodnarchuk, I., Lisovyi, V., Kharchenko, O., & Galai, I. (2018, September). Adaptive method for assessment and selection of software architecture in flexible techniques of design. In 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 292-297). IEEE.

6. National Institute of Standards and Technology. (n.d.). SP 800-63B: Digital identity guidelines: Authentication and authenticator management. NIST. https://pages.nist.gov/800-63-3/sp800-63b.html.

7. OWASP Foundation. (n.d.). Authorization cheat sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html.

8. Ihor, B., Oleksii, D., Aleksandr, K., Nataliia, K., Oleksandr, M., & Volodymyr, P. (2019, January). Multicriteria choice of software architecture using dynamic correction of quality attributes. In International Conference on Computer

Science, Engineering and Education Applications (pp. 419-427). Cham: Springer International Publishing.

9. Kharchenko, A., Raichev, I., Bodnarchuk, I., & Matsiuk, O. (2021, October). The Survey of Global Software Design Processes. In 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) (pp. 291-294). IEEE.

10. Волович, В., Береженко, Б. М., & Боднарчук, І. О. (2022). Задача проєктування програмної архітектури в процесах забезпечення якості. Матеріали X науково-технічної конференції „Інформаційні моделі, системи та технології “Тернопільського національного технічного університету імені Івана Пулюя, 104-106.

11. Боднарчук, І., Харченко, О., Хоміцький, Б., & Шимчук, Г. (2019). Проектування архітектури програмних систем в проектах з гнучкими методами управління. Матеріали XXI наукової конференції Тернопільського національного технічного університету імені Івана Пулюя, 46-48.

12. Hardt, D. (2012). The OAuth 2.0 authorization framework (RFC 6749). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc6749.

13. Jones, M., Bradley, J., & Sakimura, N. (2015). *JSON Web Token (JWT)* (RFC 7519). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc7519.

14. Google Cloud Apigee. (n.d.). API keys. Google Cloud Apigee Documentation. https://docs.cloud.google.com/apigee/docs/api-platform/security/api-keys.

15. Orobchuk, B., Buniak, O., Sysak, I., Babiuk, S., Bodnarchuk, I., & Koval, V. (2024). Development of Software for the Implementation of Automated Reserve Input Modes Operation. In CITI (pp. 316-336).

16. Microsoft. (n.d.). Access control lists. Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-lists.

17. Kharchenko, A., Halay, I., Zagorodna, N., & Bodnarchuk, I. (2015, September). Trade-off optimal decision of the problem of software system

architecture choice. In 2015 Xth International Scientific and Technical Conference" Computer Sciences and Information Technologies"(CSIT) (pp. 198-205). IEEE.

18. National Institute of Standards and Technology. (n.d.). Role based access control; RBAC glossary entry. NIST. https://csrc.nist.gov/projects/role-based-access-control.

19. Lodderstedt, T., Bradley, J., Labunets, A., & Fett, D. (2025). Best current practice for OAuth 2.0 security (RFC 9700). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc9700.

20. Kent, K., & Souppaya, M. (2006). Guide to computer security log management (NIST SP 800-92). National Institute of Standards and Technology. https://csrc.nist.gov/pubs/sp/800/92/final.

21. Google Cloud. (n.d.). API keys overview. Google Cloud Documentation. https://docs.cloud.google.com/api-keys/docs/overview.

22. Google Cloud. (n.d.). Best practices for managing API keys. Google Cloud Documentation. https://docs.cloud.google.com/docs/authentication/api-keys-best-practices.

23. Amazon Web Services. (n.d.). Usage plans and API keys for REST APIs in API Gateway. AWS Documentation. https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html.

24. Google Cloud Apigee. (n.d.). VerifyAPIKey policy. Apigee Edge Documentation. https://docs.cloud.google.com/apigee/docs/api-platform/reference/policies/verify-api-key-policy.

25. Google Cloud Apigee. (n.d.). Introduction to API products. Apigee Documentation. https://docs.cloud.google.com/apigee/docs/api-platform/publish/what-api-product.

26. Hardt, D. (2012). The OAuth 2.0 authorization framework (RFC 6749). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc6749.

27. Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc7519.

28. OWASP Foundation. (2023). OWASP Top 10 API Security Risks – 2023. OWASP API Security Project. https://owasp.org/API-Security/editions/2023/en/0x00-header/.

29. FastAPI. (n.d.). FastAPI documentation. https://fastapi.tiangolo.com.

30. FastAPI. (n.d.). OAuth2 with password and bearer with JWT tokens. https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/.

31. Python Software Foundation. (n.d.). hmac — Keyed-hashing for message authentication. Python Documentation. https://docs.python.org/3/library/hmac.html.

32. SQLAlchemy. (n.d.). SQLAlchemy ORM documentation. https://docs.sqlalchemy.org/orm/.

33. OWASP Foundation. (2023). API4:2023 unrestricted resource consumption. OWASP API Security Project. https://owasp.org/API-Security/editions/2023/en/0xa4-unrestricted-resource-consumption/.

34. Шкідливий вплив персонального комп’ютера на здоров’я людини. URL: https://ir.lib.vntu.edu.ua/bitstream/handle/123456789/21370/5363.pdf

35. Желібо Є. П., Сагайдак І. С. Безпека життєдіяльності. Навчальний посібник для аудиторної та практичної роботи. К.:ЕКОМЕН. 2011. 200 с.