

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ СЕРВЕРНОГО ПРОГРАМНОГО
ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ
DEVELOPMENT OF SERVER-SIDE SOFTWARE FOR
USER AUTHENTICATION

Виконав(ла): студент(ка) 4 курсу, групи КНТ-132

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ГОСТЄВ І.І.

(ПРИЗВИЩЕ та ініціали)

Керівник ШИТІКОВА О.В.

(ПРИЗВИЩЕ та ініціали)

Рецензент КОРОТКИЙ О.В.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ГОСТЄВА Івана Ілліча

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розроблення серверного програмного забезпечення для автентифікації користувачів. Development of Server-Side Software for User Authentication

керівник проєкту (роботи) к.т.н., ШИТІКОВА Олена Вікторівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від 07 квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області, існуючих рішень та постановка задачі. 2. Вибір технологій. 3. Проєктування серверного програмного застосунку автентифікації користувачів. 4. Реалізація серверного застосунку автентифікації користувачів.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ШИТІКОВА О.В., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 07 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження	5 тиждень	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Іван ГОСТЄВ
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Олена ШИТІКОВА
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 153 с., 8 табл., 38 рис., 3 дод., 11 джерел.

АВТЕНТИФІКАЦІЯ КОРИСТУВАЧІВ, СЕРВЕРНИЙ ЗАСТОСУНОК, БЕЗПЕКА, ДВОФАКТОРНА АВТЕНТИФІКАЦІЯ, JWT, SPRING BOOT.

Об'єкт дослідження – процес інженерії програмного забезпечення у сфері організації безпеки користувацьких даних.

Предмет дослідження – серверне програмне забезпечення для автентифікації та авторизації користувачів.

Мета роботи – зменшення ризику несанкціонованого доступу до інформаційних систем та скорочення часу автентифікації користувачів шляхом розроблення серверного програмного забезпечення із використанням JWT та двофакторної автентифікації.

Матеріали, методи та технічні засоби: об'єктно-орієнтоване програмування, фреймворк Spring Boot, використання Spring Security, технологія JWT для авторизації, база даних PostgreSQL, мова програмування Java, середовище розробки IntelliJ IDEA.

Результати. Розроблено серверний застосунок для автентифікації користувачів, який підтримує реєстрацію, авторизацію, використання JWT-токенів та двофакторну автентифікацію за допомогою Google Authenticator.

Висновки. Розроблена система забезпечує підвищений рівень захисту користувацьких даних, знижує ризик несанкціонованого доступу та може бути використана як основа для побудови безпечних вебзастосунків.

Галузь використання – інформаційні системи, вебзастосунки, корпоративні сервіси, де необхідна надійна автентифікація користувачів.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 153 pages, 8 tables, 38 figures, 3 appendixes, 11 sources.

USER AUTHENTICATION, SERVER APPLICATION, SECURITY, TWO-FACTOR AUTHENTICATION, JWT, SPRING BOOT.

The object of this study is the software engineering process in the field of user data security.

The subject of this study is server-side software for user authentication and authorization.

The objective of this work is to reduce the risk of unauthorized access to information systems and shorten user authentication time by developing server-side software using JWT and two-factor authentication.

Materials, methods, and technical tools: object-oriented programming, the Spring Boot framework, Spring Security, JWT technology for authorization, the PostgreSQL database, the Java programming language, and the IntelliJ IDEA development environment.

Results. A server-side application for user authentication has been developed, supporting registration, authorization, the use of JWT tokens, and two-factor authentication via Google Authenticator.

Conclusions. The developed system provides an enhanced level of user data protection, reduces the risk of unauthorized access, and can be used as a foundation for building secure web applications.

Fields of application: information systems, web applications, and corporate services where reliable user authentication is required.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
Вступ.....	9
1 Аналіз предметної області, існуючих рішень та постановка задачі	11
1.1 Основи автентифікації користувачів	11
1.2 Сучасні підходи до організації безпеки.....	13
1.3 Аналіз існуючих програмних рішень.....	16
1.4 Висновки з аналізу предметної області	20
1.5 Постановка задачі.....	23
2 Вибір технологій	25
2.1 Обґрунтування вибору мови програмування	25
2.2 Вибір серверного фреймворку	26
2.3 Вибір засобів організації безпеки.....	28
2.4 Вибір механізму авторизації користувачів.....	31
2.5 Вибір механізму двофакторної автентифікації.....	34
2.6 Вибір системи керування базами даних	36
2.7 Вибір середовища розробки та допоміжних інструментів	39
2.8 Висновки до розділу	41
3 Проєктування серверного програмного застосунку автентифікації користувачів.....	43
3.1 Проєктування архітектури системи	43
3.2 Проєктування моделі даних	47
3.3 Проєктування механізму автентифікації	51
3.4 Проєктування JWT-авторизації	58

3.5	Проектування двофакторної автентифікації	63
3.6	Проектування REST API	69
3.7	Висновки до розділу	73
4	Реалізація серверного застосунку автентифікації користувачів	75
4.1	Реалізація серверної архітектури.....	75
4.2	Реалізація моделі даних.....	79
4.3	Реалізація механізму автентифікації.....	83
4.4	Реалізація JWT-авторизації	87
4.5	Реалізація двофакторної автентифікації	90
4.6	Реалізація системи ролей та контролю доступу	95
4.7	Реалізація системи підтримки користувачів	99
4.8	Тестування REST API через Swagger UI	103
4.9	Висновки до розділу	110
	Висновки	115
	Перелік джерел посилання	114
	Додаток А Технічне завдання	115
	Додаток Б Текст програми	122
	Додаток В Слайди презентації.....	146

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

2FA	– Two-Factor Authentication;
API	– Application Programming Interface;
DTO	– Data Transfer Object;
HTTP	– HyperText Transfer Protocol;
HTTPS	– HyperText Transfer Protocol Secure;
IDE	– Integrated Development Environment;
JSON	– JavaScript Object Notation;
JPA	– Java Persistence API;
JWT	– JSON Web Token;
MFA	– Multi-Factor Authentication;
ORM	– Object-Relational Mapping;
REST	– Representational State Transfer;
SQL	– Structured Query Language;
TOTP	– Time-based One-Time Password;
UUID	– Universally Unique Identifier;
БД	– база даних;
ІС	– інформаційна система;
ПЗ	– програмне забезпечення.

ВСТУП

Сьогодні проблема забезпечення захисту даних користувачів стала вкрай актуальною в стрімко мінливому світі, де інформаційні технології розвиваються у швидкому темпі. Ця вимога є необхідною для більшості сучасних вебзастосунків і інформаційних систем, оскільки нині для таких середовищ потрібні звіти про стан автентифікації користувачів та механізми авторизації. Недостатній захист облікових записів користувачів може розкрити чутливі дані, надати можливість для вторгнень та створити багато інших форм атак.

Традиційні методи автентифікації лише за паролем не здатні забезпечити належний рівень безпеки. Існують різні способи атак, що використовуються для експлуатації механізму автентифікації з метою отримання несанкціонованого доступу, зокрема вгадування паролів, фішингові атаки або використання викрадених облікових даних. У цьому випадку актуальними є додаткові рівні захисту, зокрема двостороння автентифікація, яка значно підвищує рівень безпеки систем.

Сьогодні розробники використовують фреймворки на кшталт Spring Boot, щоб швидко створювати серверні застосунки, які інтегруються з сучасними технологіями безпеки. Spring Security [1] переважно застосовується для впровадження механізмів безпеки, що забезпечує повний рівень реалізації механізмів автентифікації та рівень авторизації для користувачів, організовані належним чином.

Ще один рівень безпеки досягається через використання access-токенів, зокрема технології JWT [2] (JSON Web Token), яка добре підходить для впровадження безстанної моделі авторизації. Мобільні застосунки на кшталт Google Authenticator [3], які генерують одноразові коди підтвердження для двофакторної автентифікації, також відіграють значну роль.

Об'єктом дослідження є процес інженерії програмного забезпечення у сфері організації безпеки інформаційних систем.

Предметом дослідження є серверне програмне забезпечення для автентифікації та авторизації користувачів.

Метою роботи є зменшення ризику несанкціонованого доступу до інформаційних систем та підвищення ефективності автентифікації користувачів шляхом використання токен-орієнтованої авторизації (JWT [2]) та двофакторної автентифікації.

Для досягнення поставленої мети необхідно вирішити такі задачі:

- провести аналіз предметної області та існуючих рішень у сфері автентифікації користувачів;
- обґрунтувати вибір засобів розробки та технологій;
- спроектувати архітектуру серверного застосунку;
- реалізувати механізми реєстрації, автентифікації та авторизації користувачів;
- провести тестування розробленого програмного забезпечення.

Практичне значення роботи полягає у створенні серверного застосунку, який може бути використаний як основа для побудови безпечних вебзастосунків та інформаційних систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ІСНУЮЧИХ РІШЕНЬ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Основи автентифікації користувачів

1.1.1 Поняття автентифікації та авторизації

У сучасних інформаційних системах важливу роль відіграють процеси автентифікації та авторизації користувачів. Автентифікація – це процес перевірки особи користувача [1] з метою підтвердження його автентичності, тобто відповідності заявленим обліковим даним. Найчастіше вона здійснюється шляхом введення логіна та пароля, однак сучасні системи використовують і більш складні механізми.

Авторизація, у свою чергу, є наступним етапом після успішної автентифікації та визначає рівень доступу користувача [1] до ресурсів системи. Вона відповідає на питання, які саме дії дозволено виконувати користувачу, наприклад, перегляд даних, їх редагування або адміністрування системи.

Таким чином, автентифікація відповідає за встановлення особи користувача, тоді як авторизація – за визначення його прав доступу. Обидва процеси є невід’ємними складовими систем інформаційної безпеки [2] та повинні реалізовуватися у взаємозв’язку.

1.1.2 Основні методи автентифікації

Існує декілька основних методів автентифікації користувачів, які застосовуються в сучасних інформаційних системах. Найпоширенішим є парольний метод, що базується на використанні комбінації логіна та пароля. Незважаючи на свою простоту та зручність, цей метод має ряд обмежень, пов’язаних із безпекою.

Іншим підходом є використання токенів доступу, зокрема JSON Web Token (JWT) [2], які дозволяють реалізувати безстанну модель авторизації. У

цьому випадку після успішної автентифікації користувач отримує токен, який використовується для доступу до захищених ресурсів без необхідності повторного введення облікових даних.

Біометрична автентифікація передбачає використання унікальних фізичних характеристик користувача, таких як відбитки пальців, розпізнавання обличчя або голосу. Цей метод забезпечує високий рівень безпеки, однак потребує спеціалізованого обладнання.

Також широко застосовується багатофакторна автентифікація, яка поєднує декілька незалежних факторів перевірки особи, що значно підвищує рівень захисту системи.

1.1.3 Вразливості парольної автентифікації

Незважаючи на широке використання, парольна автентифікація має значні недоліки з точки зору безпеки. Однією з основних проблем є використання користувачами слабких або повторюваних паролів, що значно спрощує їх підбір зловмисниками.

Серед поширених загроз можна виділити атаки перебору (brute-force), при яких здійснюється послідовна перевірка великої кількості можливих комбінацій паролів. Також небезпечними є атаки з використанням попередньо згенерованих таблиць (rainbow tables), що дозволяють швидко знаходити відповідності між хешами та паролями.

Окрім цього, існує загроза фішингу, при якій користувачі вводять свої облікові дані на підроблених вебресурсах. У випадку витоку бази даних паролів зловмисники можуть отримати доступ до облікових записів, особливо якщо паролі не були належним чином захищені.

Таким чином, використання лише паролів не забезпечує достатнього рівня безпеки, що обумовлює необхідність застосування додаткових механізмів захисту.

1.1.4 Багатофакторна автентифікація та її переваги

Багатофакторна автентифікація є сучасним підходом до зменшення ймовірності несанкціонованого доступу до облікових записів користувачів інформаційних систем. Вона передбачає використання двох або більше незалежних факторів для підтвердження особи користувача. Як правило, це поєднання знання (пароль), володіння (мобільний пристрій) та, у деяких випадках, біометричних характеристик.

Найбільш поширеною формою є двофакторна автентифікація [4] (2FA), яка доповнює пароль одноразовим кодом підтвердження. Такі коди можуть генеруватися мобільними додатками, наприклад Google Authenticator [3], або надсилатися через SMS чи електронну пошту.

Основною перевагою багатофакторної автентифікації є значне підвищення рівня захисту. Навіть у разі компрометації пароля зловмисник не зможе отримати доступ до облікового запису без другого фактору. Це робить систему значно стійкішою до більшості сучасних атак.

Крім того, впровадження багатофакторної автентифікації є важливим елементом забезпечення відповідності сучасним стандартам інформаційної безпеки та рекомендаціям щодо захисту даних користувачів.

1.2 Сучасні підходи до організації безпеки

1.2.1 Токен-орієнтована автентифікація (JWT)

Одним із сучасних підходів до організації безпеки інформаційних систем є використання токен-орієнтованої автентифікації. На відміну від традиційної сесійної моделі, у якій сервер зберігає інформацію про кожного користувача, токен-орієнтований підхід передбачає передачу спеціального маркера доступу між клієнтом і сервером.

Найбільш поширеним стандартом у цьому контексті є JSON Web Token (JWT) [2]. Даний механізм дозволяє створювати компактні та

самодостатні токени, що містять інформацію про користувача та його права доступу. Токен підписується за допомогою криптографічного алгоритму, що гарантує його цілісність і захищеність від підробки.

Після успішної автентифікації сервер генерує токен і передає його клієнту. У подальшому клієнт надсилає цей токен разом із кожним запитом, що дозволяє серверу перевірити автентичність користувача без збереження стану сесії. Такий підхід підвищує масштабованість системи та зменшує навантаження на сервер.

До переваг використання JWT [2] можна віднести відсутність необхідності зберігання сесій, простоту інтеграції та можливість використання у розподілених системах. Водночас необхідно враховувати ризики, пов'язані з витоком токена, що потребує застосування додаткових заходів безпеки, таких як обмеження часу дії токена.

1.2.2 Двофакторна автентифікація (2FA)

Двофакторна автентифікація є ефективним методом підвищення рівня захисту облікових записів користувачів. Вона передбачає використання двох незалежних факторів для підтвердження особи: як правило, це пароль та додатковий код підтвердження.

Основною ідеєю 2FA є те, що навіть у випадку компрометації одного з факторів (наприклад, пароля), зломисник не зможе отримати доступ до системи без другого фактору. Це значно знижує ризик несанкціонованого доступу.

Двофакторна автентифікація може реалізовуватися різними способами: за допомогою SMS-повідомлень, електронної пошти або спеціалізованих мобільних додатків. Найбільш надійним і поширеним варіантом є використання додатків-генераторів одноразових кодів.

Впровадження 2FA є важливим кроком у побудові сучасних систем безпеки та рекомендується до використання у всіх критично важливих інформаційних системах.

1.2.3 Використання одноразових паролів (TOTP)

Одноразові паролі (One-Time Password, OTP) є одним із ключових елементів двофакторної автентифікації. Особливу популярність отримав алгоритм TOTP [5] (Time-based One-Time Password), який генерує унікальні коди на основі секретного ключа та поточного часу.

Суть методу полягає у тому, що сервер і клієнт мають спільний секретний ключ, який використовується для генерації коду. Цей код змінюється через певні інтервали часу, зазвичай кожні 30 секунд, що значно ускладнює його перехоплення або повторне використання.

Для реалізації TOTP [5] широко застосовуються мобільні додатки, такі як Google Authenticator [3], які дозволяють користувачам швидко та зручно отримувати коди підтвердження.

Перевагами даного підходу є висока безпека, незалежність від мережевого з'єднання та простота використання. Водночас необхідно забезпечити надійне зберігання секретного ключа на стороні сервера.

1.2.4 Підходи до захисту паролів (хешування, salt)

Захист паролів користувачів є одним із найважливіших аспектів організації безпеки системи. Зберігання паролів у відкритому вигляді є неприпустимим, оскільки у випадку витоку бази даних це призведе до компрометації облікових записів.

Сучасні системи використовують хешування паролів – процес перетворення пароля у фіксований рядок символів за допомогою

спеціального алгоритму. При цьому відновити початковий пароль із хешу практично неможливо.

Для підвищення рівня захисту застосовується додатковий механізм – salt (сіль), який додається до пароля перед хешуванням. Це дозволяє уникнути використання попередньо згенерованих таблиць (rainbow tables) та робить кожен хеш унікальним навіть для однакових паролів.

Крім того, у сучасних системах використовуються спеціалізовані алгоритми хешування, такі як bcrypt, які є обчислювально складними та стійкими до атак перебору. Вони дозволяють регулювати складність обчислень, що забезпечує додатковий рівень захисту.

Таким чином, правильна реалізація механізмів зберігання паролів є необхідною умовою організації безпеки інформаційної системи.

1.3 Аналіз існуючих програмних рішень

Для визначення вимог до власного серверного програмного забезпечення доцільно проаналізувати існуючі рішення, які використовуються для автентифікації користувачів, керування обліковими записами та підвищення рівня безпеки входу до системи. У межах даної роботи розглянуто такі програмні засоби: Auth0 [6], Firebase Authentication [7] та Google Authenticator [3].

1.3.1 Аналіз Auth0 [6]

Auth0 [6] є платформою керування ідентифікацією користувачів, яка надає засоби автентифікації, авторизації та інтеграції з різними постачальниками облікових записів. Система підтримує вхід за допомогою електронної пошти та пароля, соціальних мереж, корпоративних провайдерів, а також багатфакторну автентифікацію. Auth0 [6] також надає засоби роботи

з токенами, зокрема JWT [2], що дозволяє використовувати платформу для захисту API та побудови сучасних вебзастосунків.

Перевагою Auth0 [6] є наявність готової інфраструктури для керування користувачами, ролями, правами доступу та додатковими факторами автентифікації. Це дозволяє швидко інтегрувати систему входу до застосунку без необхідності розробляти всі механізми самостійно. Також важливою перевагою є підтримка MFA, яка дає змогу зменшити ризик несанкціонованого доступу навіть у випадку компрометації пароля.

Недоліком Auth0 [6] є залежність від стороннього сервісу. Для навчального або дипломного проєкту це означає, що значна частина логіки автентифікації реалізується не студентом, а зовнішньою платформою. Крім того, використання такого рішення може обмежувати гнучкість налаштування внутрішньої архітектури системи. Саме тому в межах даної роботи доцільніше реалізувати власний серверний застосунок, зберігши ідеї, які використовуються в Auth0 [6]: токен-орієнтовану авторизацію, підтримку ролей та можливість використання багатфакторної автентифікації.

1.3.2 Аналіз Firebase Authentication

Firebase Authentication [7] – це сервіс автентифікації від Google, який дозволяє швидко додати до застосунку реєстрацію, вхід, відновлення пароля та інтеграцію з різними провайдерами входу. Firebase Authentication [7] підтримує автентифікацію за електронною поштою та паролем, вхід через Google, Facebook, GitHub, Apple, а також телефонну автентифікацію. Крім того, сервіс має готовий компонент Firebase UI [7], який може використовуватись як готове рішення для сторінок входу та реєстрації.

Перевагою Firebase Authentication [7] є простота інтеграції. Розробник може швидко підключити готові SDK та використати вже реалізовані механізми керування користувачами. Також Firebase [7] підтримує багатфакторну автентифікацію, зокрема SMS MFA, а в межах Firebase

Authentication [7] with Identity Platform можливе використання TOTP [5] як додаткового фактора.

Недоліком Firebase Authentication [7] є сильна прив'язка до екосистеми Google. Для невеликих застосунків це може бути перевагою, однак для власного серверного програмного забезпечення така залежність зменшує контроль над архітектурою, базою даних та внутрішньою логікою автентифікації. Також частина розширених можливостей може залежати від використання додаткових сервісів Google Cloud. У власній роботі доцільно використати загальну ідею Firebase Authentication [7] – простий процес реєстрації та входу користувача, але реалізувати його на власному сервері з можливістю повного контролю над даними та логікою безпеки.

1.3.3 Аналіз Google Authenticator

Google Authenticator [3] – це мобільний додаток, який використовується для генерації одноразових кодів підтвердження. Він підтримує TOTP-коди [5], які зазвичай змінюються кожні 30 секунд, а також HOTP-коди. Додаток дозволяє додавати облікові записи шляхом сканування QR-коду або ручного введення секретного ключа.

Основною перевагою Google Authenticator [3] є простота використання та відсутність необхідності постійного підключення до мережі для генерації кодів. Після прив'язки облікового запису додаток може генерувати коди локально на пристрої користувача. Це робить такий підхід зручним для реалізації двофакторної автентифікації у серверному застосунку.

Недоліком Google Authenticator [3] є те, що він не є повноцінною системою автентифікації, а виконує лише роль генератора одноразових кодів. Тобто сам додаток не реєструє користувачів, не перевіряє паролі, не зберігає ролі та не керує доступом до API. У межах розроблюваного програмного забезпечення Google Authenticator [3] доцільно використовувати як допоміжний інструмент для реалізації другого фактора автентифікації, тоді

як основна логіка реєстрації, входу, перевірки пароля, генерації JWT [2] та керування користувачами має виконуватись серверною частиною застосунку.

1.3.4 Порівняльний аналіз існуючих рішень

Проведений аналіз, наведений у таблиці 1.1 показав, що Auth0 [6] та Firebase Authentication [7] є потужними готовими сервісами для реалізації автентифікації користувачів. Вони мають широкий набір можливостей, однак значною мірою залежать від зовнішньої інфраструктури. Google Authenticator [3], у свою чергу, не є самостійною системою автентифікації, але є доречним інструментом для реалізації другого фактора захисту.

Отже, у власній роботі доцільно реалізувати серверне програмне забезпечення, яке поєднує переваги розглянутих рішень: реєстрацію та вхід користувачів, захист паролів за допомогою хешування, авторизацію на основі JWT [2], використання ролей, а також підтримку двофакторної автентифікації через TOTP-коди [5]. При цьому власна реалізація дозволить забезпечити повний контроль над архітектурою, базою даних, логікою обробки запитів та механізмами безпеки.

Таблиця 1.1 – Порівняння автентифікаторів

Критерій	Auth0	Firebase Authentication	Google Authenticator
1	2	3	4
Реєстрація користувачів	Так	Так	Ні
Вхід за email і паролем	Так	Так	Ні
JWT / токени доступу	Так	Так	Ні
Двофакторна автентифікація	Так	Так	Так, як генератор кодів

Продовження таблиці 1.1

1	2	3	4
TOTP	Так / через MFA-механізми	Так у Firebase Authentication with Identity Platform	Так
Контроль над базою даних	Обмежений	Обмежений	Немає БД
Контроль над логікою безпеки	Частковий	Частковий	Немає серверної логіки
Залежність від стороннього сервісу	Висока	Висока	Часткова
Простота інтеграції	Висока	Висока	Висока
Доцільність для дипломної роботи	Як аналог	Як аналог	Як частина 2FA

1.4 Висновки з аналізу предметної області

1.4.1 Виявлені недоліки існуючих рішень

Проведений аналіз сучасних програмних рішень у сфері автентифікації користувачів показав, що, незважаючи на високий рівень функціональності та безпеки, більшість із них мають ряд недоліків.

По-перше, такі системи, як Auth0 [6] та Firebase Authentication [7], характеризуються значною залежністю від сторонньої інфраструктури. Це обмежує можливості розробника щодо повного контролю над обробкою даних, логікою автентифікації та внутрішніми механізмами безпеки.

По-друге, існуючі сервіси часто мають обмежену гнучкість налаштування. У складних або нестандартних проєктах виникає необхідність реалізації специфічної бізнес-логіки, яка не завжди підтримується готовими платформами або потребує використання додаткових платних функцій.

По-третє, використання сторонніх сервісів може створювати потенційні ризики для безпеки, пов'язані з передачею конфіденційних даних третім сторонам. Це особливо важливо для систем, що працюють із чутливою інформацією.

Крім того, такі інструменти, як Google Authenticator [3], хоча й забезпечують ефективний механізм двофакторної автентифікації, не є повноцінними рішеннями для керування користувачами та потребують інтеграції із серверною частиною застосунку.

Таким чином, існуючі рішення не завжди задовольняють вимоги щодо повного контролю, гнучкості та незалежності, що є важливими для розроблення власного програмного забезпечення.

1.4.2 Обґрунтування необхідності розроблення власного застосунку

З огляду на виявлені недоліки існуючих рішень, доцільним є розроблення власного серверного програмного забезпечення для автентифікації користувачів.

Основною причиною є необхідність забезпечення повного контролю над архітектурою системи, процесами обробки даних та механізмами безпеки. Власна реалізація дозволяє адаптувати систему до конкретних вимог проєкту, впроваджувати індивідуальні алгоритми захисту та оптимізувати продуктивність.

Крім того, розроблення власного застосунку забезпечує незалежність від сторонніх сервісів, що є важливим фактором у контексті довготривалого використання системи та збереження конфіденційності даних.

Важливим аспектом є також можливість інтеграції сучасних підходів до безпеки, таких як токен-орієнтована авторизація на основі JWT [2] та двофакторна автентифікація [4] із використанням TOTP [5]. Це дозволяє створити гнучку та масштабовану систему, яка відповідає сучасним вимогам до інформаційної безпеки.

Отже, розроблення власного серверного застосунку є обґрунтованим рішенням, яке дозволяє поєднати переваги існуючих технологій із можливістю їх адаптації до конкретних умов використання.

1.4.3 Визначення вимог до системи

На основі проведеного аналізу предметної області та існуючих рішень сформульовано основні вимоги до розроблюваного програмного забезпечення.

До функціональних вимог належать:

- забезпечення реєстрації користувачів із збереженням облікових даних;
- реалізація автентифікації користувачів за допомогою електронної пошти та пароля;
- підтримка авторизації на основі токенів (JWT);
- впровадження двофакторної автентифікації з використанням одноразових кодів;
- можливість керування ролями користувачів;
- забезпечення доступу до захищених ресурсів лише для авторизованих користувачів.

До нефункціональних вимог належать:

- зменшення ймовірності несанкціонованого доступу до облікових записів користувачів – зберігання паролів (використання хешування та salt);
- масштабованість системи та можливість обробки великої кількості запитів;
- надійність і стабільність роботи серверного застосунку;
- зручність інтеграції з клієнтськими застосунками;
- мінімізація залежності від сторонніх сервісів.

Також важливою вимогою є можливість подальшого розширення функціональності системи, зокрема додавання нових методів автентифікації або інтеграції з іншими сервісами.

Таким чином, сформульовані вимоги визначають основні напрямки розроблення серверного застосунку та є основою для подальшого проєктування системи.

1.5 Постановка задачі

1.5.1 Мета роботи

Метою даної дипломної роботи є розроблення серверного програмного забезпечення для автентифікації користувачів із використанням сучасних методів організації безпеки, зокрема токен-орієнтованої авторизації на основі JWT [2] та двофакторної автентифікації.

Розроблювана система повинна забезпечувати надійний захист облікових записів користувачів, мінімізувати ризики несанкціонованого доступу та відповідати сучасним вимогам до інформаційної безпеки.

1.5.2 Основні задачі дослідження

Для досягнення поставленої мети необхідно вирішити такі задачі:

- провести аналіз предметної області автентифікації користувачів та сучасних підходів до організації безпеки;
- дослідити існуючі програмні рішення у сфері автентифікації та визначити їх переваги і недоліки;
- обґрунтувати доцільність розроблення власного серверного застосунку;
- визначити функціональні та нефункціональні вимоги до системи;
- обрати засоби та середовище розробки;
- спроектувати архітектуру серверного застосунку;
- реалізувати механізми реєстрації та автентифікації користувачів;
- реалізувати авторизацію на основі JWT;
- впровадити двофакторну автентифікацію з використанням одноразових кодів;
- забезпечити захист паролів користувачів за допомогою сучасних методів хешування;
- провести тестування розробленого програмного забезпечення;

– проаналізувати результати роботи та оцінити ефективність реалізованої системи.

2 ВИБІР ТЕХНОЛОГІЙ

2.1 Обґрунтування вибору мови програмування

2.1.1 Загальні вимоги до мови програмування

Вибір мови програмування є одним із ключових етапів розроблення серверного програмного забезпечення. Обрана мова повинна забезпечувати високу продуктивність, надійність, масштабованість та безпеку, що є критично важливим для систем автентифікації користувачів.

Крім того, важливими критеріями є наявність розвиненої екосистеми, підтримка сучасних технологій безпеки, а також можливість інтеграції з базами даних та вебтехнологіями.

2.1.2 Аналіз сучасних мов програмування

Для розроблення серверного застосунку було розглянуто кілька популярних мов програмування, а саме: Java, C# та JavaScript (Node.js).

Java є однією з найбільш поширених мов програмування для створення серверних застосунків. Вона відзначається високим рівнем стабільності, безпеки та масштабованості. Наявність великої кількості бібліотек та фреймворків, зокрема Spring Boot, дозволяє значно прискорити процес розробки.

C# у поєднанні з платформою .NET забезпечує високу продуктивність та має добре розвинену екосистему. Однак його використання часто пов'язане з технологіями Microsoft, що може обмежувати гнучкість вибору середовища розгортання.

JavaScript (Node.js) дозволяє реалізувати як клієнтську, так і серверну частину застосунку однією мовою. Проте через асинхронну природу та меншу структурованість коду він є менш придатним для реалізації складних систем безпеки.

Порівняльний аналіз мов програмування наведено у таблиці 2.1.

Таблиця 2.1 – Порівняння мов програмування для серверної розробки

Критерій	Java	C# (.NET)	Node.js
Продуктивність	Висока	Висока	Середня
Безпека	Висока	Висока	Середня
Масштабованість	Висока	Висока	Висока
Екосистема	Дуже широка	Широка	Широка
Портативність	Висока	Середня	Висока
Зручність підтримки	Висока	Висока	Середня

2.1.3 Висновки щодо вибору мови програмування

На основі проведеного аналізу було обрано мову програмування Java, оскільки вона забезпечує високий рівень безпеки, стабільності та масштабованості, що є критично важливим для серверного програмного забезпечення автентифікації користувачів. Крім того, Java дозволяє ефективно інтегрувати сучасні технології, такі як JWT [2] та двофакторна автентифікація [4].

2.2 Вибір серверного фреймворку

2.2.1 Вимоги до серверного фреймворку

Серверний фреймворк повинен забезпечувати швидку розробку застосунку, підтримку REST-архітектури, інтеграцію з базами даних, а також можливість реалізації механізмів безпеки.

2.2.2 Аналіз популярних фреймворків

Було розглянуто такі фреймворки: Spring Boot, ASP.NET Core та Express.js.

Spring Boot дозволяє швидко створювати серверні застосунки, має вбудовану підтримку безпеки через Spring Security [1] та інтеграцію з базами даних.

ASP.NET Core забезпечує високу продуктивність, однак має часткову залежність від екосистеми Microsoft.

Express.js є легким фреймворком, але не має вбудованих механізмів безпеки.

Порівняльний аналіз фреймворків наведено у таблиці 2.2.

Таблиця 2.2 – Порівняння серверних фреймворків

Критерій	Spring Boot	ASP.NET Core	Express.js
1	2	3	4
Продуктивність	Висока	Висока	Середня
Вбудована безпека	Так (Spring Security)	Так	Ні
Масштабованість	Висока	Висока	Висока
Критерій	Spring Boot	ASP.NET Core	Express.js
Простота налаштування	Висока	Середня	Висока
Інтеграція з БД	Вбудована (JPA)	Вбудована	Через бібліотеки

Продовження таблиці 2.2

1	2	3	4
Простота налаштування	Висока	Середня	Висока
Інтеграція з БД	Вбудована (JPA)	Вбудована	Через бібліотеки
Підтримка JWT	Так	Так	Так

2.2.3 Висновки щодо вибору фреймворку

Було обрано Spring Boot, оскільки він забезпечує швидку розробку, має потужні механізми безпеки та добре інтегрується з іншими компонентами системи.

2.3 Вибір засобів організації безпеки

2.3.1 Основні вимоги до організації безпеки системи

У процесі розроблення серверного програмного забезпечення для автентифікації користувачів одним із ключових аспектів є забезпечення належного рівня безпеки. Система повинна гарантувати захист облікових даних користувачів, запобігати несанкціонованому доступу та відповідати сучасним вимогам інформаційної безпеки.

До основних вимог, що висуваються до засобів організації безпеки, належать:

- підтримка автентифікації та авторизації користувачів;
- можливість реалізації токен-орієнтованої моделі доступу;
- підтримка багатофакторної автентифікації;
- захист паролів користувачів (хешування, використання salt);
- можливість обмеження доступу до ресурсів системи;

- гнучкість налаштування правил безпеки;
- відсутність залежності від сторонніх сервісів.

З урахуванням цих вимог було проведено аналіз існуючих рішень для реалізації безпеки.

2.3.2 Аналіз існуючих засобів організації безпеки

Для реалізації механізмів безпеки у серверному застосунку було розглянуто такі рішення: Spring Security [1], Keycloak та Auth0 [6].

Spring Security [1] є модулем безпеки для фреймворку Spring Boot, який забезпечує комплексний підхід до реалізації автентифікації та авторизації. Він дозволяє налаштовувати правила доступу до ресурсів, реалізовувати фільтрацію HTTP-запитів, інтегрувати токен-орієнтовану авторизацію та використовувати різні механізми перевірки користувачів. Завдяки гнучкій конфігурації розробник має повний контроль над логікою безпеки.

Keycloak є окремою системою керування ідентифікацією та доступом (Identity and Access Management). Вона надає готові механізми автентифікації, авторизації, підтримку соціальних входів та багатофакторної автентифікації. Однак використання Keycloak потребує розгортання окремого сервера, що ускладнює архітектуру системи та збільшує складність її супроводу.

Auth0 [6] є хмарним сервісом, який дозволяє швидко інтегрувати механізми автентифікації та авторизації у застосунок. Він підтримує JWT, багатофакторну автентифікацію та інтеграцію з різними постачальниками облікових записів. Проте його використання пов'язане з високою залежністю від сторонньої інфраструктури, що не є доцільним у межах розроблення власного програмного забезпечення.

Порівняльний аналіз розглянутих рішень наведено у таблиці 2.3.

Таблиця 2.3 – Порівняння засобів організації безпеки

Критерій	Spring Security	Keycloak	Auth0
Контроль над логікою	Повний	Частковий	Обмежений
Вбудована інтеграція	Так	Ні	Ні
Підтримка JWT	Так	Так	Так
Підтримка 2FA	Так	Так	Так
Необхідність окремого сервера	Ні	Так	Ні
Залежність від сторонніх сервісів	Ні	Часткова	Висока

2.3.3 Обґрунтування вибору засобу організації безпеки

На основі проведеного аналізу було обрано Spring Security [1] як основний засіб організації безпеки у розроблюваному серверному застосунку.

Даний вибір обумовлений тим, що Spring Security [1] забезпечує повний контроль над логікою автентифікації та авторизації, дозволяє реалізувати токен-орієнтований підхід із використанням JWT [2], а також підтримує інтеграцію механізмів двофакторної автентифікації.

Крім того, використання Spring Security [1] дозволяє уникнути залежності від сторонніх сервісів, що є важливим фактором у контексті забезпечення конфіденційності даних та гнучкості системи. Інтеграція даного рішення з іншими компонентами застосунку, зокрема Spring Boot та базою даних PostgreSQL [8], забезпечує створення єдиної та узгодженої архітектури системи.

Таким чином, обраний підхід дозволяє реалізувати сучасні механізми безпеки, що відповідають вимогам до серверного програмного забезпечення автентифікації користувачів.

2.4 Вибір механізму авторизації користувачів

2.4.1 Загальні підходи до авторизації користувачів

Після успішної автентифікації користувача система повинна визначити його права доступу до ресурсів. Для цього використовується механізм авторизації, який перевіряє, чи має користувач право виконувати певні дії.

У сучасних інформаційних системах використовуються різні підходи до авторизації, серед яких найбільш поширеними є сесійна авторизація, токен-орієнтована авторизація та протокол OAuth 2.0.

2.4.2 Сесійна авторизація

Сесійна авторизація базується на збереженні інформації про користувача на сервері у вигляді сесії. Після входу користувач отримує ідентифікатор сесії, який зберігається у браузері та передається при кожному запиті.

Основним недоліком такого підходу є необхідність зберігання стану на сервері, що ускладнює масштабування системи. Крім того, при великій кількості користувачів це призводить до збільшення навантаження на сервер.

2.4.3 Токен-орієнтована авторизація (JWT)

Токен-орієнтована авторизація передбачає використання спеціального токена, який передається між клієнтом і сервером. Одним із найпоширеніших стандартів є JSON Web Token (JWT) [2].

У розроблюваному застосунку доцільно реалізувати два типи токенів:

- access token, який використовується для доступу до захищених ресурсів;
- тимчасовий токен, який використовується на етапі проходження двофакторної автентифікації.

Після введення коректного логіна та пароля, у разі активованої двофакторної автентифікації, користувачу не одразу видається повний доступ. Замість цього генерується тимчасовий токен із обмеженим часом дії, який дозволяє виконати лише перевірку другого фактора. Після успішного підтвердження коду двофакторної автентифікації генерується повноцінний access token.

Такий підхід підвищує рівень безпеки, оскільки навіть у разі перехоплення облікових даних злоумисник не зможе отримати доступ до системи без проходження другого етапу перевірки.

2.4.4 Авторизація на основі OAuth 2.0

OAuth 2.0 є протоколом авторизації, який дозволяє надавати доступ до ресурсів від імені користувача через сторонні сервіси. Він широко використовується для інтеграції з соціальними мережами та іншими платформами.

Однак використання OAuth 2.0 є доцільним у випадках, коли необхідна інтеграція з зовнішніми сервісами. У межах даного проєкту реалізується власна система автентифікації, тому застосування цього протоколу є надмірним.

Порівняльний аналіз підходів до авторизації наведено у таблиці 2.4.

Таблиця 2.4 – Порівняння механізмів авторизації

Критерій	Сесійна авторизація	JWT	OAuth 2.0
Зберігання стану	На сервері	Не зберігається	Частково
Масштабованість	Середня	Висока	Висока
Продуктивність	Середня	Висока	Середня
Складність реалізації	Низька	Середня	Висока
Залежність від сторонніх сервісів	Ні	Ні	Так
Підходить для REST API	Ні	Так	Так

2.4.5 Обґрунтування вибору механізму авторизації

На основі проведеного аналізу було обрано токен-орієнтований підхід із використанням JSON Web Token (JWT) [2].

Це рішення дозволяє реалізувати безстанну модель авторизації, що значно підвищує масштабованість системи та зменшує навантаження на сервер. Крім того, використання JWT [2] забезпечує просту інтеграцію з REST API та дозволяє ефективно реалізувати механізм двофакторної автентифікації.

Застосування додаткового тимчасового токена для проходження другого етапу автентифікації підвищує рівень безпеки системи та зменшує ризик несанкціонованого доступу.

Таким чином, обраний підхід є оптимальним для реалізації серверного програмного забезпечення автентифікації користувачів.

2.5 Вибір механізму двофакторної автентифікації

2.5.1 Загальні принципи двофакторної автентифікації

Двофакторна автентифікація (Two-Factor Authentication, 2FA) є додатковим рівнем захисту, який передбачає використання двох незалежних факторів для підтвердження особи користувача.

Як правило, використовуються такі фактори:

- те, що користувач знає (пароль);
- те, що користувач має (мобільний пристрій або токен);
- те, що є властивістю користувача (біометричні дані).

У даному проєкті реалізовано двофакторну автентифікацію з використанням одноразових кодів, що генеруються на мобільному пристрої користувача.

2.5.2 Аналіз методів реалізації 2FA

Було розглянуто кілька основних підходів до реалізації двофакторної автентифікації:

SMS-автентифікація – передбачає надсилання одноразового коду на номер телефону користувача. Основною перевагою є простота використання, однак цей метод має суттєві недоліки, зокрема залежність від мобільного оператора та вразливість до атак типу SIM-swapping.

Email-автентифікація – код підтвердження надсилається на електронну пошту користувача. Даний метод є простим у реалізації, проте рівень безпеки залежить від захищеності поштового акаунта, що робить його менш надійним.

TOTP [5]-автентифікація – цей метод базується на генерації одноразових кодів на основі секретного ключа та поточного часу. Коди генеруються на стороні клієнта (наприклад, у застосунку Google Authenticator [3]) і не передаються через мережу.

Основними перевагами TOTP [5] є:

- відсутність залежності від сторонніх сервісів;
- високий рівень безпеки;
- неможливість перехоплення коду під час передачі.

Порівняльний аналіз методів двофакторної автентифікації наведено у таблиці 2.5.

Таблиця 2.5 – Порівняння методів 2FA

Критерій	SMS	Email	TOTP (Google Authenticator)
Рівень безпеки	Середній	Низький	Високий
Залежність від мережі	Висока	Висока	Низька
Вразливість до атак	Висока	Середня	Низька
Простота реалізації	Середня	Висока	Середня
Необхідність сторонніх сервісів	Так	Так	Ні

2.5.3 Реалізація двофакторної автентифікації у системі

У розроблюваному застосунку реалізовано механізм двофакторної автентифікації на основі алгоритму TOTP [5].

При активації 2FA сервер генерує унікальний секретний ключ, який тимчасово зберігається та використовується для створення QR-коду. Користувач сканує QR-код за допомогою мобільного застосунку (наприклад, Google Authenticator [3]), після чого вводить згенерований код для підтвердження налаштування.

Після успішної перевірки секретний ключ зберігається у системі, а двофакторна автентифікація [4] активується для користувача.

Під час входу в систему, після перевірки логіна та пароля, користувач повинен ввести одноразовий код, згенерований на основі секретного ключа. Лише після успішної перевірки цього коду користувачу надається повний доступ до системи.

2.5.4 Обґрунтування вибору механізму 2FA

На основі проведеного аналізу було обрано метод двофакторної автентифікації на основі TOTP [5].

Даний підхід забезпечує високий рівень безпеки, не залежить від сторонніх сервісів та дозволяє реалізувати автономну систему автентифікації. Крім того, використання TOTP [5] добре інтегрується з токен-орієнтованою моделлю авторизації та підвищує загальний рівень захисту користувацьких облікових записів.

Таким чином, обраний механізм є оптимальним для реалізації серверного програмного забезпечення автентифікації користувачів.

2.6 Вибір системи керування базами даних

2.6.1 Вимоги до системи керування базами даних

Система керування базами даних (СКБД) є важливим компонентом серверного програмного забезпечення, оскільки забезпечує зберігання, обробку та захист даних користувачів.

Для розроблюваного застосунку були визначені такі основні вимоги до СКБД:

- забезпечення надійного зберігання даних користувачів;
- підтримка транзакцій та цілісності даних;
- висока продуктивність при обробці запитів;

- можливість масштабування;
- підтримка інтеграції з Java та Spring Boot;
- організації безпеки даних.

З урахуванням цих вимог було проведено аналіз найбільш поширених систем керування базами даних.

2.6.2 Аналіз сучасних СКБД

Для реалізації серверного застосунку було розглянуто такі СКБД: PostgreSQL [8], MySQL та MongoDB.

PostgreSQL [8] є об'єктно-реляційною системою керування базами даних з відкритим вихідним кодом. Вона забезпечує високу надійність, підтримує транзакції (ACID), має розвинені механізми забезпечення цілісності даних та підтримує складні запити. PostgreSQL [8] добре інтегрується з Java через технологію JPA/Hibernate, що робить її зручною для використання у застосунках на базі Spring Boot.

MySQL також є популярною реляційною СКБД, яка забезпечує високу продуктивність та простоту використання. Однак у деяких випадках вона поступається PostgreSQL [8] у питаннях підтримки складних запитів та гнучкості.

MongoDB є документно-орієнтованою (NoSQL) базою даних, яка забезпечує високу швидкість роботи та масштабованість. Проте відсутність суворої схеми даних та обмежена підтримка транзакцій роблять її менш придатною для систем, де важлива цілісність даних, зокрема систем автентифікації.

Порівняльний аналіз систем керування базами даних наведено у таблиці 2.6.

Таблиця 2.6 – Порівняння систем керування базами даних

Критерій	PostgreSQL	MySQL	MongoDB
Тип БД	Реляційна	Реляційна	NoSQL
Підтримка ACID	Так	Так	Часткова
Продуктивність	Висока	Висока	Висока
Підтримка складних запитів	Висока	Середня	Обмежена
Масштабованість	Висока	Висока	Дуже висока
Інтеграція з Spring	Висока	Висока	Середня
Цілісність даних	Висока	Висока	Середня

2.6.3 Обґрунтування вибору СКБД

На основі проведеного аналізу було обрано систему керування базами даних PostgreSQL [8].

Даний вибір обумовлений тим, що PostgreSQL [8] забезпечує високий рівень надійності, підтримує транзакції та гарантує цілісність даних, що є критично важливим для систем автентифікації користувачів. Крім того, PostgreSQL [8] має широкі можливості для роботи зі складними запитами та добре інтегрується з технологіями, використаними у проєкті, зокрема Spring Boot та Hibernate.

Таким чином, використання PostgreSQL [8] дозволяє забезпечити надійне зберігання даних користувачів та ефективну роботу серверного застосунку.

2.7 Вибір середовища розробки та допоміжних інструментів

2.7.1 Вимоги до середовища розробки

Для ефективного створення серверного програмного забезпечення необхідно обрати сучасне середовище розробки та інструменти, які забезпечують зручність написання коду, його тестування, налагодження та подальший супровід.

Основними вимогами до середовища розробки є:

- підтримка мови програмування Java;
- інтеграція з фреймворком Spring Boot;
- наявність засобів налагодження та аналізу коду;
- підтримка систем контролю версій;
- можливість роботи з базами даних;
- зручність розробки вебзастосунків.

2.7.2 Аналіз середовищ розробки

Для реалізації проєкту було розглянуто такі середовища розробки: IntelliJ IDEA, Eclipse та Visual Studio Code.

IntelliJ IDEA є одним із найпопулярніших середовищ розробки для Java. Воно забезпечує глибоку інтеграцію з фреймворком Spring Boot, має потужні засоби автодоповнення коду, рефакторингу, налагодження та аналізу помилок. Крім того, IntelliJ IDEA підтримує інтеграцію з системами контролю версій та базами даних.

Eclipse є безкоштовним середовищем розробки з відкритим вихідним кодом. Воно також підтримує Java та Spring Boot, проте поступається IntelliJ IDEA у зручності використання та швидкості роботи.

Visual Studio Code є легким редактором коду, який підтримує різні мови програмування через розширення. Він є зручним для фронтенд-

розробки, однак для створення повноцінних Java-застосунків потребує додаткової конфігурації.

2.7.3 Допоміжні інструменти розробки

Окрім середовища розробки, у процесі створення застосунку використовувалися додаткові інструменти:

- Gradle – система автоматизації збірки, яка використовується для керування залежностями та процесом компіляції проєкту;
- Git – система контролю версій, що дозволяє відстежувати зміни в коді та забезпечує спільну роботу над проєктом;
- PostgreSQL [8] – система керування базами даних, яка використовується для зберігання даних користувачів;
- Postman – інструмент для тестування REST API, який дозволяє перевіряти коректність роботи серверних endpoint-ів;
- Google Authenticator [3] – мобільний застосунок, який використовується для генерації одноразових кодів у межах двофакторної автентифікації.

2.7.4 Обґрунтування вибору інструментів розробки

На основі проведеного аналізу було обрано середовище розробки IntelliJ IDEA, оскільки воно забезпечує найвищий рівень підтримки мови Java та фреймворку Spring Boot, має зручний інтерфейс та потужні засоби налагодження.

Використання системи Gradle дозволяє автоматизувати процес збірки та ефективно керувати залежностями проєкту. Застосування системи контролю версій Git забезпечує надійне збереження історії змін та можливість відновлення попередніх версій коду.

Таким чином, обрані інструменти дозволяють забезпечити ефективний процес розроблення, тестування та супроводу серверного програмного забезпечення.

2.8 Висновки до розділу

У другому розділі дипломної роботи було проведено аналіз та обґрунтування вибору технологій і засобів, що використовуються для розроблення серверного програмного забезпечення автентифікації користувачів.

Зокрема, було обрано мову програмування Java, яка забезпечує високий рівень продуктивності, безпеки та масштабованості, а також має розвинену екосистему для створення серверних застосунків. Як основний серверний фреймворк використано Spring Boot, що дозволяє швидко реалізувати REST API та інтегрувати сучасні механізми безпеки.

Для організації захисту системи обрано модуль Spring Security [1], який надає гнучкі можливості для реалізації автентифікації та авторизації користувачів. У якості механізму авторизації використано токен-орієнтований підхід із застосуванням JSON Web Token (JWT) [2], що дозволяє реалізувати безстанну модель доступу та підвищити масштабованість системи.

З метою підвищення рівня захисту облікових записів користувачів було обрано механізм двофакторної автентифікації на основі алгоритму TOTP [5]. Даний підхід забезпечує високий рівень безпеки та не потребує використання сторонніх сервісів.

Для зберігання даних було обрано систему керування базами даних PostgreSQL [8], яка забезпечує надійність, цілісність та ефективну обробку даних. У якості середовища розробки використано IntelliJ IDEA, а також допоміжні інструменти, такі як Gradle, Git та Postman.

Таким чином, обраний стек технологій дозволяє реалізувати надійне, масштабоване та безпечне серверне програмне забезпечення автентифікації користувачів, що відповідає сучасним вимогам до інформаційних систем.

З ПРОЄКТУВАННЯ СЕРВЕРНОГО ПРОГРАМНОГО ЗАСТОСУНКУ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ

3.1 Проєктування архітектури системи

3.1.1 Загальна структура системи

Розроблений програмний застосунок являє собою серверну систему автентифікації користувачів, побудовану на основі клієнт-серверної архітектури. Основним призначенням системи є організація безпечної реєстрації, автентифікації, авторизації та керування доступом користувачів до захищених ресурсів вебзастосунку.

Архітектура системи побудована із використанням технології REST API, що забезпечує взаємодію між клієнтською та серверною частинами через HTTP-запити у форматі JSON. Серверна частина реалізована із використанням фреймворку Spring Boot та функціонує як окремий backend-застосунок.

У процесі роботи клієнт надсилає HTTP-запити до REST API, після чого сервер виконує їх обробку, звертається до бази даних PostgreSQL, здійснює перевірку облікових даних користувача та формує відповідь у форматі JSON. Для авторизації доступу до захищених ресурсів використовується JWT-токен, який генерується після успішної автентифікації користувача.

До складу системи входять такі основні компоненти:

- модуль автентифікації користувачів;
- модуль JWT-авторизації;
- модуль двофакторної автентифікації (2FA);
- модуль керування ролями користувачів;
- модуль адміністративного керування;
- модуль підтримки користувачів та обробки тікетів;
- модуль взаємодії з базою даних PostgreSQL.

Загальна архітектура серверного застосунку автентифікації користувачів наведена на рисунку 3.1.

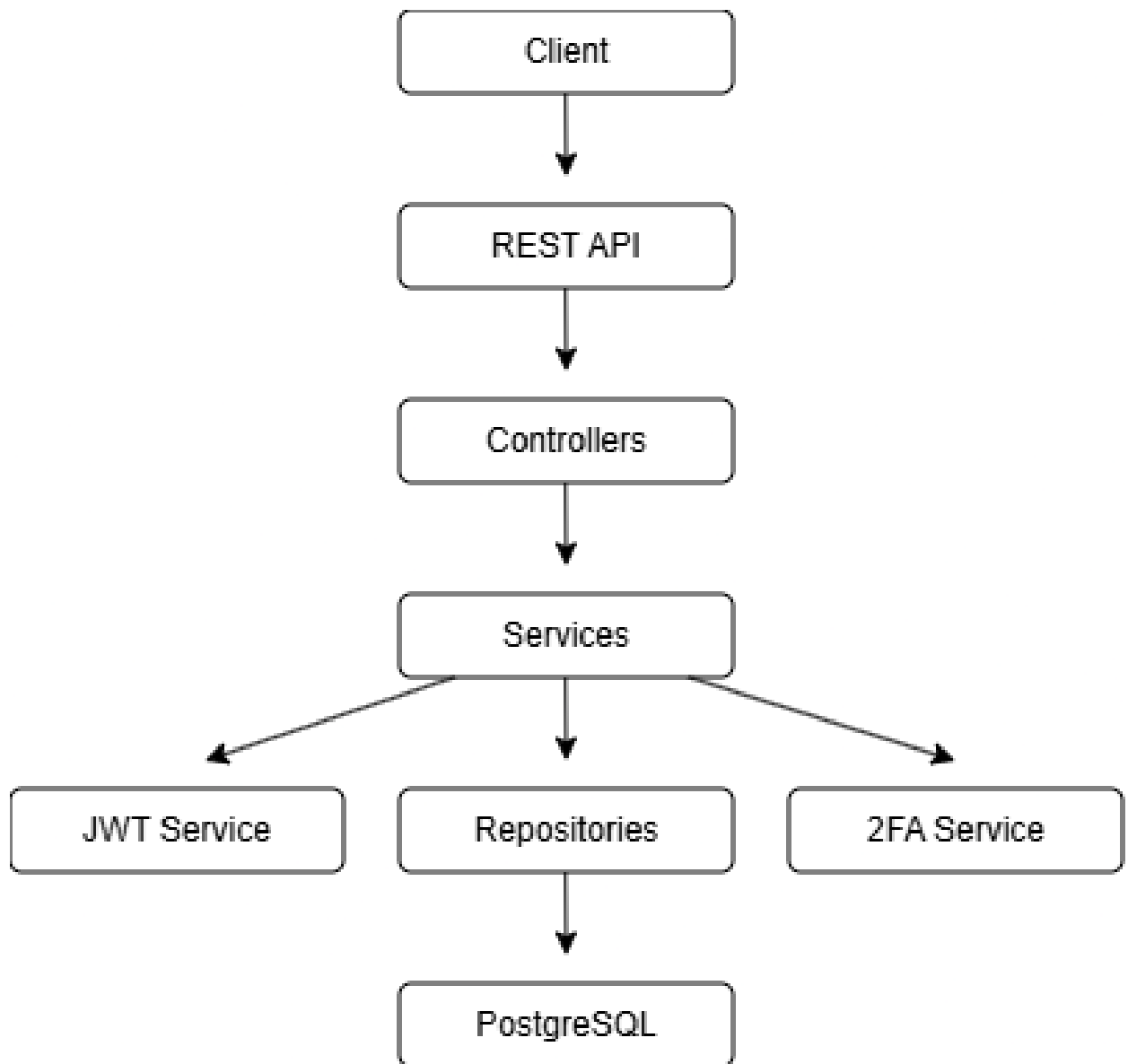


Рисунок 3.1 – Загальна архітектура серверного застосунку автентифікації користувачів

3.1.2 Архітектура Controller–Service–Repository

Для організації модульності, масштабованості та підтримуваності програмного коду в системі використано багаторівневу архітектуру типу Controller–Service–Repository.

Рівень Controller відповідає за обробку HTTP-запитів, приймання вхідних параметрів, виклик відповідних сервісів та формування HTTP-відповідей. Контролери реалізують REST API системи та забезпечують взаємодію клієнта із серверною частиною застосунку.

Рівень Service реалізує основну бізнес-логіку системи. На цьому рівні виконуються:

- перевірка коректності вхідних даних;
- реєстрація користувачів;
- генерація JWT-токенів;
- перевірка двофакторної автентифікації;
- обробка ролей користувачів;
- керування тікетами підтримки;
- аудит подій безпеки.

Рівень Repository забезпечує взаємодію із базою даних PostgreSQL через технологію Spring Data JPA. Репозиторії виконують операції створення, читання, оновлення та видалення даних без необхідності ручного написання SQL-запитів.

Використання такої архітектури дозволяє:

- розділити відповідальність між компонентами системи;
- спростити тестування програмного коду;
- забезпечити незалежність бізнес-логіки від рівня зберігання даних;
- підвищити масштабованість та підтримуваність застосунку.

Структура пакетів програмного застосунку організована відповідно до функціонального призначення компонентів системи. Основні класи розподілені між пакетами controller, service, repository, model, dto, security та config.

Багаторівнева архітектура серверного застосунку із розподілом компонентів за рівнями відповідальності наведена на рисунку 3.2.

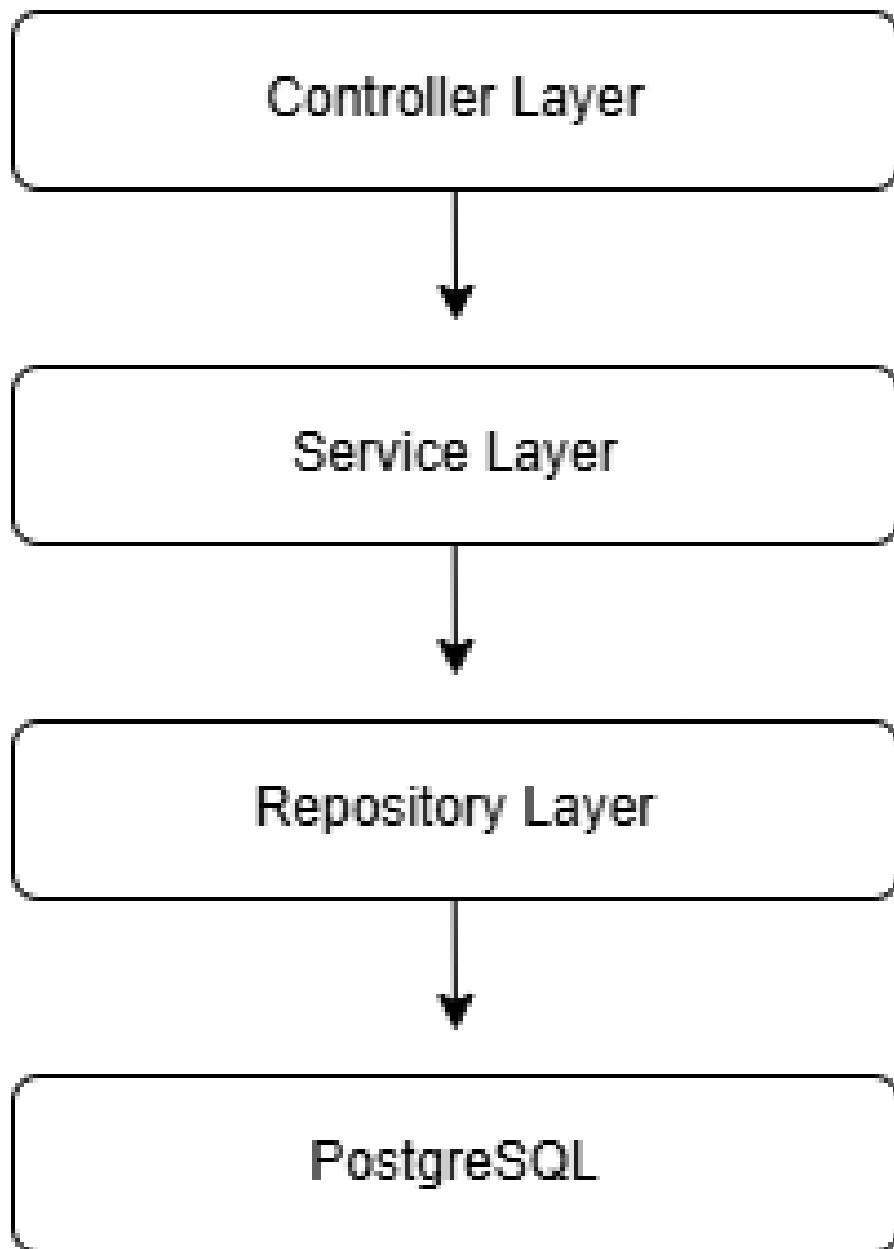


Рисунок 3.2 – Багаторівнева архітектура серверного застосунку

3.1.3 Взаємодія компонентів системи

Взаємодія між компонентами системи здійснюється відповідно до принципів багаторівневої архітектури. Після надходження HTTP-запиту контролер виконує його первинну обробку та передає дані до відповідного сервісу. Сервісний рівень реалізує бізнес-логіку та, у разі необхідності, звертається до репозиторіїв для отримання або збереження інформації у базі даних.

Послідовність взаємодії компонентів системи під час автентифікації користувача наведена на рисунку 3.3.

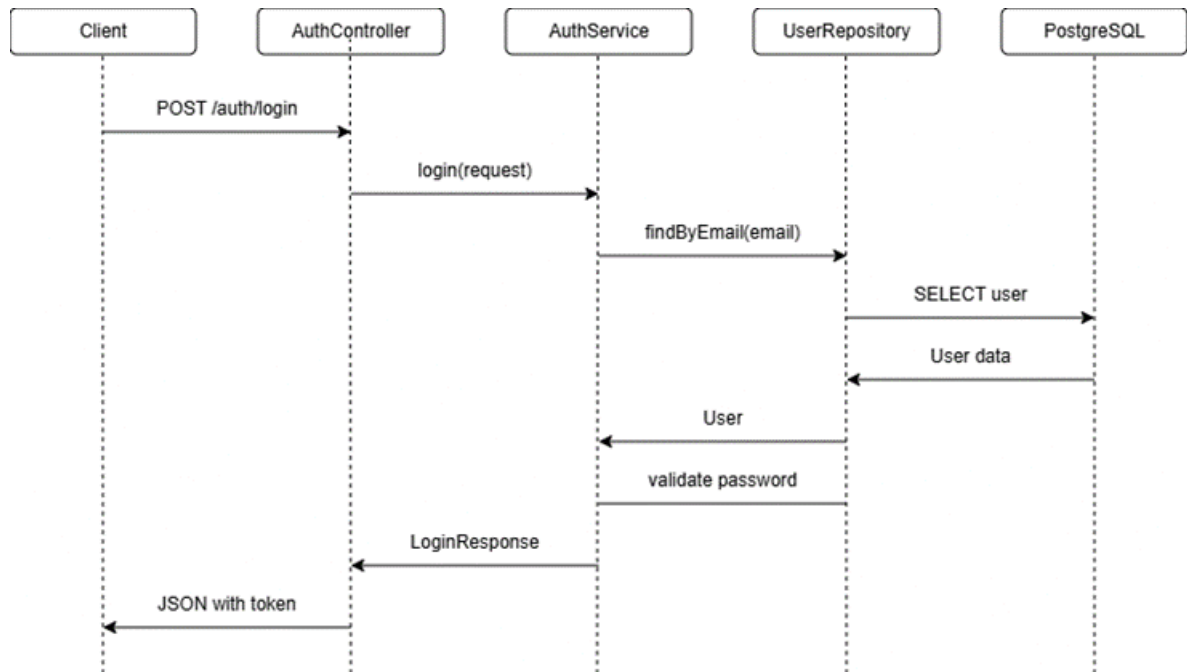


Рисунок 3.3 – UML-діаграма послідовності автентифікації користувача

3.2 Проектування моделі даних

3.2.1 Структура бази даних

Для зберігання інформації у серверному застосунку використовується реляційна система керування базами даних PostgreSQL [8]. Вибір PostgreSQL обумовлений підтримкою транзакційності, високою надійністю, підтримкою складних типів даних та можливістю ефективної інтеграції із Spring Data JPA [9].

Структура бази даних спроектована відповідно до функціональних вимог системи автентифікації користувачів. Основними сутностями системи є користувачі, журнали аудиту подій безпеки та тікети підтримки користувачів.

Центральною таблицею бази даних є таблиця users, яка містить інформацію про облікові записи користувачів, параметри безпеки та дані для

двофакторної автентифікації. Для ідентифікації користувачів використовується UUID, що дозволяє забезпечити унікальність ідентифікаторів та ускладнює їх перебір сторонніми користувачами.

Структура бази даних серверного застосунку наведена на рисунку 3.4.

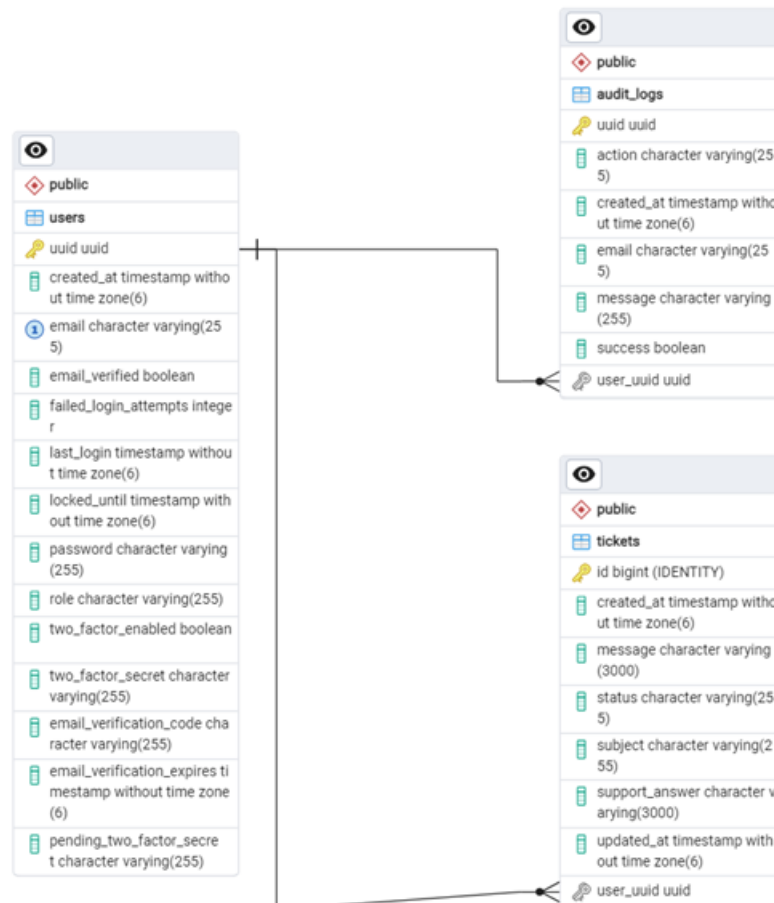


Рисунок 3.4 – ER-діаграма структури бази даних серверного застосунку

3.2.2 Проєктування сутності User

Сутність User є основною моделлю системи та використовується для зберігання інформації про облікові записи користувачів. Дана сутність реалізована у вигляді JPA Entity та відповідає таблиці users у базі даних PostgreSQL.

Для первинного ключа використовується тип UUID, який генерується автоматично під час створення нового користувача. Використання UUID

підвищує безпеку системи та ускладнює можливість перебору ідентифікаторів користувачів.

Поле email використовується як унікальний логін користувача та має обмеження `unique = true`. Поле password містить хешований пароль користувача, сформований із використанням алгоритму BCrypt [1], [9].

Для реалізації рольової моделі доступу використовується поле role, яке містить роль користувача у системі. У застосунку реалізовано ролі USER, SUPPORT та ADMIN.

Для реалізації захисту від brute-force атак у сутності User використовуються поля `failedLoginAttempts` та `lockedUntil`. Після перевищення допустимої кількості невдалих спроб входу система тимчасово блокує обліковий запис користувача.

Підтримка двофакторної автентифікації реалізована через поля:

- `twoFactorEnabled`;
- `twoFactorSecret`;
- `pendingTwoFactorSecret`.

Поле `twoFactorSecret` використовується для зберігання секретного ключа TOTP, а `pendingTwoFactorSecret` – для тимчасового зберігання секрету до завершення активації 2FA.

Для реалізації підтвердження електронної пошти використовуються поля:

- `emailVerificationCode`;
- `emailVerificationExpires`;
- `emailVerified`.

Поле `emailVerified` визначає, чи було підтверджено адресу електронної пошти користувача.

Також у сутності зберігаються службові поля `createdAt` та `lastLogin`, які дозволяють відстежувати час створення облікового запису та останнього входу користувача до системи.

3.2.3 Організація захисту даних

Для організації безпеки персональних даних користувачів у системі реалізовано комплекс механізмів захисту інформації.

Паролі користувачів не зберігаються у відкритому вигляді. Перед збереженням у базі даних пароль обробляється за допомогою алгоритму BCryptPasswordEncoder [1], [9], який виконує хешування із використанням salt-значення. Такий підхід ускладнює можливість компрометації паролів у випадку витоку бази даних.

Для реалізації JWT-авторизації [2] у системі використовуються цифрові токени доступу, які містять інформацію про користувача та його роль. У токенах не зберігаються конфіденційні дані користувачів, а доступ до захищених ресурсів виконується лише після перевірки цифрового підпису JWT.

Двофакторна автентифікація реалізована із використанням алгоритму TOTP [5]. Секретні ключі для генерації одноразових кодів зберігаються у базі даних та використовуються під час перевірки кодів автентифікації.

Для розмежування доступу до функціональних можливостей застосунку використовується рольова модель RBAC [1]. Доступ до адміністративних endpoint-ів дозволений лише користувачам із відповідними ролями.

Також у системі реалізовано механізм аудиту подій безпеки, який дозволяє фіксувати:

- успішні входи у систему;
- невдалі спроби автентифікації;
- блокування облікових записів;
- зміну ролей користувачів;
- операції адміністративного керування.

3.2.4 Взаємодія із PostgreSQL через JPA

Для взаємодії серверного застосунку із базою даних PostgreSQL використовується технологія Spring Data JPA та ORM-фреймворк Hibernate [9]-[10].

Кожна сутність системи представлена у вигляді JPA Entity, яка відображається на відповідну таблицю у базі даних. Для опису структури таблиць використовуються анотації @Entity, @Table, @Id, @Column, @Enumerated та інші.

Доступ до даних реалізований через репозиторії, які наслідують інтерфейс JpaRepository [9]. Такий підхід дозволяє виконувати CRUD-операції без необхідності ручного написання SQL-запитів.

Наприклад, UserRepository використовується для:

- пошуку користувача за email;
- перевірки існування облікового запису;
- збереження користувача;
- оновлення інформації користувача.

Hibernate автоматично виконує генерацію SQL-запитів, синхронізацію сутностей із таблицями бази даних та управління життєвим циклом об'єктів.

Використання Spring Data JPA дозволяє:

- зменшити кількість шаблонного коду;
- спростити роботу із базою даних;
- підвищити підтримуваність програмного застосунку;
- забезпечити незалежність бізнес-логіки від конкретної СУБД.

3.3 Проєктування механізму автентифікації

3.3.1 Алгоритм реєстрації користувача

Реєстрація користувача є одним із основних функціональних процесів серверного застосунку. Під час реєстрації система виконує перевірку

коректності введених даних, створення нового облікового запису та формування початкових параметрів безпеки користувача.

Процес реєстрації починається із надсилання клієнтом HTTP POST-запиту на endpoint `/auth/register`, який містить адресу електронної пошти та пароль користувача у форматі JSON.

Після отримання запиту серверний застосунок виконує:

- перевірку коректності введених даних;
- перевірку унікальності email-адреси;
- хешування пароля за допомогою `BCryptPasswordEncoder` [1]-[9];
- створення нового об'єкта користувача;
- генерацію коду підтвердження електронної пошти;
- збереження користувача у базі даних PostgreSQL [8].

Пароль користувача не зберігається у відкритому вигляді. Перед збереженням у базі даних пароль проходить процедуру BCrypt-хешування, що дозволяє забезпечити додатковий рівень захисту облікових даних користувача [11].

Після успішної реєстрації система надсилає користувачу лист із кодом або посиланням для підтвердження електронної пошти. До моменту підтвердження email доступ до окремих функцій системи може бути обмежений.

Алгоритм реєстрації користувача наведений на рисунку 3.5.



Рисунок 3.5 – Блок-схема алгоритму реєстрації користувача

3.3.2 Алгоритм входу в систему

Процес автентифікації користувача реалізований через endpoint `/auth/login`. Під час входу користувач надсилає адресу електронної пошти та пароль, після чого сервер виконує перевірку облікових даних та формує JWT-токен доступу.

Після отримання запиту система виконує:

- пошук користувача у базі даних;
- перевірку правильності пароля;
- перевірку статусу блокування облікового запису;
- перевірку підтвердження електронної пошти;
- перевірку стану двофакторної автентифікації.

У випадку успішної автентифікації система генерує JWT access token [2], який використовується для доступу до захищених endpoint-ів REST API.

Якщо для користувача активована двофакторна автентифікація, система додатково генерує тимчасовий tempToken, який використовується під час перевірки TOTP-коду. Після успішного підтвердження TOTP-коду користувач отримує основний JWT access token.

Під час невдалих спроб входу система збільшує значення failedLoginAttempts. Після перевищення допустимої кількості невдалих спроб обліковий запис тимчасово блокується.

Послідовність процесу автентифікації користувача наведена на рисунку 3.6.

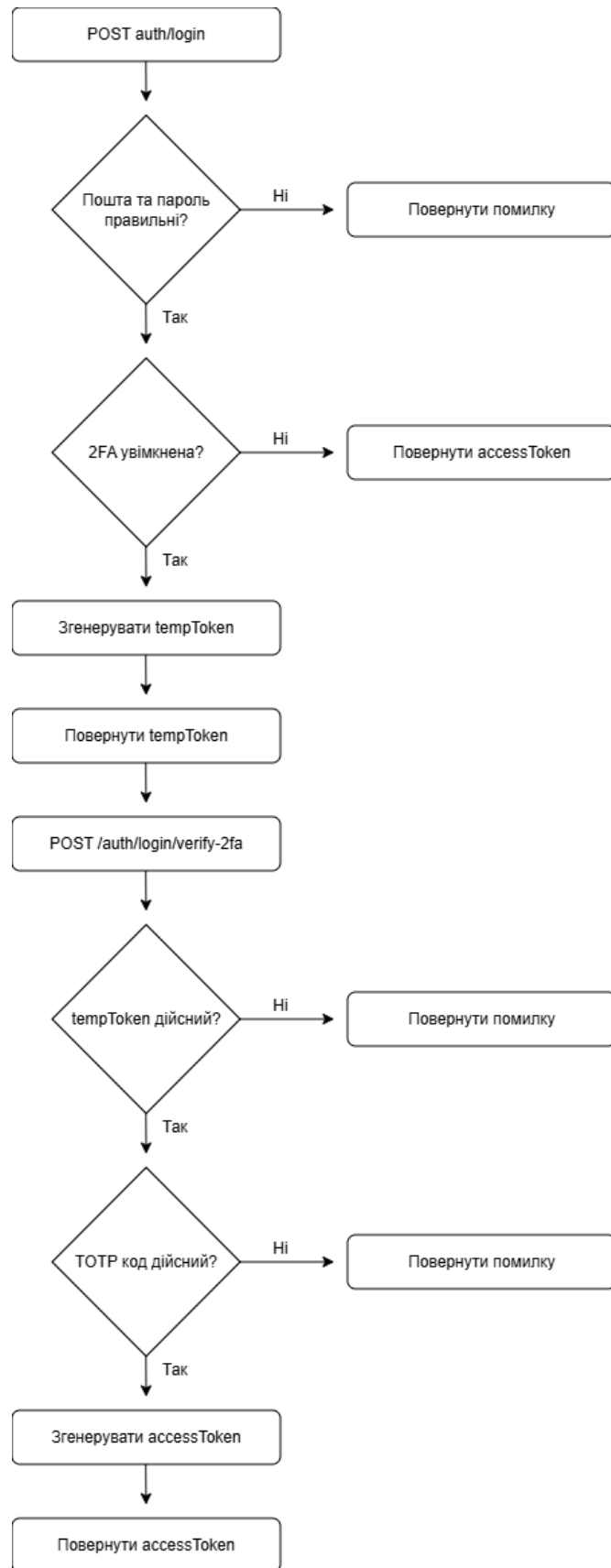


Рисунок 3.6 – Блок-схема алгоритму входу з використанням двофакторної автентифікації

3.3.3 Реалізація tempToken

Для реалізації двоетапної автентифікації у серверному застосунку використовується механізм тимчасових токенів tempToken.

Життєвий цикл tempToken наведений на рисунку 3.7.

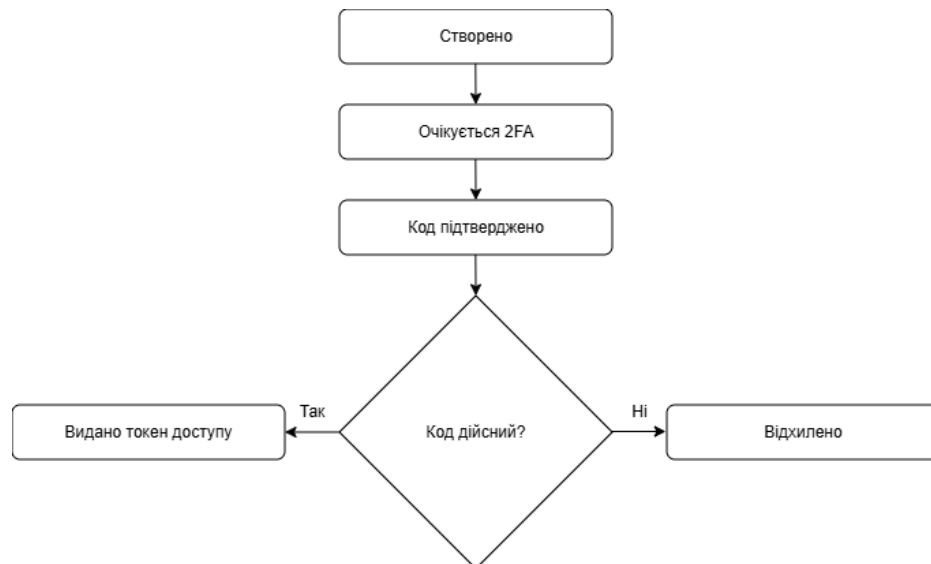


Рисунок 3.7 – Діаграма станів tempToken у процесі двофакторної автентифікації

TempToken являє собою спеціальний JWT-токен із обмеженим часом дії, який видається користувачу після успішної перевірки email та пароля, але до завершення перевірки ТОТР-коду двофакторної автентифікації.

Основною метою використання tempToken є:

- розділення етапів автентифікації;
- недопущення видачі повноцінного access token до завершення 2FA;
- організація додаткового рівня безпеки під час входу у систему.

TempToken містить:

- email користувача;
- тип токена;
- час створення;

- час завершення дії токена.

Для tempToken використовується окремий тип токена 2FA_PENDING, що дозволяє системі відокремлювати тимчасові токени від основних JWT access token.

Час дії tempToken обмежений декількома хвилинами, після чого токен автоматично стає недійсним. Такий підхід дозволяє мінімізувати ризик компрометації тимчасових токенів.

3.3.4 Захист від brute-force атак

Для організації захисту від brute-force атак у системі реалізовано механізм обмеження кількості невдалих спроб автентифікації користувача.

Після кожної невдалої спроби входу система збільшує значення failedLoginAttempts у сутності User. Після перевищення допустимого ліміту спроб обліковий запис користувача тимчасово блокується.

Інформація про час завершення блокування зберігається у полі lockedUntil. До завершення періоду блокування система не дозволяє користувачу виконувати повторні спроби входу у систему.

Після успішної автентифікації лічильник failedLoginAttempts автоматично скидається.

Такий підхід дозволяє:

- зменшити ризик підбору паролів;
- ускладнити автоматизовані атаки;
- підвищити рівень безпеки системи автентифікації [4].

Алгоритм захисту від brute-force атак наведений на рисунку 3.8.

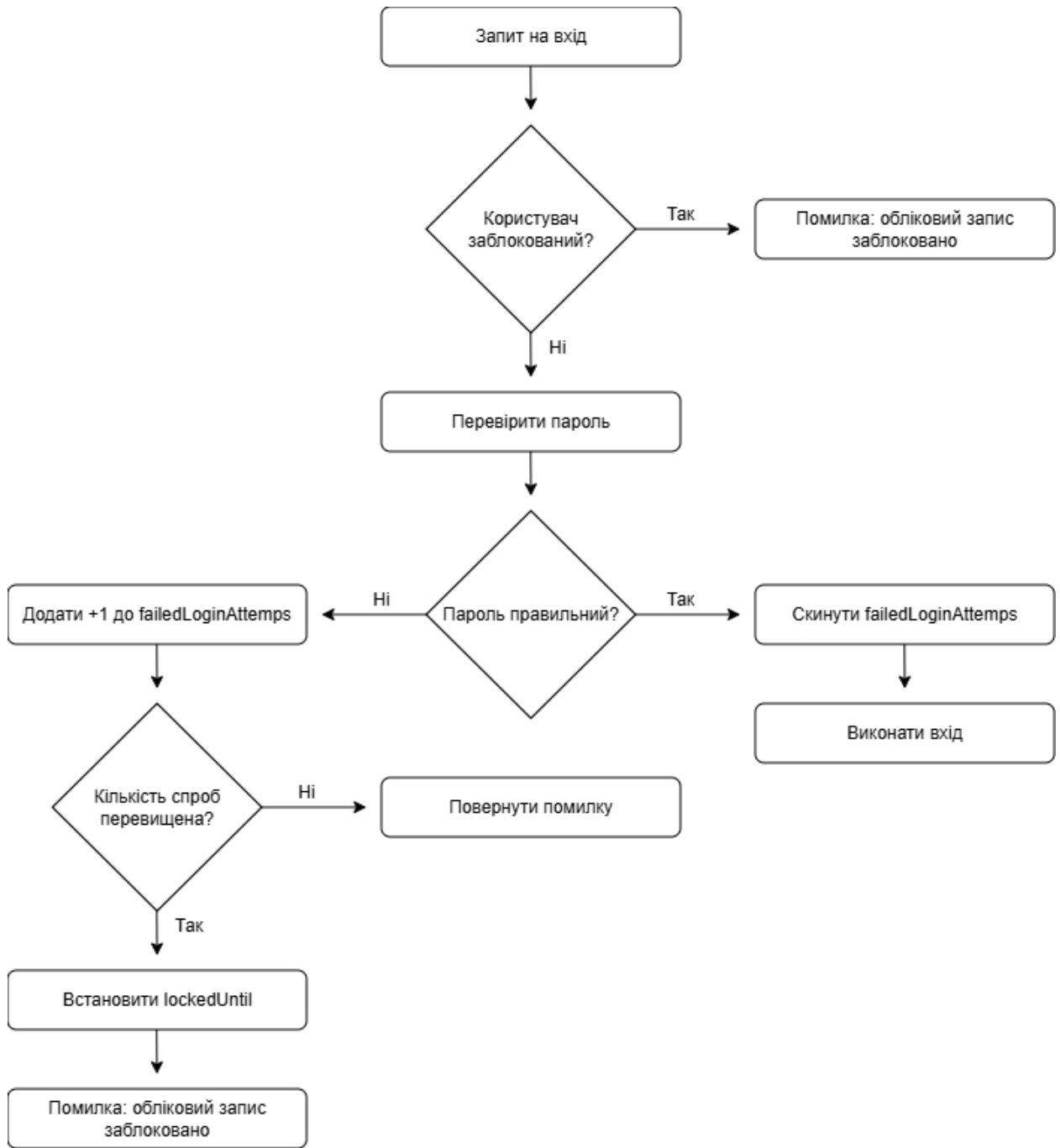


Рисунок 3.8 – Блок-схема алгоритму захисту від brute-force атак

3.4 Проєктування JWT-авторизації

3.4.1 Структура JWT у системі

Для реалізації механізму авторизації у серверному застосунку використовується технологія JSON Web Token (JWT) [2]. JWT використовується для підтвердження автентичності користувача та надання доступу до захищених ресурсів REST API.

Після успішної автентифікації сервер генерує JWT access token, який передається клієнту та використовується під час виконання наступних HTTP-запитів. Токен передається у заголовок Authorization у форматі Bearer Token.

У серверному застосунку JWT-токен містить такі основні дані:

- subject (sub) – email користувача;
- role – роль користувача у системі;
- type – тип токена;
- iat – час створення токена;
- exp – час завершення дії токена.

Для реалізації двофакторної автентифікації у системі використовуються два типи tokenів:

- ACCESS – основний токен доступу;
- 2FA_PENDING – тимчасовий токен для проходження двофакторної автентифікації.

Структура JWT-токена, що використовується у системі, наведена на рисунку 3.9.

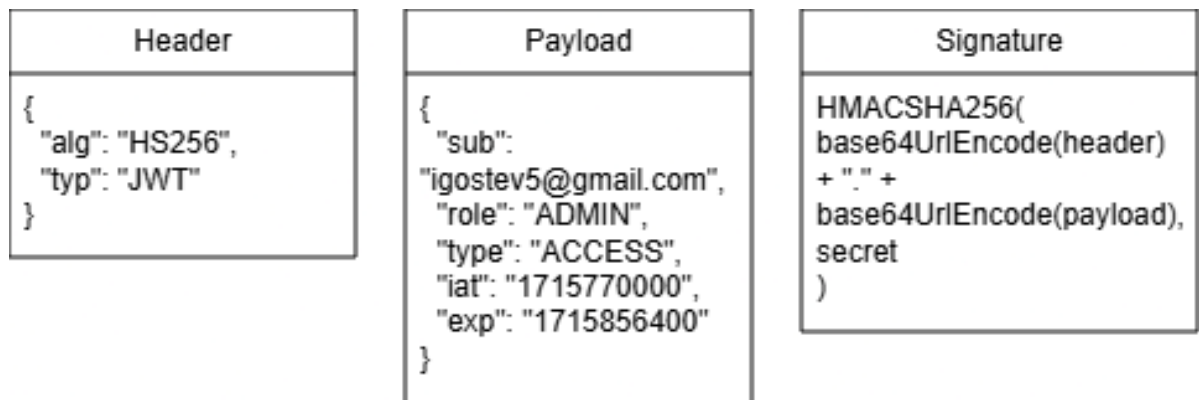


Рисунок 3.9 – Структура JWT-токена у серверному застосунку

3.4.2 Генерація access token

Генерація JWT access token виконується після успішної автентифікації користувача. Для створення токенів у застосунку використовується бібліотека `io.jsonwebtoken` та алгоритм цифрового підпису HS256 [2].

Процес генерації токена реалізований у компоненті `JwtUtils`. Під час створення токена система формує `payload` із даними користувача та встановлює час завершення дії токена.

У процесі генерації access token до токена додаються:

- email користувача;
- роль користувача;
- тип токена;
- дата створення;
- дата завершення дії.

Для access token використовується обмежений час життя, після завершення якого користувач повинен повторно пройти автентифікацію у системі.

Підпис JWT-токена виконується за допомогою секретного ключа, що зберігається у конфігурації серверного застосунку. Після генерації токен передається клієнту у форматі JSON-відповіді.

3.4.3 Перевірка JWT

Для перевірки JWT-токенів у серверному застосунку використовується компонент `JwtAuthenticationFilter`, який інтегрований у Spring Security [1].

`JwtAuthenticationFilter` перехоплює HTTP-запити до захищених endpoint-ів та виконує:

- перевірку наявності Authorization header;
- перевірку формату Bearer Token;

- перевірку цифрового підпису JWT;
- перевірку часу завершення дії токена;
- перевірку типу токена;
- отримання ролі користувача із payload токена.

Після успішної перевірки токена система формує Authentication object та додає його до SecurityContext Spring Security. Це дозволяє виконувати рольову авторизацію під час доступу до endpoint-ів системи.

Послідовність перевірки JWT під час обробки HTTP-запиту наведена на рисунку 3.10.

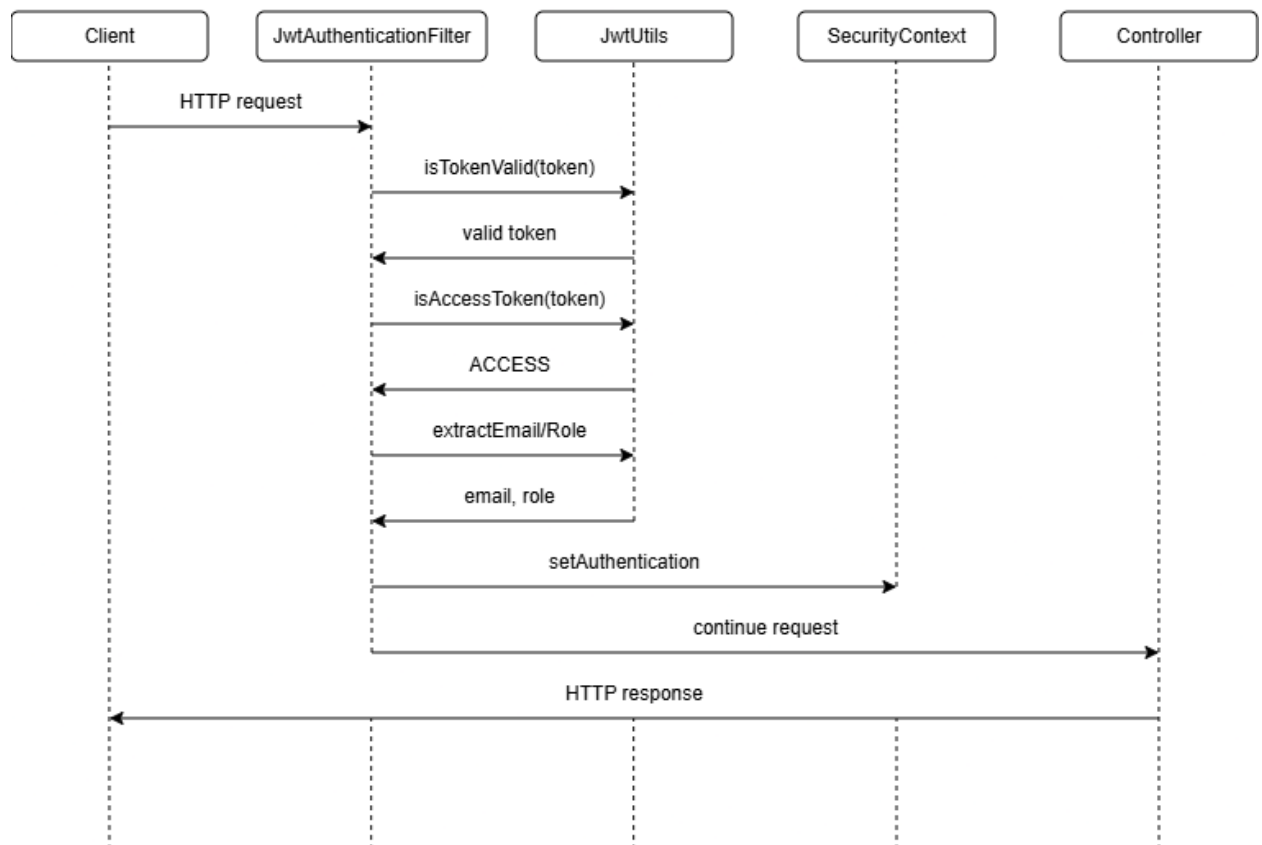


Рисунок 3.10 – UML-діаграма перевірки JWT під час обробки HTTP-запиту

3.4.4 Авторизація endpoint-ів

Для розмежування доступу до функціональних можливостей системи використовується рольова модель авторизації RBAC (Role-Based Access Control) [1].

У серверному застосунку реалізовано три основні ролі:

- USER;
- SUPPORT;
- ADMIN.

Публічні endpoint-и системи доступні без JWT-авторизації. До них належать:

- /auth/register;
- /auth/login;
- /auth/verify;
- Swagger UI endpoint-и.

Доступ до захищених endpoint-ів дозволений лише після успішної перевірки JWT-токена. Для окремих endpoint-ів додатково виконується перевірка ролі користувача.

Наприклад:

- endpoint-и /admin/** доступні лише користувачам із роллю ADMIN;
- endpoint-и /support/** доступні користувачам із ролями SUPPORT або ADMIN;
- endpoint-и /tickets/** доступні автентифікованим користувачам.

Для реалізації рольової авторизації використовуються:

- requestMatchers у SecurityConfig;
- анотації @PreAuthorize;
- Spring Security SecurityContext.

Такий підхід дозволяє реалізувати централізовану систему контролю доступу до ресурсів серверного застосунку.

3.5 Проєктування двофакторної автентифікації

3.5.1 Архітектура 2FA

У серверному застосунку реалізовано механізм двофакторної автентифікації на основі алгоритму TOTP. Даний механізм використовується як додатковий етап перевірки користувача після успішного введення email та пароля. Для генерації одноразових кодів користувач може використовувати мобільний застосунок Google Authenticator [3], а перевірка коду виконується на стороні серверного застосунку відповідно до алгоритму TOTP [5].

Основними компонентами механізму 2FA є:

- AuthController, який приймає запити на активацію, підтвердження та вимкнення двофакторної автентифікації;
- AuthService, який виконує основну логіку генерації секретного ключа, перевірки TOTP-коду та зміни стану 2FA;
- сутність User, у якій зберігаються поля twoFactorEnabled, twoFactorSecret та pendingTwoFactorSecret;
- мобільний застосунок Google Authenticator, який генерує одноразові коди на стороні користувача.

Під час активації 2FA сервер генерує тимчасовий секретний ключ і зберігає його у полі pendingTwoFactorSecret. Після цього формується QR-код, який користувач сканує у Google Authenticator. Двофакторна автентифікація активується лише після успішного введення першого TOTP-коду.

Архітектура механізму двофакторної автентифікації наведена на рисунку 3.11.

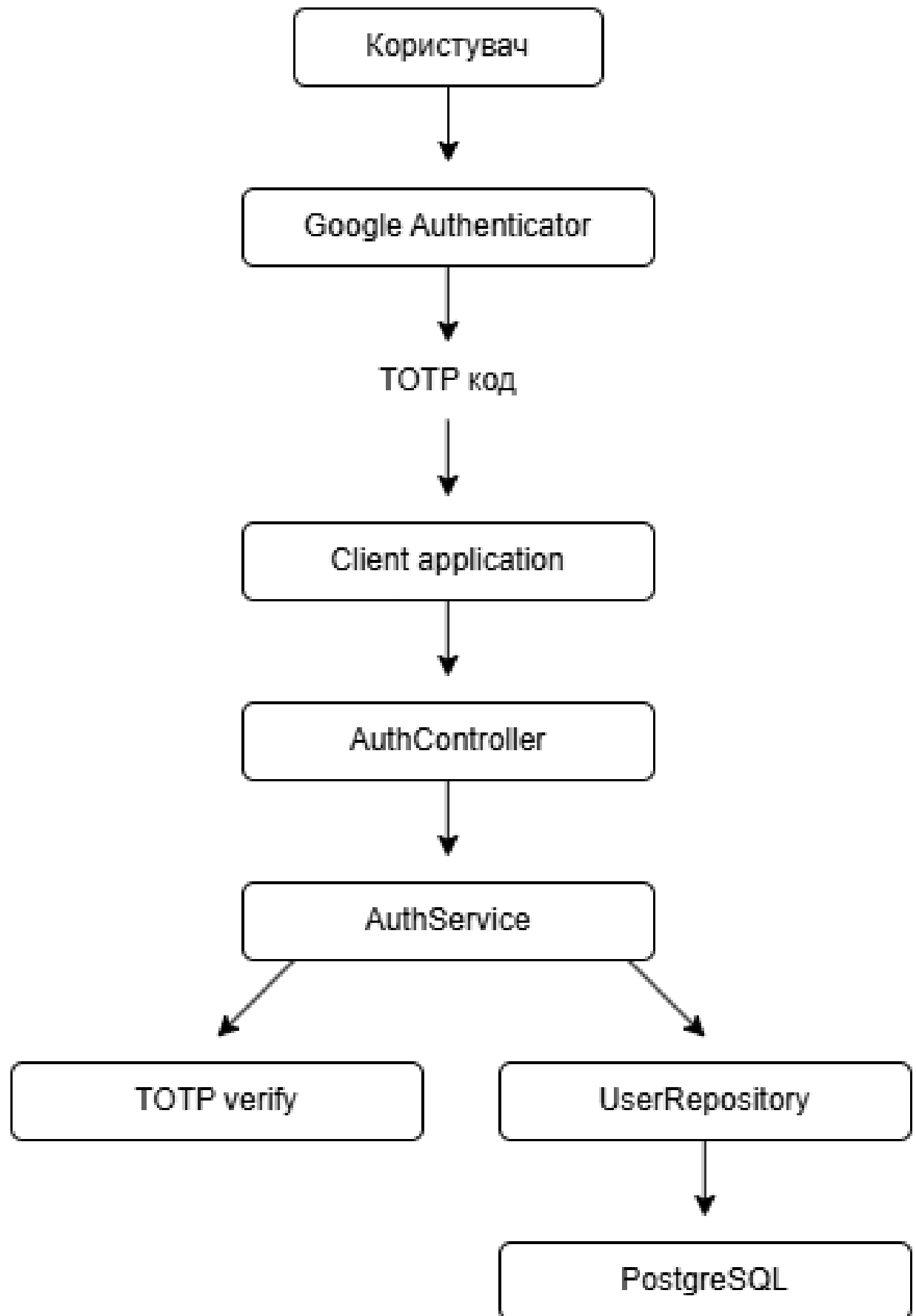


Рисунок 3.11 – Архітектура механізму двофакторної автентифікації

3.5.2 Алгоритм активації 2FA

Процес активації двофакторної автентифікації починається після того, як авторизований користувач надсилає запит на увімкнення 2FA. Система перевіряє користувача за JWT-токеном, після чого генерує секретний ключ для TOTP.

Алгоритм активації 2FA включає такі етапи:

- користувач надсилає запит на генерацію даних для 2FA;
- сервер генерує тимчасовий секретний ключ;
- секрет зберігається у полі `pendingTwoFactorSecret`;
- сервер формує QR-код для додавання облікового запису у Google Authenticator;
- користувач сканує QR-код у мобільному застосунку;
- користувач вводить перший TOTP-код;
- сервер перевіряє код;
- у разі успішної перевірки секрет переноситься у поле `twoFactorSecret`;
- поле `twoFactorEnabled` отримує значення `true`.

Використання тимчасового поля `pendingTwoFactorSecret` дозволяє не активувати 2FA до моменту успішного підтвердження користувачем. Це запобігає ситуації, коли 2FA була увімкнена, але користувач не додав обліковий запис у застосунок-генератор кодів.

Алгоритм активації двофакторної автентифікації наведений на рисунку 3.12.

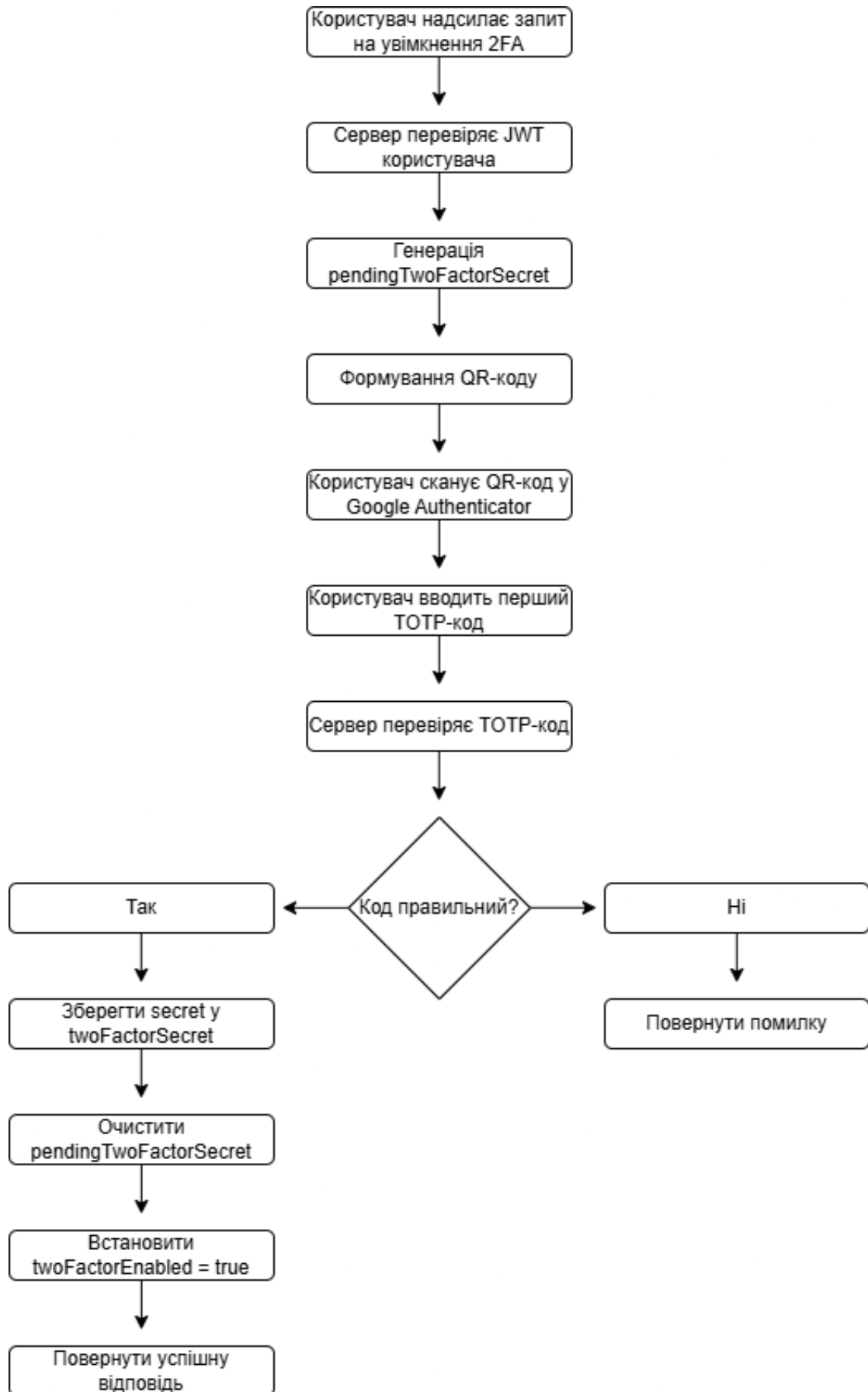


Рисунок 3.12 – Блок-схема алгоритму активації двофакторної автентифікації

3.5.3 Алгоритм перевірки TOTP-коду

Перевірка TOTP-коду виконується під час підтвердження активації 2FA, входу користувача в систему та вимкнення двофакторної автентифікації.

Під час перевірки сервер отримує код, введений користувачем, та порівнює його з кодом, згенерованим на основі секретного ключа користувача і поточного часу. Оскільки TOTP-коди мають обмежений термін дії, система приймає лише актуальний код, сформований у межах допустимого часового інтервалу [5].

У процесі перевірки TOTP-коду виконуються такі дії:

- отримання користувача з бази даних;
- отримання секретного ключа користувача;
- отримання коду, введеного користувачем;
- генерація очікуваного TOTP-коду на сервері;
- порівняння введеного та очікуваного значення;
- повернення результату перевірки.

Якщо код є правильним, система продовжує виконання відповідної операції: активує 2FA, видає access token або вимикає 2FA. Якщо код неправильний, користувач отримує повідомлення про помилку.

3.5.4 Використання 2FA під час входу

Під час входу користувача система спочатку перевіряє email та пароль. Якщо облікові дані правильні, сервер додатково перевіряє, чи активована для користувача двофакторна автентифікація.

Якщо 2FA не активована, користувач одразу отримує JWT access token. Якщо 2FA активована, сервер не видає повний токен доступу, а формує тимчасовий tempToken з типом 2FA_PENDING. Такий токен має

обмежений термін дії та використовується лише для підтвердження другого фактора.

Після цього користувач надсилає TOTP-код на endpoint підтвердження 2FA. Сервер перевіряє tempToken, отримує користувача та перевіряє введений код. Якщо код правильний, система генерує повноцінний JWT access token.

Під час входу користувача система спочатку перевіряє email та пароль. Якщо облікові дані правильні, сервер додатково перевіряє, чи активована для користувача двофакторна автентифікація.

Якщо 2FA не активована, користувач одразу отримує JWT access token. Якщо 2FA активована, сервер не видає повний токен доступу, а формує тимчасовий tempToken з типом 2FA_PENDING. Такий токен має обмежений термін дії та використовується лише для підтвердження другого фактора.

Після цього користувач надсилає TOTP-код на endpoint підтвердження 2FA. Сервер перевіряє tempToken, отримує користувача та перевіряє введений код. Якщо код правильний, система генерує повноцінний JWT access token.

Послідовність автентифікації користувача та механізм використання tempToken наведені на рисунках 3.6 та 3.7 відповідно.

3.5.5 Вимкнення двофакторної автентифікації

Для вимкнення двофакторної автентифікації користувач повинен бути авторизованим і додатково підтвердити операцію актуальним TOTP-кодом. Це необхідно для того, щоб запобігти вимкненню 2FA сторонньою особою у випадку компрометації JWT-токена.

Після отримання запиту система:

- отримує email користувача з контексту безпеки;
- знаходить користувача у базі даних;

- перевіряє, чи активована 2FA;
- перевіряє введений TOTP-код;
- у разі успіху очищує twoFactorSecret;
- встановлює twoFactorEnabled = false.

Такий підхід дозволяє зберегти контроль над важливою дією безпеки та не дозволяє вимкнути другий фактор без підтвердження через мобільний застосунок.

3.5.6 Висновки щодо проєктування 2FA

У межах серверного застосунку спроектовано механізм двофакторної автентифікації, який складається з генерації секретного ключа, формування QR-коду, підтвердження першого TOTP-коду, використання tempToken під час входу та перевірки одноразового коду перед видачею access token.

Запропонований механізм дозволяє підвищити рівень захисту облікових записів користувачів, оскільки навіть після введення правильного пароля доступ до системи не надається без додаткового підтвердження через TOTP-код.

3.6 Проєктування REST API

3.6.1 Структура REST API

Для взаємодії між клієнтською та серверною частинами застосунку використовується REST API. Передача даних між компонентами системи здійснюється через HTTP-запити із використанням JSON-формату [9].

REST API серверного застосунку побудоване відповідно до принципів REST-архітектури:

- використання HTTP-методів;
- розділення ресурсів за endpoint-ами;
- stateless-взаємодія;

- передача даних у форматі JSON;
- використання JWT для авторизації.

У серверному застосунку реалізовано окремі групи endpoint-ів:

- endpoint-и автентифікації;
- endpoint-и двофакторної автентифікації;
- endpoint-и адміністративного керування;
- endpoint-и системи підтримки користувачів.

Для доступу до захищених endpoint-ів використовується JWT access token, який передається у заголовок Authorization у форматі Bearer Token [2].

Структура REST API серверного застосунку наведена на рисунку 3.13.

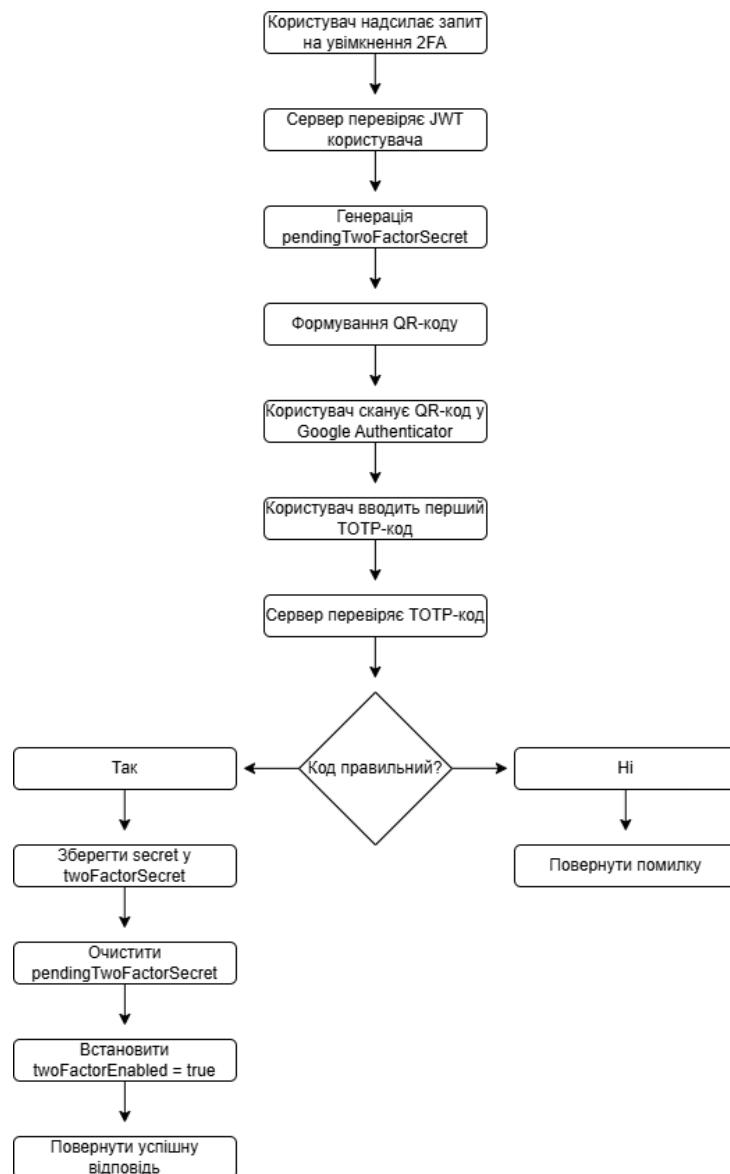


Рисунок 3.13 – Структура REST API серверного застосунку

3.6.2 Основні endpoint-и системи

Endpoint-и системи поділяються на:

- публічні endpoint-и;
- endpoint-и для авторизованих користувачів;
- адміністративні endpoint-и;
- endpoint-и підтримки користувачів.

Основні endpoint-и REST API серверного застосунку наведені у таблиці 3.1.

Таблиця 3.1 – Основні endpoint-и REST API

HTTP Method	Endpoint	Призначення
POST	/auth/register	Реєстрація користувача
POST	/auth/login	Автентифікація користувача
POST	/auth/login/verify-2fa	Підтвердження TOTP-коду
GET	/auth/profile	Отримання інформації про користувача
POST	/auth/2fa/generate	Генерація QR-коду для 2FA
POST	/auth/2fa/confirm	Підтвердження активації 2FA
POST	/auth/2fa/disable	Вимкнення двофакторної автентифікації
GET	/support/tickets	Отримання списку тікетів
POST	/tickets/create	Створення тікета підтримки
PATCH	/support/tickets/{id}	Оновлення статусу тікета
GET	/admin/users	Отримання списку користувачів
PATCH	/admin/users/{id}/role	Зміна ролі користувача

3.6.3 Формат запитів і відповідей

Всі запити до серверу передаються у форматі JSON. Приклад запиту на вхід наведено у лістингу 3.1. Приклад відповіді від сервера коли вимкнена 2FA наведено у лістингу 3.2, а коли 2FA увімкнена наведено у лістингу 3.3.

Лістинг 3.1 – Запит на автентифікацію користувача

```
{  
  "email": "user@example.com",  
  "password": "password123"  
}
```

Лістинг 3.2 – Відповідь сервера

```
{  
  "token": "jwt_token_here"  
}
```

Лістинг 3.3 – Відповідь сервера у разі використання 2FA

```
{  
  "tempToken": "temp_token_here"  
}
```

Для розмежування структури запитів та відповідей у застосунку використовуються окремі DTO-класи:

- RegisterRequest;
- LoginRequest;
- LoginResponse;
- UserResponse;
- CreateTicketRequest;
- UpdateTicketRequest.

Використання DTO дозволяє:

- відокремити внутрішню модель даних від REST API;
- підвищити безпеку застосунку;
- спростити валідацію вхідних даних;
- забезпечити стабільність структури API.

3.6.4 Обробка помилок

Для організації коректної взаємодії між клієнтом та сервером у застосунку реалізовано механізм централізованої обробки помилок.

Під час виникнення помилок сервер повертає відповідний HTTP status code та JSON-відповідь із описом помилки.

У серверному застосунку використовуються такі основні статуси:

- 200 OK – успішне виконання запиту;
- 400 Bad Request – помилка вхідних даних;
- 401 Unauthorized – відсутній або недійсний JWT-токен;
- 403 Forbidden – недостатньо прав доступу;
- 404 Not Found – ресурс не знайдено;
- 500 Internal Server Error – внутрішня помилка сервера.

Для перевірки вхідних даних використовуються механізми валідації Spring Boot та Spring Security [1]-[9].

Обробка помилок реалізована через:

- RuntimeException;
- перевірку JWT-токенів;
- перевірку ролей користувачів;
- перевірку коректності TOTP-кодів;
- перевірку існування користувачів та тікетів.

Такий підхід дозволяє забезпечити стабільну роботу REST API та коректне інформування клієнта про причини виникнення помилок.

3.7 Висновки до розділу

У третьому розділі дипломної роботи було виконано проєктування серверного програмного застосунку автентифікації користувачів. У межах розділу розроблено архітектуру системи, структуру бази даних, механізми

автентифікації та авторизації, а також REST API для взаємодії між клієнтською та серверною частинами застосунку.

Під час проектування системи використано багаторівневу архітектуру Controller–Service–Repository, що дозволило розділити компоненти застосунку за рівнями відповідальності та підвищити модульність програмного коду. Для організації захисту даних користувачів було реалізовано механізми BCrypt-хешування паролів [1], [11], JWT-авторизації [2] та двофакторної автентифікації на основі алгоритму TOTP [5].

У процесі проектування моделі даних було сформовано структуру бази даних PostgreSQL [8], визначено основні сутності системи та взаємозв'язки між ними. Для взаємодії із базою даних обрано технології Spring Data JPA та Hibernate [9]-[10].

Також у межах розділу було спроектовано:

- механізм реєстрації користувачів;
- механізм входу в систему;
- алгоритм перевірки JWT-токенів;
- систему рольової авторизації;
- механізм двофакторної автентифікації;
- захист від brute-force атак;
- REST API серверного застосунку.

Особливістю реалізованої системи є використання tempToken для організації двоетапної автентифікації користувача. Такий підхід дозволяє відокремити перевірку логіна і пароля від перевірки TOTP-коду та підвищує рівень безпеки системи.

У результаті виконаного проектування було сформовано основу для реалізації серверного програмного застосунку автентифікації користувачів, який забезпечує безпечну обробку облікових даних, рольову модель доступу та підтримку сучасних механізмів захисту інформації.

4 РЕАЛІЗАЦІЯ СЕРВЕРНОГО ЗАСТОСУНКУ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ

4.1 Реалізація серверної архітектури

4.1.1 Реалізація структури серверного проєкту

Реалізація серверного застосунку автентифікації користувачів виконана із використанням фреймворку Spring Boot, який забезпечує створення REST API, інтеграцію із базою даних PostgreSQL та підтримку сучасних механізмів безпеки [1], [9].

Серверний застосунок реалізований як окремий backend-проєкт із використанням багаторівневої архітектури Controller–Service–Repository. Такий підхід дозволяє розділити логіку застосунку за рівнями відповідальності та забезпечити модульність програмного коду.

Структура серверного проєкту організована відповідно до функціонального призначення компонентів системи. Основні класи застосунку розподілені між пакетами:

- controller – REST-контролери;
- service – бізнес-логіка системи;
- repository – взаємодія із базою даних;
- model – JPA-сутності;
- dto – DTO-об'єкти для REST API;
- config – конфігураційні класи;
- security – компоненти безпеки.

Для автоматизації процесу збірки застосунку використовується система Gradle. Основні залежності проєкту визначені у файлі build.gradle.

Структура серверного проєкту у середовищі IntelliJ IDEA наведена на рисунку 4.1.

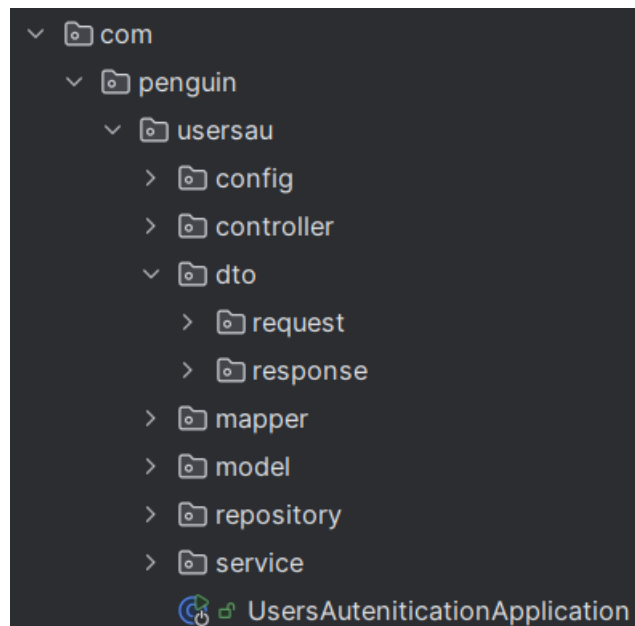


Рисунок 4.1 – Структура серверного проєкту у середовищі IntelliJ IDEA

4.1.2 Реалізація REST API для взаємодії клієнта та сервера

Для забезпечення взаємодії між клієнтською та серверною частинами застосунку у розробленій системі було реалізовано REST API. Використання архітектурного стилю REST дозволяє організувати стандартизований механізм обміну даними через HTTP-запити та забезпечує незалежність клієнтської частини від серверної реалізації.

Основною перевагою REST API є простота інтеграції, масштабованість та підтримка stateless-архітектури, при якій сервер не зберігає інформацію про стан сесії користувача між запитами. Це дозволяє зменшити навантаження на сервер та підвищити безпеку системи.

У розробленому програмному забезпеченні REST API використовується для виконання таких операцій:

- реєстрація користувачів;
- автентифікація користувачів;
- перевірка електронної пошти;
- генерація та перевірка JWT-токенів;
- робота з двофакторною автентифікацією;

- отримання даних профілю користувача.

Для передачі даних між клієнтом і сервером використовується формат JSON, який є платформонезалежним та широко застосовується у сучасних вебзастосунках. Усі запити до серверу виконуються за допомогою HTTP-методів POST та GET залежно від типу операції.

Доступ до захищених ресурсів реалізовано за допомогою JWT-токенів. Після успішної автентифікації користувач отримує токен доступу, який надалі передається у заголовок Authorization за схемою Bearer Token. Сервер перевіряє валідність токена перед наданням доступу до захищених endpoint-ів.

Для реалізації REST API було використано фреймворк Spring Boot, який забезпечує зручний механізм створення контролерів, маршрутизації HTTP-запитів та автоматичної серіалізації JSON-об'єктів. Основні endpoint-и системи реалізовані у контролері автентифікації, що відповідає за обробку запитів користувачів.

Архітектура REST API була спроектована з урахуванням можливості подальшого розширення функціональності системи. Це дозволяє у майбутньому додавати нові endpoint-и та інтегрувати зовнішні сервіси без значних змін у структурі застосунку.

4.1.3 Реалізація сервісного рівня

Сервісний рівень реалізує основну бізнес-логіку серверного застосунку. Усі операції обробки даних виконуються через класи сервісів, позначені анотацією @Service.

У серверному застосунку реалізовано такі основні сервіси:

- AuthService;
- TicketService;
- AuditService.

Сервісний рівень виконує:

- реєстрацію користувачів;
- автентифікацію;
- генерацію JWT-токенів;
- перевірку TOTP-кодів;
- блокування користувачів;
- керування ролями;
- роботу із тікетами підтримки.

Взаємодія між компонентами системи реалізована через механізм dependency injection Spring Framework [1].

Приклад реалізації сервісного класу наведений у тексті файлу додаток Б.1, який демонструє реалізацію бізнес-логіки автентифікації користувача із використанням BCrypt та JWT.

4.1.4 Реалізація взаємодії з базою даних

Для взаємодії із базою даних PostgreSQL у серверному застосунку використовується технологія Spring Data JPA та ORM-фреймворк Hibernate [8]-[9].

Модель даних реалізована через JPA-сутності із використанням анотацій:

- @Entity;
- @Table;
- @Id;
- @GeneratedValue;
- @Column.

Для доступу до даних використовуються repository-інтерфейси, які успадковуються від JpaRepository.

Використання Spring Data JPA дозволило реалізувати операції створення, читання, оновлення та видалення даних без необхідності ручного написання SQL-запитів.

Підключення серверного застосунку до PostgreSQL виконується через файл конфігурації `application.properties`.

4.1.5 Реалізація конфігурації Spring Security

Для організації безпеки REST API у серверному застосунку використовується Spring Security [1].

Конфігурація механізмів безпеки реалізована у класі `SecurityConfig`, який визначає:

- правила доступу до endpoint-ів;
- інтеграцію JWT-фільтра;
- stateless-режим роботи;
- рольову авторизацію користувачів.

Для перевірки JWT-токенів реалізовано окремий компонент `JwtAuthenticationFilter`, який перехоплює HTTP-запити та перевіряє токен користувача.

Приклад конфігурації Spring Security наведений у тексті файлу додаток Б.2 у методі `filterChain`, він демонструє реалізацію правил доступу до REST API та інтеграцію JWT-фільтра у Spring Security.

Використання Spring Security дозволило реалізувати централізований механізм авторизації та захисту endpoint-ів серверного застосунку.

4.2 Реалізація моделі даних

4.2.1 Реалізація структури бази даних

Для зберігання даних серверного застосунку використовується реляційна база даних PostgreSQL [8]. Взаємодія із базою даних реалізована через технологію Spring Data JPA та ORM-фреймворк Hibernate [9]-[10].

Структура бази даних включає основні таблиці:

- `users`;

- tickets;
- audit_logs.

Таблиця users використовується для зберігання облікових записів користувачів, таблиця tickets – для системи підтримки користувачів, а таблиця audit_logs – для зберігання журналу подій безпеки.

Для ідентифікації користувачів використовується UUID, що дозволяє уникнути прогнозованих числових ідентифікаторів та підвищує рівень безпеки системи.

Структура бази даних серверного застосунку була спроектована у третьому розділі дипломної роботи та наведена на рисунку 3.4. Реалізація таблиць та зв'язків виконана у PostgreSQL із використанням Hibernate ORM та Spring Data JPA [8], [10].

4.2.2 Реалізація сутності User

Основною сутністю серверного застосунку є сутність User, яка використовується для зберігання даних користувачів та параметрів автентифікації.

Сутність User реалізована із використанням JPA-анотацій @Entity та @Table. Поля сутності відповідають структурі таблиці users у базі даних PostgreSQL.

До складу сутності входять такі основні поля:

- uuid – унікальний ідентифікатор користувача;
- email – адреса електронної пошти;
- password – BCrypt-хеш пароля;
- role – роль користувача;
- twoFactorEnabled – стан двофакторної автентифікації;
- twoFactorSecret – секретний ключ TOTP;
- pendingTwoFactorSecret – тимчасовий секрет для активації 2FA;

- failedLoginAttempts – кількість невдалих спроб входу;
- lockedUntil – час блокування облікового запису;
- emailVerified – стан підтвердження електронної пошти;
- createdAt – дата створення облікового запису;
- lastLogin – дата останнього входу.

Реалізація сутності User наведений у тексті файлу додаток Б.3, яка демонструє реалізацію основних полів користувача та використання JPA-анотацій для зв'язку моделі із таблицею бази даних.

4.2.3 Реалізація системи ролей користувачів

Для розмежування доступу до функціональних можливостей серверного застосунку реалізована рольова модель доступу RBAC (Role-Based Access Control) [1].

У системі реалізовано такі ролі:

- USER;
- SUPPORT;
- ADMIN.

Роль користувача зберігається у полі role таблиці users та використовується під час авторизації endpoint-ів через Spring Security.

Користувачі з роллю:

- USER можуть використовувати функції автентифікації та створювати тікети підтримки;
- SUPPORT можуть переглядати та обробляти тікети;
- ADMIN мають повний доступ до адміністративних endpoint-ів.

Перевірка ролей користувачів виконується через:

- requestMatchers;
- hasRole;
- hasAnyRole;
- @PreAuthorize.

Такий підхід дозволяє реалізувати централізований контроль доступу до ресурсів серверного застосунку.

4.2.4 Реалізація захисту даних користувачів

Для організації безпеки облікових даних користувачів у серверному застосунку використовується BCrypt-хешування паролів [1].

Під час реєстрації користувача пароль не зберігається у відкритому вигляді. Перед збереженням у базу даних пароль хешується через BCryptPasswordEncoder. Реалізація BCrypt-хешування наведена у тексті файлу додаток Б.1 у методі register.

Під час входу користувача сервер порівнює введений пароль із BCrypt-хешем, збереженим у базі даних.

Крім захисту паролів, у системі реалізовано:

- перевірку JWT-токенів;
- двофакторну автентифікацію;
- блокування облікових записів;
- підтвердження електронної пошти;
- аудит подій безпеки.

Такий підхід дозволяє підвищити рівень захисту персональних даних користувачів та REST API серверного застосунку.

4.2.5 Реалізація взаємодії із PostgreSQL через JPA

Для роботи із базою даних у серверному застосунку використовується Spring Data JPA та Hibernate ORM [9]-[10].

Репозиторії реалізовані через інтерфейси, які успадковуються від JpaRepository. Це дозволяє автоматично отримати реалізацію CRUD-операцій без необхідності ручного написання SQL-запитів.

Приклад реалізації репозиторію наведений у тексті файлу додаток Б.4 та демонструє реалізацію репозиторію для роботи із сутністю Ticket.

Використання Spring Data JPA дозволило:

- спростити взаємодію із базою даних;
- зменшити кількість SQL-коду;
- автоматизувати ORM-відображення сутностей;
- забезпечити інтеграцію із Hibernate ORM.

Конфігурація підключення до PostgreSQL виконується через файл `application.properties`, у якому визначаються параметри бази даних та налаштування Hibernate.

4.3 Реалізація механізму автентифікації

4.3.1 Реалізація реєстрації користувача

Реєстрація користувача реалізована через REST endpoint `/auth/register`, який приймає HTTP POST-запит із даними нового користувача.

Під час реєстрації система виконує:

- перевірку коректності email;
- перевірку унікальності користувача;
- BCrypt-хешування пароля;
- генерацію коду підтвердження електронної пошти;
- створення нового облікового запису;
- збереження користувача у базі даних;
- надсилання листа підтвердження.

Для реалізації реєстрації використовується DTO-об'єкт `RegisterRequest`, який містить email та пароль користувача.

Основна бізнес-логіка реєстрації реалізована у сервісі `AuthService` та наведена у тексті файлу додаток Б.1, а алгоритм наведений на рисунку 3.5.

4.3.2 Реалізація автентифікації користувача

Автентифікація користувача реалізована через endpoint `/auth/login`. Під час входу система перевіряє email користувача, BCrypt-хеш пароля та стан облікового запису.

У процесі автентифікації виконуються:

- пошук користувача у базі даних;
- перевірка блокування облікового запису;
- перевірка BCrypt-хешу пароля;
- перевірка стану 2FA;
- генерація JWT access token або tempToken.

Для порівняння паролів використовується BCryptPasswordEncoder, який виконує порівняння введеного пароля із BCrypt-хешем, збереженим у базі даних [1].

Послідовність автентифікації користувача була наведена у третьому розділі дипломної роботи на рисунку 3.3.

4.3.3 Реалізація механізму tempToken

Однією з особливостей реалізованої системи є використання тимчасового JWT-токена tempToken, який застосовується під час двофакторної автентифікації.

Після успішної перевірки email та пароля користувач не отримує повноцінний access token, якщо для нього активована 2FA. Замість цього сервер генерує тимчасовий токен із типом 2FA_PENDING.

TempToken використовується виключно для:

- підтвердження TOTP-коду;
- завершення процесу автентифікації.

Тимчасовий токен має:

- окремий тип;

- обмежений час життя;
- неможливість доступу до захищених endpoint-ів.

Генерацію tempToken наведено у тексті файлу додаток Б.5 у методі genTempToken. Механізм використання tempToken був спроектований у третьому розділі та наведений на рисунку 3.7.

4.3.4 Реалізація захисту від brute-force атак

Для захисту облікових записів користувачів від brute-force атак у серверному застосунку реалізовано механізм обмеження кількості невдалих спроб входу [4].

У моделі користувача зберігаються:

- failedLoginAttempts;
- lockedUntil.

Після кожної невдалої спроби входу система:

- збільшує лічильник невдалих спроб;
- перевіряє перевищення допустимого обмеження;
- блокує обліковий запис на визначений час.

Реалізації блокування наведена у тексті файлу додаток Б.6 у методі lockUser. Під час входу система перевіряє, чи не заблокований користувач, і у випадку активного блокування повертає повідомлення про помилку.

Алгоритм захисту від brute-force атак був наведений у третьому розділі дипломної роботи на рисунку 3.8.

4.3.5 Реалізація підтвердження електронної пошти

Для підтвердження адреси електронної пошти у серверному застосунку реалізовано механізм email verification.

Під час реєстрації система:

- генерує код підтвердження;

- надсилає лист користувачу;
- зберігає код та час його завершення;
- очікує підтвердження через окремий endpoint.

Приклад листа підтвердження електронної пошти наведений на рисунку 4.2.

Після переходу користувача за посиланням підтвердження система перевіряє:

- email користувача;
- код підтвердження;
- термін дії коду.

У разі успішної перевірки поле emailVerified отримує значення true.

Для надсилання електронних листів використовується Spring Boot Starter Mail [9].

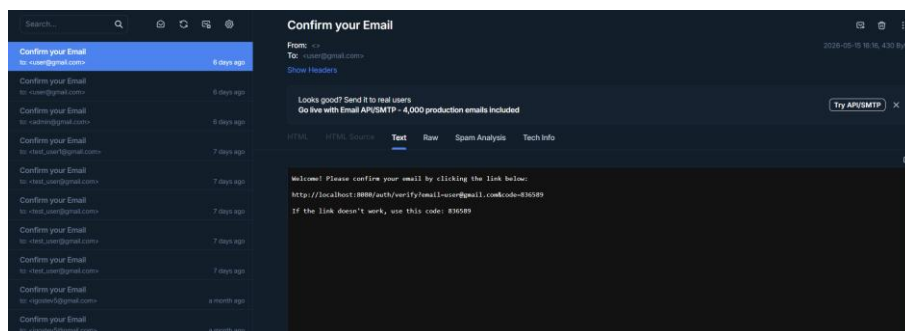


Рисунок 4.2 – Лист підтвердження електронної пошти

4.3.6 Висновки щодо реалізації механізму автентифікації

У межах серверного застосунку реалізовано механізм автентифікації користувачів із підтримкою:

- BCrypt-хешування паролів;
- JWT access token;
- tempToken;
- двофакторної автентифікації;
- підтвердження електронної пошти;

– захисту від brute-force атак.

Реалізований механізм дозволяє забезпечити безпечну автентифікацію користувачів та відповідає сучасним вимогам до захисту REST API та облікових записів користувачів [1]-[2], [4].

4.4 Реалізація JWT-авторизації

4.4.1 Реалізація генерації JWT-токена

Для реалізації авторизації користувачів у серверному застосунку використано механізм JSON Web Token [2]. Генерація токенів виконується у класі JwtUtils, який відповідає за створення access token та тимчасового tempToken для проходження двофакторної автентифікації.

Access token створюється після успішної автентифікації користувача або після підтвердження TOTP-коду, якщо для користувача активована двофакторна автентифікація. У токен додаються email користувача, роль, тип токена, час створення та час завершення дії.

Генерацію access token наведено у тексті файлу додаток Б.5 у методі genToken, цей метод демонструє створення JWT-токена із зазначенням ролі користувача та типу токена. Значення type = ACCESS використовується для відокремлення повноцінного токена доступу від тимчасового токена двофакторної автентифікації.

4.4.2 Реалізація перевірки JWT-токена

Перевірка JWT-токена виконується перед наданням доступу до захищених endpoint-ів. Для цього у системі реалізовано методи отримання claims, перевірки терміну дії токена та перевірки його типу наведено у тексті файлу додаток Б.5.

Метод `isTokenValid` перевіряє коректність токена та термін його дії. Метод `isAccessToken` додатково перевіряє тип токена, що дозволяє не допустити використання `tempToken` для доступу до захищених ресурсів.

4.4.3 Реалізація `JwtAuthenticationFilter`

Для інтеграції JWT-авторизації із Spring Security використано фільтр `JwtAuthenticationFilter`. Він перехоплює HTTP-запити, перевіряє наявність заголовка `Authorization`, витягує JWT-токен, перевіряє його коректність і формує контекст безпеки користувача [1].

У тексті файлу додаток Б.7 реалізовано перевірку JWT-токена та створення об'єкта автентифікації. Роль користувача додається у форматі `ROLE_`, що необхідно для коректної роботи методів `hasRole` та `hasAnyRole` у Spring Security.

Послідовність перевірки JWT-токена під час обробки HTTP-запиту була спроектована у третьому розділі та наведена на рисунку 3.10.

4.4.4 Реалізація рольового доступу до endpoint-ів

Для розмежування доступу у серверному застосунку реалізовано рольову модель. У системі використовуються ролі `USER`, `SUPPORT` та `ADMIN`.

Правила доступу налаштовані у класі `SecurityConfig`. Публічні endpoint-и доступні без авторизації, тоді як захищені endpoint-и потребують JWT-токена. Для адміністративних та support endpoint-ів додатково перевіряється роль користувача.

Реалізація правил доступу наведена у тексті файлу додаток Б.2, ця конфігурація забезпечує розмежування доступу до REST API залежно від ролі користувача. Наприклад, endpoint-и `/admin/**` доступні лише

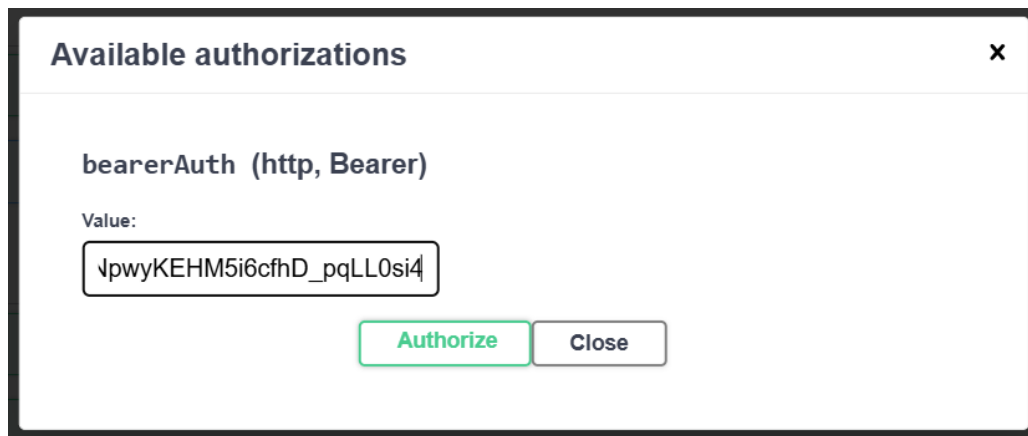


Рисунок 4.4 – Авторизація JWT-токена у Swagger UI

4.4.6 Висновки щодо реалізації JWT-авторизації

У серверному застосунку реалізовано механізм JWT-авторизації, який забезпечує безстанну перевірку користувачів та захист REST API. Реалізація включає генерацію access token, перевірку токена у JwtAuthenticationFilter, формування SecurityContext та рольове розмежування доступу до endpoint-ів.

Використання JWT у поєднанні зі Spring Security дозволило реалізувати централізований механізм авторизації користувачів, підтримку ролей USER, SUPPORT і ADMIN, а також захист адміністративних і support-функцій застосунку [1]-[2].

4.5 Реалізація двофакторної автентифікації

4.5.1 Реалізація генерації секретного ключа 2FA

Для реалізації двофакторної автентифікації у серверному застосунку використовується алгоритм TOTP (Time-Based One-Time Password) [5]. Генерація одноразових кодів виконується через мобільний застосунок Google Authenticator [3].

Під час активації 2FA сервер генерує секретний ключ, який використовується для формування TOTP-кодів. До моменту підтвердження першого коду секрет зберігається у полі pendingTwoFactorSecret.

Генерація секретного ключа виконується у сервісі AuthService, код якого наведений у тексті файлу додаток Б.1. Після генерації секрет використовується для формування QR-коду, який користувач додає у Google Authenticator.

4.5.2 Реалізація генерації QR-коду

Для спрощення підключення двофакторної автентифікації у серверному застосунку реалізовано механізм генерації QR-коду. Приклад QR-коду для підключення двофакторної автентифікації наведений на рисунку 4.5.

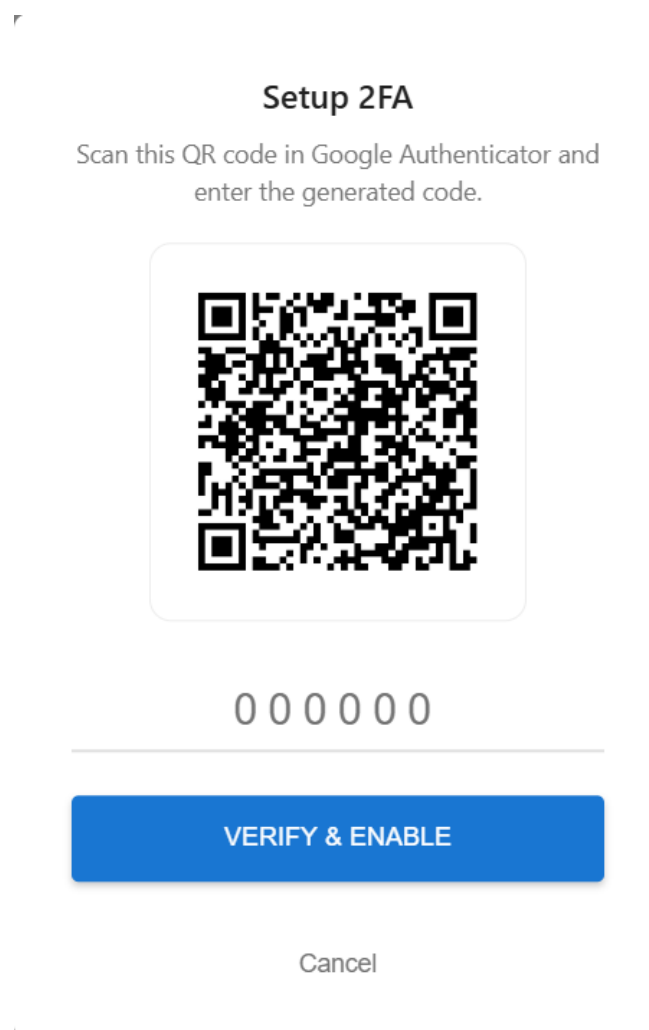


Рисунок 4.5 – QR-код для підключення двофакторної автентифікації

QR-код містить:

- email користувача;
- назву застосунку;
- секретний ключ TOTP.

Після генерації QR-код повертається клієнту у вигляді Base64-зображення та відображається у вебінтерфейсі.

Фрагмент реалізації endpoint-а генерації QR-коду наведений у тексті файлу додаток Б.8 у методі generate2FA.

4.5.3 Реалізація перевірки TOTP-коду

Після сканування QR-коду користувач вводить одноразовий TOTP-код для підтвердження активації двофакторної автентифікації.

Перевірка TOTP-коду виконується на сервері через порівняння введеного коду із кодом, сформованим на основі секретного ключа користувача [5].

У разі успішної перевірки:

- pendingTwoFactorSecret переноситься у twoFactorSecret;
- поле twoFactorEnabled отримує значення true;
- двофакторна автентифікація активується.

Алгоритм активації двофакторної автентифікації був спроектований у третьому розділі та наведений на рисунку 3.12.

4.5.4 Реалізація входу з використанням 2FA

Після успішної перевірки email та пароля система перевіряє, чи активована двофакторна автентифікація для користувача.

Якщо 2FA вимкнена, сервер одразу генерує JWT access token. Якщо 2FA активована, сервер генерує тимчасовий tempToken із типом 2FA_PENDING.

Після цього користувач повинен:

- ввести TOTP-код;
- надіслати tempToken;
- пройти підтвердження другого фактора.

Лише після успішної перевірки TOTP-коду система генерує повноцінний JWT access token.

Механізм використання tempToken був спроектований у третьому розділі дипломної роботи та наведений на рисунку 3.7.

4.5.5 Реалізація вимкнення двофакторної автентифікації

Для вимкнення двофакторної автентифікації користувач повинен бути авторизованим та додатково підтвердити свою особу через актуальний TOTP-код.

Після отримання запиту система:

- отримує користувача із SecurityContext;
- перевіряє TOTP-код;
- очищує twoFactorSecret;
- встановлює twoFactorEnabled = false.

4.5.6 Реалізація вимкнення двофакторної автентифікації

Для генерації одноразових кодів у серверному застосунку використовується сумісність із застосунком Google Authenticator [3].

Система формує спеціальний URI у форматі:
otppath://totp/AppName:user@email.com?secret=SECRET

На основі цього URI створюється QR-код, який користувач може сканувати через:

- Google Authenticator;
- Microsoft Authenticator;

- Authy;
- інші TOTP-застосунки.

Такий підхід забезпечує:

- сумісність із популярними мобільними застосунками;
- автономну генерацію кодів;
- відсутність необхідності SMS-підтвердження.

Приклад роботи двофакторної автентифікації через Google Authenticator наведений на рисунку 4.6.

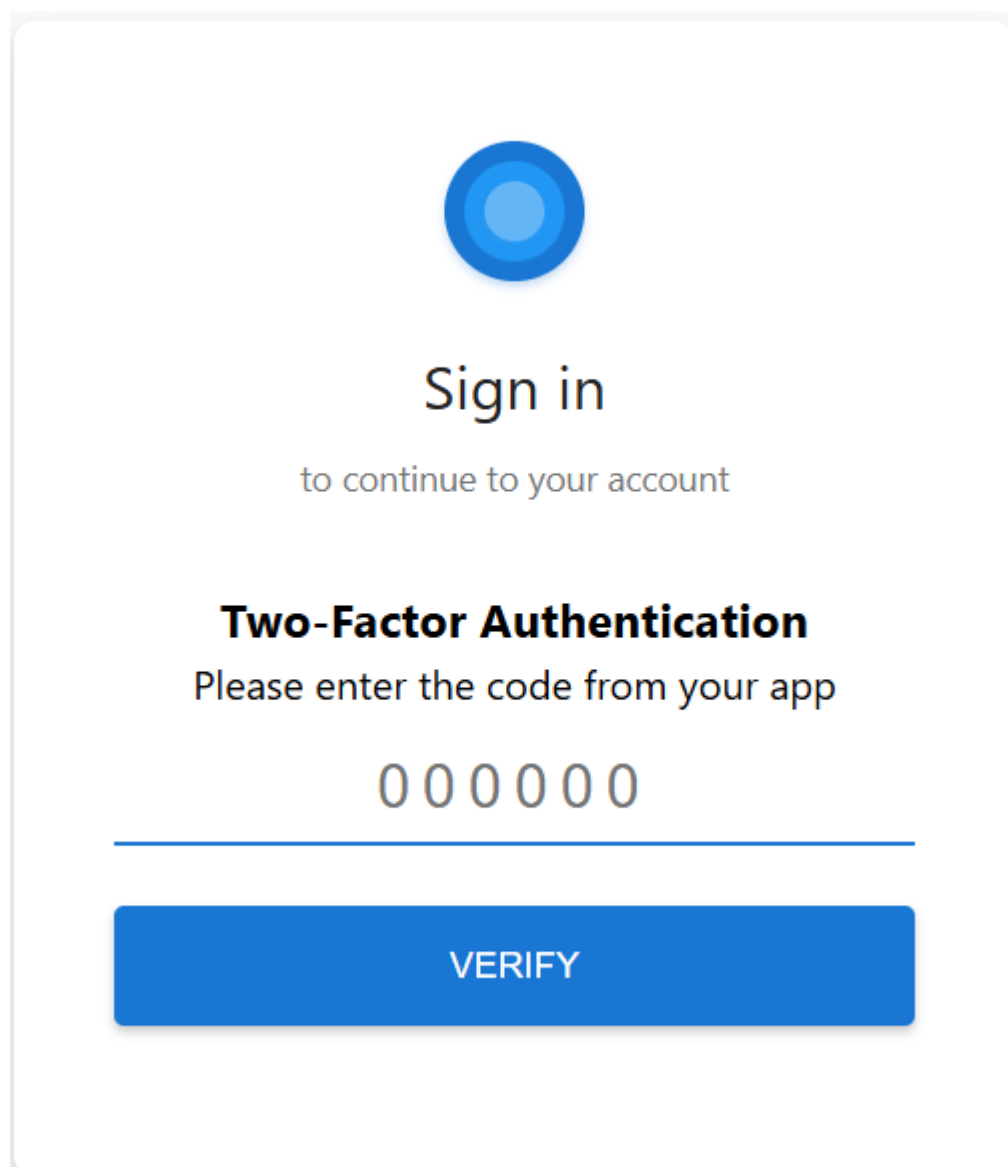


Рисунок 4.6 – Генерація TOTP-кодів у Google Authenticator

4.5.7 Висновки щодо реалізації двофакторної автентифікації

У серверному застосунку реалізовано механізм двофакторної автентифікації на основі алгоритму TOTP [5]. Реалізація включає генерацію секретного ключа, створення QR-коду, перевірку одноразових кодів та інтеграцію із Google Authenticator [3].

Для підвищення безпеки входу використовується tempToken, який дозволяє відокремити перевірку логіна та пароля від підтвердження другого фактора автентифікації.

Реалізований механізм 2FA забезпечує додатковий рівень захисту облікових записів користувачів та відповідає сучасним вимогам безпеки вебзастосунків [4]-[5].

4.6 Реалізація системи ролей та контролю доступу

4.6.1 Реалізація рольової моделі доступу

Для розмежування доступу до функціональних можливостей серверного застосунку реалізовано рольову модель RBAC (Role-Based Access Control) [1].

У системі визначено три основні ролі:

- USER;
- SUPPORT;
- ADMIN.

Роль користувача зберігається у полі role сутності User та додається до JWT-токена під час автентифікації користувача.

Роль користувачу додається за допомогою методу JwtBuilder#claim. Після перевірки JWT-токена роль користувача додається до SecurityContext, що дозволяє Spring Security виконувати перевірку прав доступу до endpoint-ів.

4.6.2 Реалізація авторизації endpoint-ів

Для контролю доступу до REST API використовується Spring Security [1]. Авторизація endpoint-ів виконується через конфігураційний клас SecurityConfig.

У серверному застосунку реалізовано:

- публічні endpoint-и;
- endpoint-и для авторизованих користувачів;
- адміністративні endpoint-и;
- support endpoint-и.

Публічні endpoint-и доступні без JWT-токена:

- /auth/register;
- /auth/login;
- /auth/login/verify-2fa;
- /auth/verify.

Для інших endpoint-ів необхідна авторизація через JWT access token. Реалізація правил доступу наведена у тексті файлу додаток Б.2, у методі filterChain.

Наведена конфігурація забезпечує:

- доступ до /admin/** лише для адміністраторів;
- доступ до /support/** для SUPPORT та ADMIN;
- доступ до /tickets/** для авторизованих користувачів.

4.6.3 Реалізація SecurityContext

Після успішної перевірки JWT-токена система формує SecurityContext, який містить інформацію про поточного користувача та його роль.

Формування контексту безпеки виконується у тексті файлу додаток Б.7 у методі doFilterInternal.

Після створення SecurityContext Spring Security автоматично використовує інформацію про роль користувача під час перевірки доступу до endpoint-ів.

Послідовність перевірки JWT-токена та створення контексту безпеки була наведена у третьому розділі дипломної роботи на рисунку 3.10.

4.6.4 Реалізація доступу до адміністративних функцій

Для адміністраторів серверного застосунку реалізовано окремі endpoint-и адміністративного керування.

Адміністративні функції включають:

- перегляд списку користувачів;
- зміну ролей користувачів;
- доступ до журналу подій;
- керування системою підтримки.

Доступ до адміністративних endpoint-ів дозволений лише користувачам із роллю ADMIN.

У випадку спроби доступу без необхідної ролі сервер повертає HTTP-відповідь 403 Forbidden.

Приклад відповіді сервера при недостатніх правах доступу наведений на рисунку 4.7.

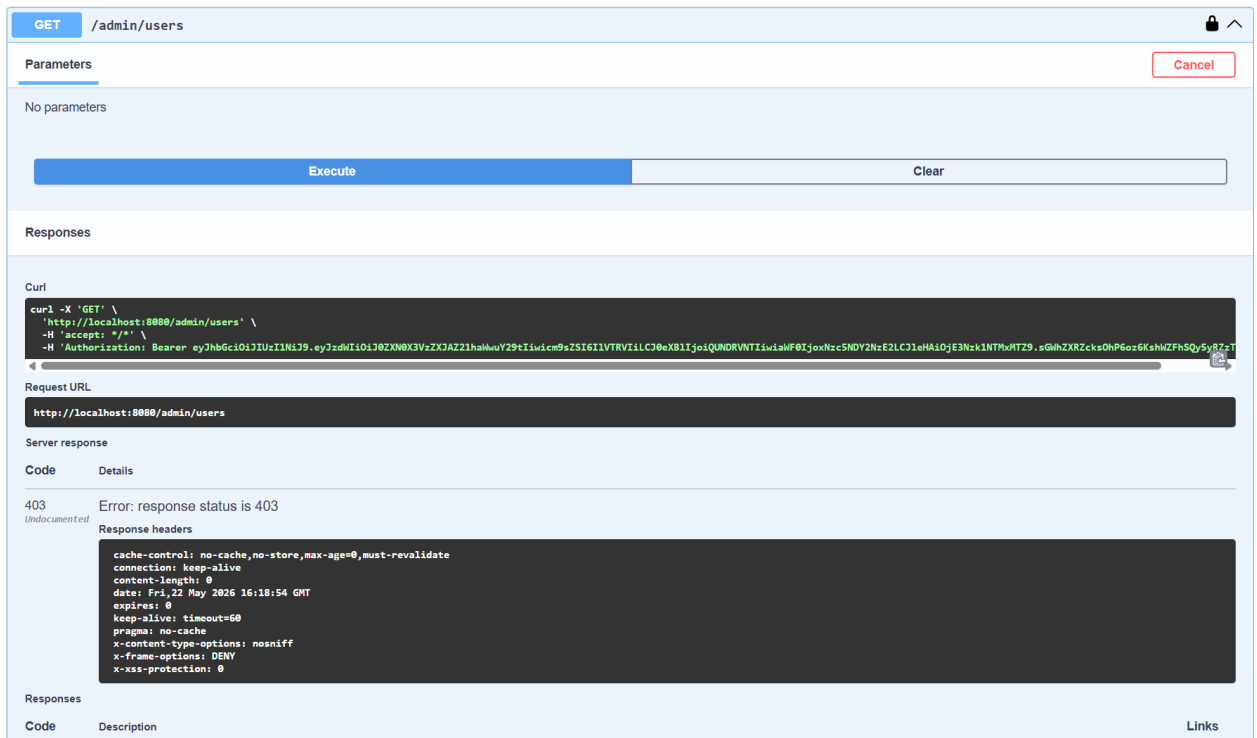


Рисунок 4.7 – Помилка 403 Forbidden при недостатніх правах доступу

4.6.5 Реалізація доступу до системи підтримки

Для користувачів із ролями SUPPORT та ADMIN реалізовано окремі endpoint-и системи підтримки користувачів.

Support endpoint-и дозволяють:

- переглядати список тікетів;
- змінювати статус тікетів;
- обробляти звернення користувачів.

Реалізація support controller наведена у тексті файлу додаток Б.9. Доступ до support endpoint-ів дозволений лише користувачам із ролями SUPPORT та ADMIN.

4.6.6 Висновки щодо реалізації системи ролей

У серверному застосунку реалізовано систему рольового контролю доступу на основі Spring Security [1].

Система підтримує ролі:

- USER;
- SUPPORT;
- ADMIN.

Для захисту REST API використовується перевірка JWT-токена, формування SecurityContext та механізми hasRole і hasAnyRole.

Реалізована система дозволяє:

- обмежити доступ до адміністративних функцій;
- розділити права користувачів;
- захистити support endpoint-и;
- забезпечити централізований контроль доступу до REST API

серверного застосунку.

4.7 Реалізація системи підтримки користувачів

4.7.1 Реалізація моделі тікетів підтримки

Для організації взаємодії користувачів із адміністрацією системи у серверному застосунку реалізовано систему підтримки користувачів на основі тікетів.

Кожен тікет містить:

- унікальний ідентифікатор;
- автора звернення;
- тему звернення;
- текст повідомлення;
- статус тікета;
- дату створення.

Для реалізації системи підтримки створено JPA-сутність Ticket, яка відповідає таблиці tickets у базі даних PostgreSQL.

Реалізація сутності Ticket наведена у тексті файлу додаток Б.10. Для ідентифікації тікетів використовується числовий ідентифікатор типу Long, який автоматично генерується базою даних.

4.7.2 Реалізація створення тікетів користувачами

Для створення звернень користувачів реалізовано endpoint `/tickets/create`, доступний лише авторизованим користувачам.

Приклад створення тікета через Swagger UI наведений на рисунку 4.8.

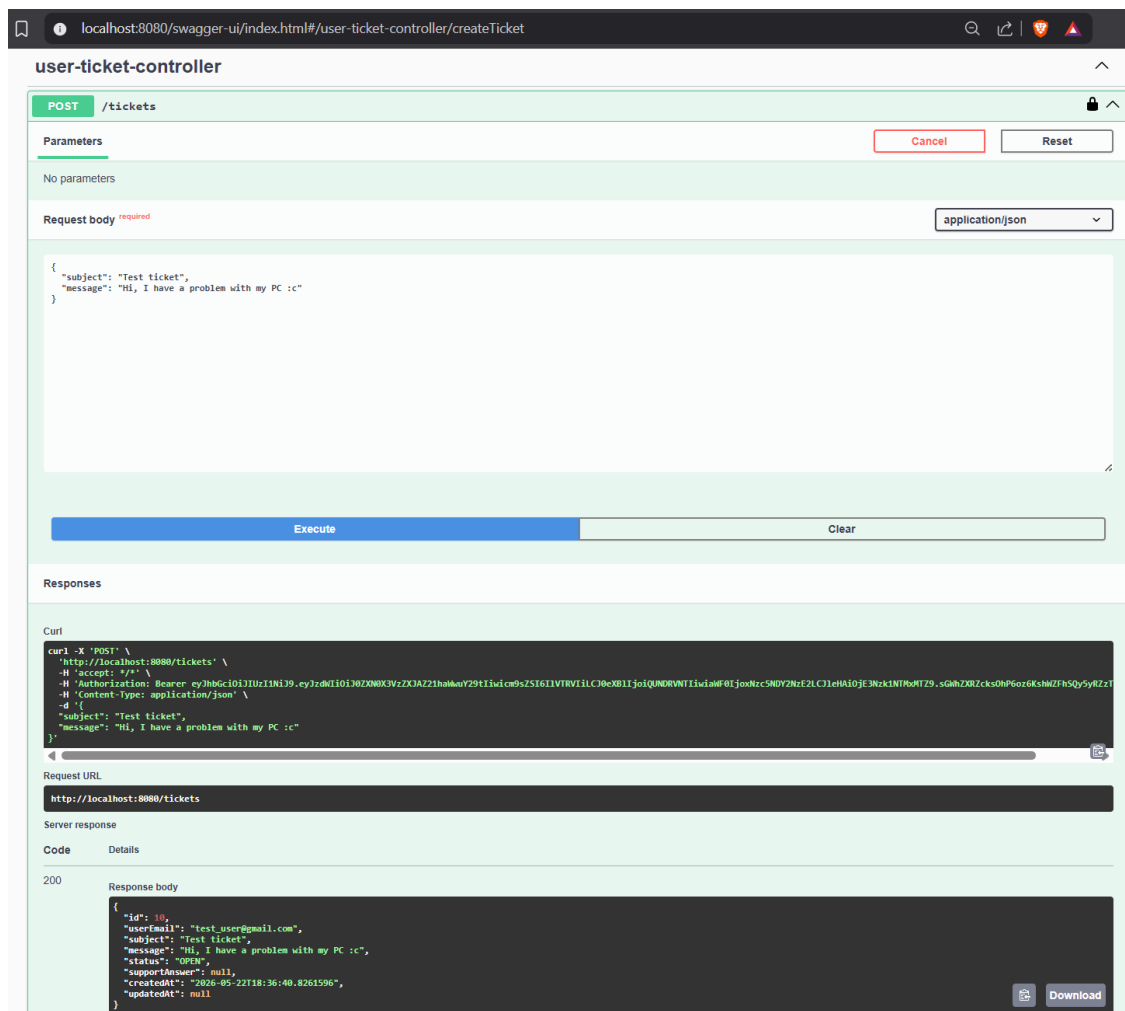


Рисунок 4.8 – Створення тікета через Swagger UI

Під час створення тікета система:

- отримує поточного користувача із `SecurityContext`;

- формує новий об'єкт Ticket;
- встановлює початковий статус;
- зберігає тикет у базі даних.

Після створення тикета користувач отримує повідомлення про успішне створення звернення.

4.7.3 Реалізація статусів тикетів

Для відображення стану звернень у системі реалізовано механізм статусів тикетів.

У серверному застосунку використовуються такі статуси:

- OPEN;
- IN_PROGRESS;
- CLOSED.

Статус тикета зберігається у полі status та реалізований через enum TicketStatus. Після створення тикета система автоматично встановлює статус OPEN. Під час обробки звернення support-користувачі можуть змінювати статус тикета.

4.7.4 Реалізація support endpoint-ів

Для користувачів із ролями SUPPORT та ADMIN реалізовано окремі endpoint-и для адміністрування тикетів.

Support endpoint-и дозволяють:

- отримувати список усіх тикетів;
- змінювати статус звернень;
- обробляти запити користувачів.

Доступ до support endpoint-ів обмежений через Spring Security та дозволений лише користувачам із ролями SUPPORT та ADMIN.

4.7.5 Реалізація сервісу роботи із тікетами

Основна бізнес-логіка системи підтримки реалізована у сервісі TicketService.

Сервіс виконує:

- створення тікетів;
- зміну статусів;
- отримання списку звернень;
- перевірку існування тікетів;
- взаємодію із базою даних через TicketRepository.

Реалізація сервісного рівня наведена у тексті файлу додаток Б.11. Використання окремого сервісного рівня дозволило відокремити бізнес-логіку від REST-контролерів та забезпечити модульність програмного коду.

4.7.6 Реалізація доступу до системи підтримки

Доступ до створення тікетів мають лише авторизовані користувачі. Для адміністрування тікетів необхідна роль SUPPORT або ADMIN.

Обмеження доступу наведено у тексті файлу додаток Б.2, у методі filterChain.

Наведена конфігурація дозволяє:

- створювати тікети лише авторизованим користувачам;
- обробляти звернення лише support-користувачам та адміністраторам.

4.7.7 Висновки щодо реалізації системи підтримки

У серверному застосунку реалізовано систему підтримки користувачів на основі тікетів. Система дозволяє користувачам створювати звернення, а support-користувачам – переглядати та обробляти їх.

Реалізація включає:

- модель тікетів;
- систему статусів;
- REST API для роботи із тікетами;
- support endpoint-и;
- рольове розмежування доступу.

Використання Spring Boot, Spring Security та Spring Data JPA дозволило реалізувати масштабовану систему підтримки користувачів із централізованим контролем доступу та інтеграцією із REST API серверного застосунку [1], [9]-[10].

4.8 Тестування REST API через Swagger UI

4.8.1 Реалізація інтеграції Swagger UI

Для документування та тестування REST API у серверному застосунку використовується Swagger UI, реалізований через бібліотеку SpringDoc OpenAPI [11]. Swagger UI дозволяє автоматично формувати інтерактивну документацію API на основі REST-контролерів та HTTP endpoint-ів.

Після запуску серверного застосунку Swagger UI стає доступним через endpoint: <http://localhost:8080/swagger-ui/index.html>.

Загальний вигляд Swagger UI серверного застосунку наведений на рисунку 4.9.

Swagger автоматично аналізує REST-контролери застосунку та формує:

- список endpoint-ів;
- HTTP-методи;
- параметри запитів;
- JSON-структури request та response;
- HTTP status codes.

Інтеграція Swagger UI значно спрощує процес тестування серверного застосунку, оскільки дозволяє виконувати HTTP-запити без використання сторонніх REST-клієнтів.

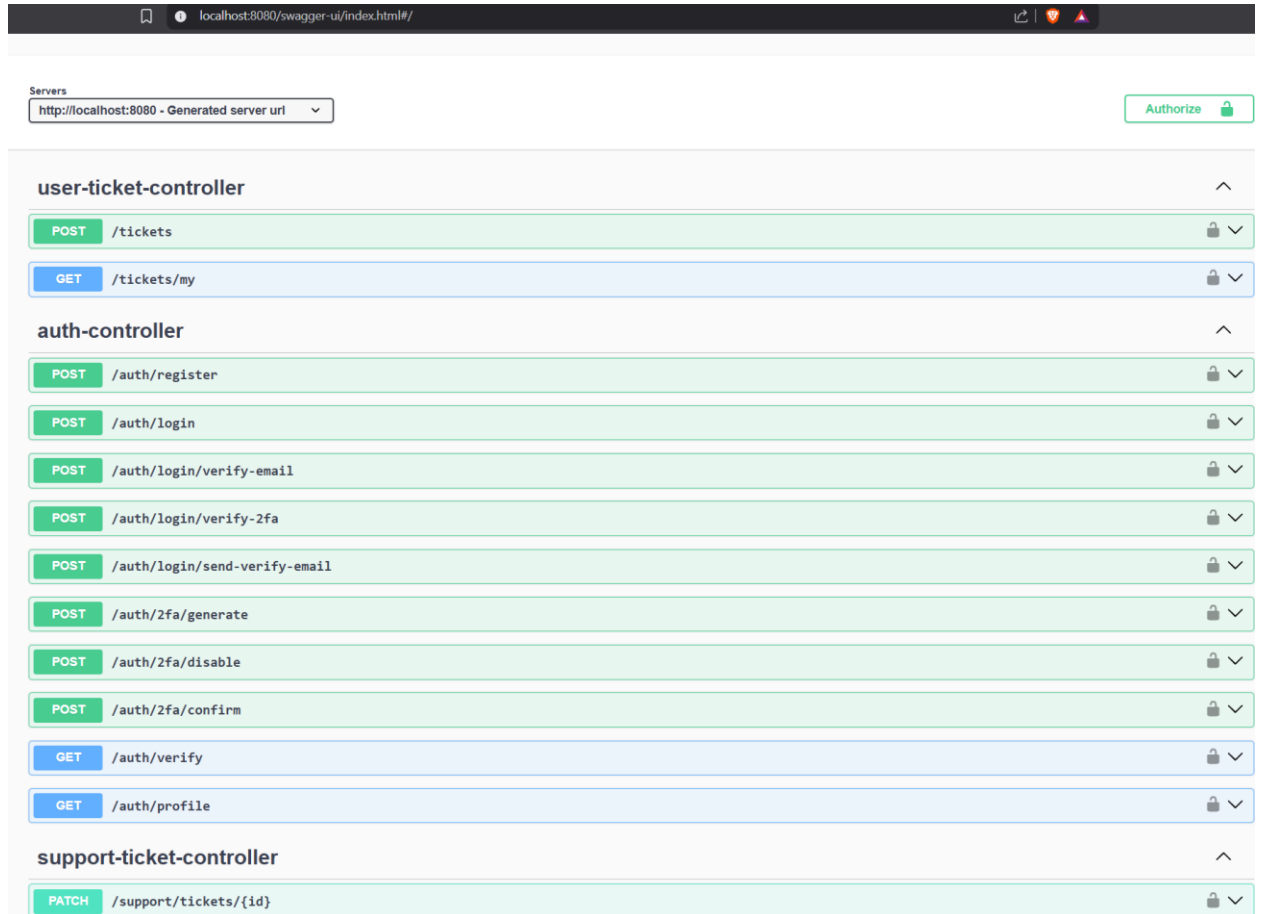


Рисунок 4.9 – Інтерфейс Swagger UI серверного застосунку

4.8.2 Тестування endpoint-ів автентифікації

За допомогою Swagger UI було виконано тестування основних endpoint-ів автентифікації користувачів.

У процесі тестування перевірялися:

- реєстрація користувача;
- вхід у систему;
- підтвердження електронної пошти;
- генерація JWT access token;
- перевірка двофакторної автентифікації.

4.8.3 Тестування JWT-авторизації

Для тестування захищених endpoint-ів у Swagger UI використовується механізм авторизації через Bearer Token.

Після отримання JWT access token користувач додає його через кнопку Authorize, після чого Swagger автоматично додає токен у заголовок: Authorization: Bearer <token>. Це дозволяє виконувати запити до endpoint-ів, які потребують авторизації. Приклад авторизації JWT-токена у Swagger UI наведений на рисунку 4.4.

Після додавання JWT access token Swagger UI автоматично використовує токен для всіх захищених endpoint-ів серверного застосунку.

У процесі тестування було перевірено:

- доступ до /auth/profile;
- перевірку JWT-токена;
- формування SecurityContext;
- коректність role-based authorization.

4.8.4 Тестування двофакторної автентифікації

Для перевірки механізму двофакторної автентифікації у Swagger UI тестувалися endpoint-и:

- /auth/2fa/generate;
- /auth/2fa/confirm;
- /auth/login/verify-2fa.

Під час тестування система:

- генерувала QR-код;
- формувала tempToken;
- перевіряла TOTP-код;
- видавала JWT access token після підтвердження другого фактора.

Особливу увагу було приділено перевірці неможливості використання tempToken для доступу до захищених endpoint-ів.

Приклад перевірки TOTP-коду наведений на рисунку 4.11 та 4.12. У процесі тестування було підтверджено коректну роботу механізму двофакторної автентифікації та перевірки tempToken.

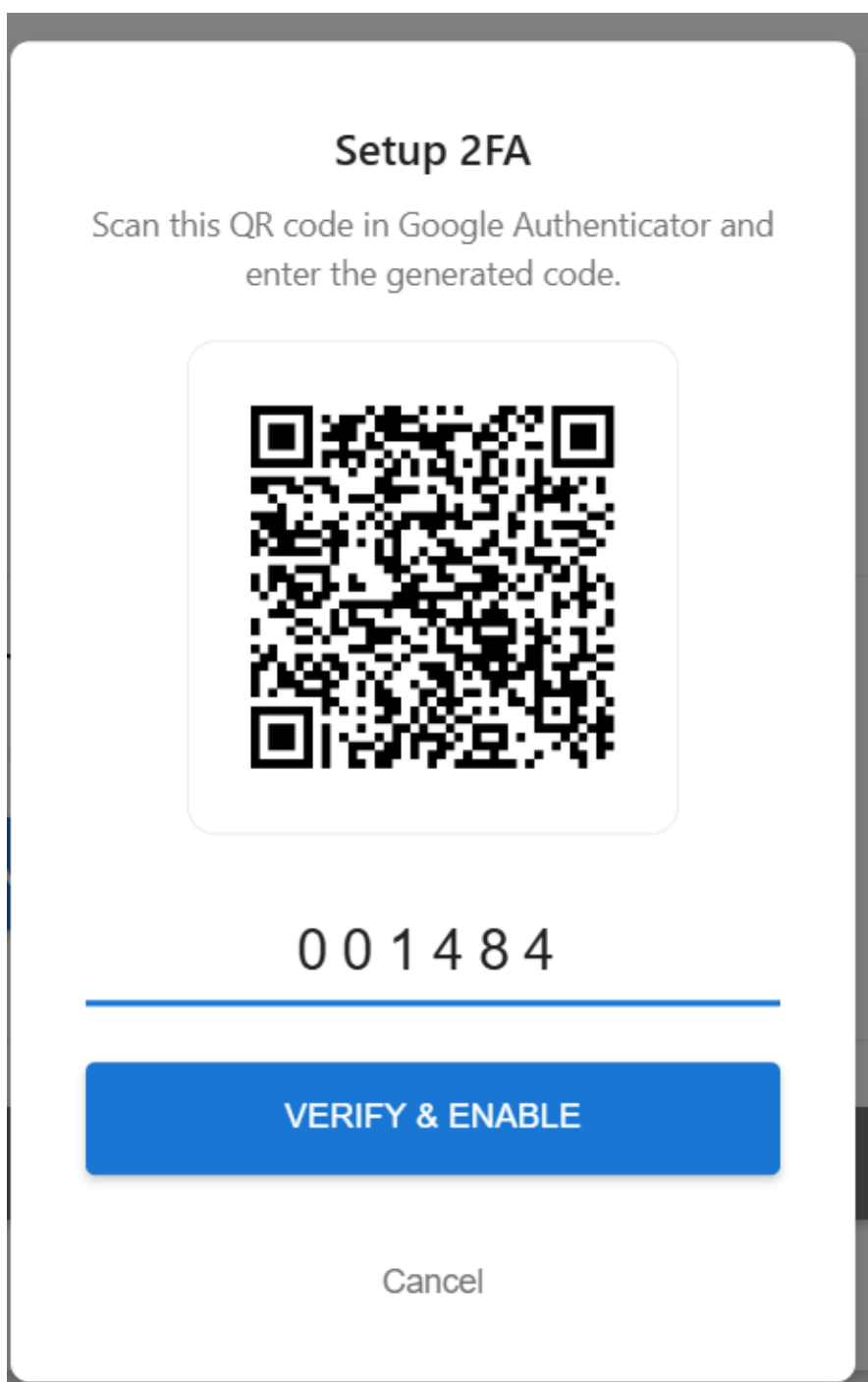


Рисунок 4.11 – Меню підтвердження TOTP-коду

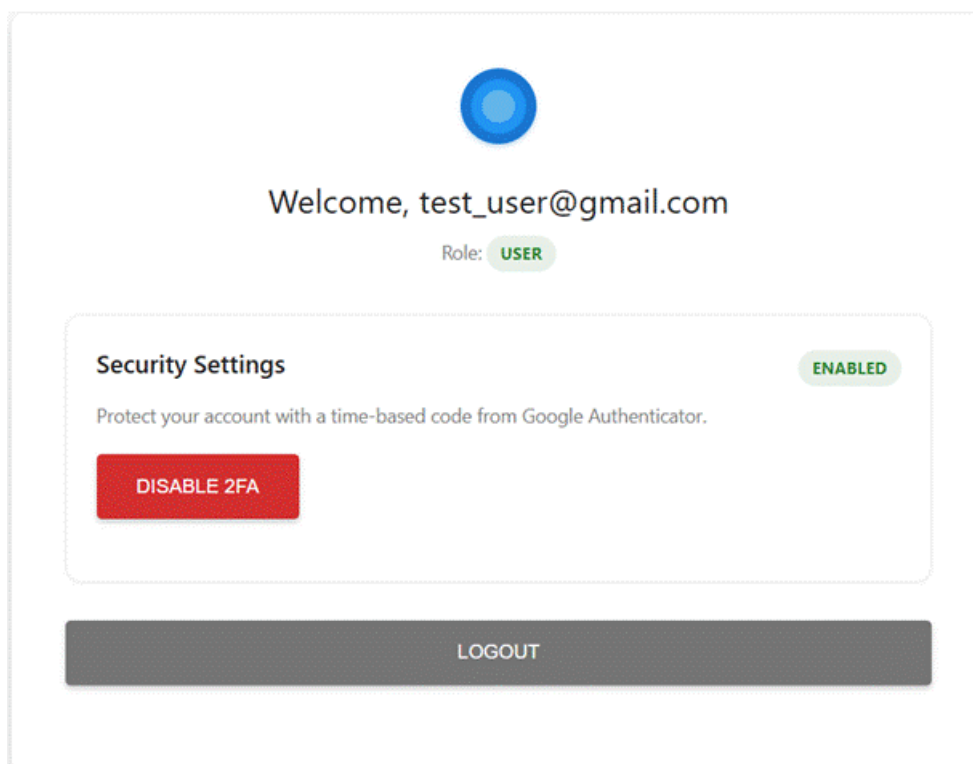


Рисунок 4.11 – Підтвердження TOTP-коду

4.8.5 Тестування системи ролей та контролю доступу

Окремо виконувалося тестування рольового доступу до endpoint-ів серверного застосунку.

У процесі тестування перевірялися:

- доступ USER до користувацьких endpoint-ів;
- доступ SUPPORT до support endpoint-ів;
- доступ ADMIN до адміністративних endpoint-ів;
- блокування доступу при недостатніх правах.

Під час спроби доступу до адміністративних endpoint-ів без ролі ADMIN сервер повертав HTTP-відповідь 403 Forbidden.

Також було перевірено:

- роботу hasRole;
- роботу hasAnyRole;
- перевірку ролей через JWT claims;
- обмеження доступу до support endpoint-ів.

Приклад відповіді сервера при недостатніх правах доступу наведений на рисунку 4.8. Наведений приклад демонструє коректну роботу механізму рольового контролю доступу у серверному застосунку.

4.8.6 Тестування системи підтримки користувачів

Для перевірки системи підтримки користувачів через Swagger UI тестувалися endpoint-и створення та адміністрування тікетів.

У процесі тестування перевірялися:

- створення тікетів;
- отримання списку звернень;
- зміна статусу тікета;
- role-based access control для support endpoint-ів.

Користувачі із роллю USER могли створювати нові тікети, але не мали доступу до endpoint-ів адміністрування. Користувачі із ролями SUPPORT та ADMIN могли переглядати список звернень та змінювати їх статус.

Приклад створення тікета через Swagger UI наведений на рисунку 4.9. Результати тестування підтвердили коректну роботу системи підтримки користувачів та рольового розмежування доступу.

4.8.7 Висновки щодо тестування REST API

У процесі тестування REST API через Swagger UI було перевірено роботу основних механізмів серверного застосунку:

- реєстрації користувачів;
- автентифікації;
- JWT-авторизації;
- двофакторної автентифікації;
- рольового контролю доступу;

- системи підтримки користувачів.

Тестування підтвердило:

- коректність роботи REST API;
- правильність генерації та перевірки JWT-токенів;
- працездатність механізму tempToken;
- коректність role-based authorization;
- стабільність роботи endpoint-ів серверного застосунку.

Використання Swagger UI дозволило значно спростити процес тестування та документування REST API серверного застосунку [11].

4.9 Висновки до розділу

У четвертому розділі дипломної роботи було реалізовано серверний застосунок автентифікації користувачів на основі фреймворку Spring Boot. У процесі реалізації створено багаторівневу архітектуру типу Controller–Service–Repository, що забезпечує модульність, масштабованість та підтримуваність програмного коду.

У межах розділу реалізовано:

- REST API серверного застосунку;
- механізм реєстрації та автентифікації користувачів;
- JWT-авторизацію;
- двофакторну автентифікацію на основі TOTP;
- систему рольового контролю доступу;
- систему підтримки користувачів;
- інтеграцію із PostgreSQL через Spring Data JPA та Hibernate ORM.

Для організації безпеки REST API використано Spring Security [1], BCrypt-хешування паролів [1], JWT-токени [2] та механізм двофакторної автентифікації [5]. Реалізовано захист від brute-force атак через обмеження кількості невдалих спроб входу та тимчасове блокування облікових записів користувачів [4].

Особливістю реалізованого серверного застосунку є використання механізму tempToken, який дозволяє розділити перевірку логіна і пароля та підтвердження другого фактора автентифікації. Такий підхід підвищує рівень безпеки системи та унеможливорює використання тимчасового токена для доступу до захищених endpoint-ів.

У процесі реалізації також було інтегровано Swagger UI [11], який використовується для тестування та документування REST API серверного застосунку. За допомогою Swagger UI було виконано перевірку:

- механізму автентифікації;
- JWT-авторизації;
- двофакторної автентифікації;
- role-based access control;
- системи підтримки користувачів.

Результати тестування підтвердили коректну роботу серверного застосунку, механізмів захисту REST API та системи рольового доступу.

У результаті виконаної реалізації було створено серверний програмний застосунок автентифікації користувачів, який забезпечує безпечну роботу із обліковими записами, підтримує сучасні механізми захисту інформації та може бути використаний як основа для вебзастосунків із підвищеними вимогами до безпеки.

ВИСНОВКИ

У результаті виконання дипломної роботи було розроблено серверний програмний застосунок для автентифікації користувачів із використанням сучасних механізмів захисту інформації та контролю доступу. У межах роботи було проведено аналіз існуючих підходів до автентифікації та авторизації користувачів, досліджено сучасні технології захисту REST API та обґрунтовано вибір архітектурних і програмних рішень для реалізації системи.

У процесі виконання роботи було спроектовано архітектуру серверного застосунку на основі багаторівневої моделі Controller–Service–Repository. Такий підхід дозволив забезпечити розділення відповідальності між компонентами системи, підвищити підтримуваність програмного коду та спростити подальше масштабування програмного забезпечення.

У розробленому застосунку реалізовано:

- REST API для взаємодії клієнта та сервера;
- механізм реєстрації та автентифікації користувачів;
- JWT-авторизацію;
- двофакторну автентифікацію на основі алгоритму TOTP;
- систему рольового контролю доступу;
- механізм перевірки електронної пошти;
- систему підтримки користувачів на основі тікетів;
- механізм захисту від brute-force атак;
- інтеграцію Swagger UI для тестування REST API.

Для забезпечення безпеки серверного застосунку використано Spring Security, BCrypt-хешування паролів, JWT-токени та двофакторну автентифікацію через Google Authenticator. Особливістю реалізованої системи є використання механізму tempToken, який дозволяє відокремити перевірку логіна і пароля від підтвердження другого фактора автентифікації та підвищує рівень захисту REST API.

У процесі роботи було виконано тестування основних функціональних можливостей серверного застосунку через Swagger UI. Результати тестування підтвердили коректність роботи механізмів автентифікації, JWT-авторизації, двофакторної автентифікації, рольового контролю доступу та системи підтримки користувачів.

Під час виконання дипломної роботи було досягнуто поставленої мети – розроблено серверне програмне забезпечення для автентифікації користувачів, яке забезпечує безпечну взаємодію із REST API, підтримує сучасні механізми захисту інформації та може бути використане як основа для вебзастосунків із підвищеними вимогами до безпеки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Spring. Spring Security Reference. URL: <https://docs.spring.io/spring-security/reference> (date of access: 02.05.2026).
2. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. 2015. URL: <https://datatracker.ietf.org/doc/html/rfc7519> (date of access: 02.05.2026).
3. Google. Google Authenticator. URL: <https://support.google.com/accounts/answer/1066447> (date of access: 02.05.2026).
4. OWASP Foundation. OWASP Top 10 Web Application Security Risks. 2021. URL: <https://owasp.org/Top10/2021> (date of access: 02.05.2026).
5. M'Raihi D., Machani S., Pei M., Rydell J. TOTP: Time-Based One-Time Password Algorithm. RFC 6238. 2011. URL: <https://datatracker.ietf.org/doc/html/rfc6238> (date of access: 02.05.2026).
6. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. 2012. URL: <https://datatracker.ietf.org/doc/html/rfc6749> (date of access: 02.05.2026).
7. Google. Firebase Authentication Documentation. URL: <https://firebase.google.com/docs/auth> (date of access: 02.05.2026).
8. PostgreSQL Global Development Group. PostgreSQL Documentation. URL: <https://www.postgresql.org/docs> (date of access: 02.05.2026).
9. Walls C. Spring in Action. 6th ed. Manning Publications, 2022.
10. Hibernate ORM Documentation. URL: <https://hibernate.org/orm/documentation/> (date of access: 15.05.2026).
11. OWASP Foundation. Password Storage Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (date of access: 15.05.2026).

ДОДАТОК А
Технічне завдання

Вступ

Розроблюваний програмний продукт використовується для забезпечення автентифікації, авторизації та захисту облікових записів користувачів у вебзастосунках. Програмний продукт «User Authentication System» реалізований у вигляді серверного REST API-застосунку, який забезпечує реєстрацію користувачів, вхід у систему, перевірку електронної пошти, JWT-авторизацію, двофакторну автентифікацію та рольове розмежування доступу.

Система також надає можливість виконувати такі функції, як керування ролями користувачів, обмеження доступу до захищених endpoint-ів, захист від brute-force атак, ведення журналу подій безпеки та обробка звернень користувачів через систему підтримки.

A.1 Мета розробки і її призначення

Об'єкт дослідження – процес автентифікації та авторизації користувачів у вебзастосунках.

Предмет дослідження – методи захисту облікових записів користувачів, механізми JWT-авторизації, двофакторної автентифікації та рольового контролю доступу.

Мета роботи – дослідження та програмна реалізація серверного програмного забезпечення для автентифікації користувачів із використанням сучасних механізмів захисту інформації для підвищення безпеки REST API та облікових записів користувачів.

Метод дослідження – метод аналізу та порівняння сучасних механізмів автентифікації й авторизації.

Актуальність роботи полягає у необхідності забезпечення захисту облікових записів користувачів у сучасних вебзастосунках. Використання слабких механізмів автентифікації може призвести до несанкціонованого

доступу до персональних даних, компрометації облікових записів та порушення безпеки інформаційних систем. Використання JWT-авторизації, двофакторної автентифікації та рольового контролю доступу дозволяє значно підвищити рівень захисту серверних застосунків.

A.2 Стислий зміст основних теоретичних посилок до розробки на основі проведених пошуків

У сучасних вебзастосунках автентифікація та авторизація є одними з основних механізмів забезпечення інформаційної безпеки. Використання традиційної session-based авторизації ускладнює масштабування серверних застосунків та створює додаткове навантаження на сервер.

Для розв'язання даної проблеми широко використовується технологія JSON Web Token (JWT), яка забезпечує stateless-авторизацію та дозволяє передавати інформацію про користувача у вигляді цифрового токена.

Для додаткового захисту облікових записів використовується двофакторна автентифікація (2FA), яка реалізується через алгоритм TOTP та застосунок Google Authenticator.

Також важливим елементом безпеки є рольовий контроль доступу (RBAC), який дозволяє розмежувати права користувачів та обмежити доступ до адміністративних функцій системи.

У зв'язку із постійним зростанням кількості атак на вебзастосунки розробка серверного застосунку із сучасними механізмами автентифікації та авторизації є актуальною та доцільною.

A.3 Підстави для розробки

Підставою для виконання розробки є завдання на дипломний проект на тему “Розроблення серверного програмного забезпечення для

автентифікації користувачів”, затверджене наказом по НУ «Запорізька політехніка» № 139 від 07 квітня 2026 року.

A.4 Основні вимоги до програми, що розробляється, початковим даним і результату

A.4.1 Вимоги до функціональних характеристик

Вимоги до функціональних характеристик програмного продукту:

- повинна бути реалізована операція реєстрації користувачів;
- повинна бути реалізована операція автентифікації користувачів;
- повинна бути реалізована JWT-авторизація для доступу до захищених endpoint-ів;
- повинна бути реалізована операція перевірки електронної пошти;
- повинна бути реалізована двофакторна автентифікація (2FA) на основі TOTP;
- повинна бути реалізована генерація та перевірка JWT access token;
- повинна бути реалізована генерація тимчасового tempToken для проходження 2FA;
- повинна бути реалізована система ролей користувачів USER, SUPPORT та ADMIN;
- повинна бути реалізована система підтримки користувачів на основі тікетів;
- повинна бути реалізована операція блокування облікового запису після перевищення допустимої кількості невдалих спроб входу;
- повинна бути реалізована можливість ведення журналу подій безпеки;
- повинна бути реалізована можливість тестування REST API через Swagger UI.

A.4.2 Вимоги до надійності

Програмне забезпечення має відповідати певним вимогам щодо надійності, серед яких такі:

- захист від несанкціонованого доступу до захищених endpoint-ів;
- захист від використання некоректних JWT-токенів;
- захист від brute-force атак;
- захист від помилкових дій користувача;
- перевірка коректності вхідних даних.

У разі виникнення помилок система повинна повертати відповідні HTTP-відповіді та повідомлення про помилки.

A.4.3 Умови експлуатації

Умови експлуатації:

- програмне забезпечення функціонує як серверний REST API-застосунок;
- програмне забезпечення не потребує додаткового обслуговуючого персоналу;
- для використання REST API не потрібна спеціальна підготовка користувача;
- доступ до API може здійснюватися через Swagger UI або сторонні REST-клієнти.

A.4.4 Вимоги до складу та параметрів технічних засобів

Вимоги до складу та параметрів технічних засобів:

- платформа x64-based;
- операційна система Microsoft Windows 11/10 або Linux;
- Java Development Kit 17;

- PostgreSQL 15 або новіше;
- оперативна пам'ять від 4 ГБ;
- процесор із мінімальною тактовою частотою 1,8 ГГц;
- мінімум 2 ГБ вільного місця на жорсткому диску.

A.4.5 Вимоги до маркування та упакування

Програмне забезпечення не потребує додаткових дій при маркуванні та упакуванні.

A.4.6 Вимоги до транспортування та збереження

Вимоги до транспортування та збереження аналогічні вимогам, які висуваються до носія, на якому записане програмне забезпечення.

A.5 Вимоги до програмної документації

Склад програмної документації повинен включати в себе:

- технічне завдання;
- керівництво програміста;
- керівництво користувача;
- опис REST API;
- опис структури бази даних.

A.6 Основні обмеження на установку і використання програмного продукту

Розроблюваний програмний продукт потребує встановлення Java Runtime Environment та PostgreSQL. Серверний застосунок повинен

використовуватись на комп'ютерах, які відповідають вимогам до технічних засобів.

Для коректної роботи системи необхідне підключення до бази даних PostgreSQL.

Програмний продукт може використовуватись користувачем у будь-який момент часу та не конфліктує з іншими програмами, які функціонують у системі.

ДОДАТОК Б
Текст програми

Б.1 Текст файла AuthService.java

```
package com.penguin.usersau.service;

import com.penguin.usersau.config.JwtUtils;
import com.penguin.usersau.dto.request.LoginRequest;
import com.penguin.usersau.dto.request.RegisterRequest;
import com.penguin.usersau.dto.response.LoginResponse;
import com.penguin.usersau.dto.response.UserResponse;
import com.penguin.usersau.mapper.UserMapper;
import com.penguin.usersau.model.AuditAction;
import com.penguin.usersau.model.User;
import com.penguin.usersau.repository.UserRepository;
import dev.samstevens.totp.code.CodeVerifier;
import dev.samstevens.totp.code.DefaultCodeGenerator;
import dev.samstevens.totp.code.DefaultCodeVerifier;
import dev.samstevens.totp.code.HashingAlgorithm;
import dev.samstevens.totp.exceptions.QrGenerationException;
import dev.samstevens.totp.qr.QrData;
import dev.samstevens.totp.qr.QrGenerator;
import dev.samstevens.totp.qr.ZxingPngQrGenerator;
import dev.samstevens.totp.secret.DefaultSecretGenerator;
import dev.samstevens.totp.secret.SecretGenerator;
import dev.samstevens.totp.time.SystemTimeProvider;
import lombok.RequiredArgsConstructor;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;

import static dev.samstevens.totp.util.Utils.getDataUriForImage;

@Service
@RequiredArgsConstructor
public class AuthService {
    private final UserRepository userRepository;
    private final EmailService emailService;
    private final PasswordEncoder passwordEncoder;
    private final JwtUtils jwtUtils;
    private final AuditLogService auditLogService;

    private final QrGenerator qrGenerator = new ZxingPngQrGenerator();
    private final SecretGenerator secretGenerator = new DefaultSecretGenerator();

    public void register(RegisterRequest request) {
```

```

    if (request.email == null || request.password == null) {
        throw new RuntimeException("Email and password are required");
    }

    if (!request.email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
        throw new RuntimeException("Invalid email format");
    }

    if (userRepository.findByEmail(request.email).isPresent()) {
        throw new RuntimeException("User with this email already exists");
    }

    User user = new User();
    user.setEmail(request.email);
    user.setPassword(passwordEncoder.encode(request.password));

    sendVerifyEmail(user);
    userRepository.save(user);
    auditLogService.log(AuditAction.REGISTER, user.getEmail(), true, "User
registered");
}

public LoginResponse login(LoginRequest request) {
    User user = userRepository.findByEmail(request.email)
        .orElseThrow(() -> new RuntimeException("User not found"));

    if (user.getLockedUntil() != null &&
user.getLockedUntil().isAfter(LocalDate.now())) {
        auditLogService.log(AuditAction.LOGIN_FAILED, user.getEmail(), false,
"Account is locked");
        throw new RuntimeException("Account is locked until " +
user.getLockedUntil());
    }

    if (!passwordEncoder.matches(request.password, user.getPassword())) {
        handleFailedLogin(user);
        auditLogService.log(AuditAction.LOGIN_FAILED, request.email, false,
"Invalid credentials");
        throw new RuntimeException("Invalid credentials");
    }

    if (!user.isEmailVerified()) {
        throw new RuntimeException("Email not verified. Please check your
inbox.");
    }
}

```

```

    resetFailedLogins(user);
    user.setLastLogin(LocalDateTime.now());
    userRepository.save(user);

    UserResponse userResponse = UserMapper.toResponse(user);

    if (user.isTwoFactorEnabled()) {
        auditLogService.log(AuditAction.LOGIN_2FA_REQUIRED,
            user.getEmail(), true, "2FA required");
        return LoginResponse.builder()
            .tempToken(jwtUtils.genTempToken(user.getEmail()))
            .requires2fa(true)
            .user(userResponse)
            .build();
    }

    auditLogService.log(AuditAction.LOGIN_SUCCESS, user.getEmail(), true,
        "User logged in");
    return LoginResponse.builder()
        .token(jwtUtils.genToken(user.getEmail(), user.getRole().name()))
        .requires2fa(false)
        .user(userResponse)
        .build();
}

public LoginResponse verifyLogin2FA(String tempToken, String code) {
    if (tempToken == null || !jwtUtils.isTokenValid(tempToken) ||
        !jwtUtils.isTempToken(tempToken)) {
        throw new RuntimeException("Invalid or expired temporary token");
    }

    String email = jwtUtils.extractEmail(tempToken);

    User user = userRepository.findByEmail(email)
        .orElseThrow(() -> new RuntimeException("User not found"));

    if (!user.isTwoFactorEnabled() || user.getTwoFactorSecret() == null) {
        throw new RuntimeException("2FA is not enabled");
    }

    if (!isValidTotpCode(user.getTwoFactorSecret(), code)) {
        auditLogService.log(AuditAction.LOGIN_2FA_FAILED, user.getEmail(),
            false, "Invalid 2FA code");
        throw new RuntimeException("Invalid 2FA code");
    }
}

```

```

    }

    auditLogService.log(AuditAction.LOGIN_2FA_SUCCESS, user.getEmail(),
true, "2FA login confirmed");
    return LoginResponse.builder()
        .token(jwtUtils.genToken(user.getEmail(), user.getRole().name()))
        .requires2fa(false)
        .user(UserMapper.toResponse(user))
        .build();
}

public String generate2FAData(String email) {
    User user = userRepository.findByEmail(email)
        .orElseThrow(() -> new RuntimeException("User not found"));

    if (user.isTwoFactorEnabled()) {
        throw new RuntimeException("2FA is already enabled");
    }

    String secret = secretGenerator.generate();
    user.setPendingTwoFactorSecret(secret);
    userRepository.save(user);

    QrData data = new QrData.Builder()
        .label(user.getEmail())
        .secret(secret)
        .issuer("DiplomAuth")
        .algorithm(HashingAlgorithm.SHA1)
        .digits(6)
        .period(30)
        .build();

    try {
        byte[] imageData = qrGenerator.generate(data);
        auditLogService.log(AuditAction.ENABLE_2FA_STARTED,
user.getEmail(), true, "QR for 2FA generated");
        return getDataUriForImage(imageData,
qrGenerator.getImageMimeType());
    } catch (QrGenerationException e) {
        throw new RuntimeException("QR generation failed");
    }
}

public void confirmEnable2FA(String email, String code) {
    User user = userRepository.findByEmail(email)

```

```

        .orElseThrow(() -> new RuntimeException("User not found"));

    if (user.getPendingTwoFactorSecret() == null) {
        throw new RuntimeException("2FA setup was not started");
    }

    if (!isValidTotpCode(user.getPendingTwoFactorSecret(), code)) {
        throw new RuntimeException("Invalid 2FA code");
    }

    user.setTwoFactorSecret(user.getPendingTwoFactorSecret());
    user.setPendingTwoFactorSecret(null);
    user.setTwoFactorEnabled(true);

    userRepository.save(user);
    auditLogService.log(AuditAction.ENABLE_2FA_CONFIRMED,
user.getEmail(), true, "2FA enabled");
    }

    public void disable2FA(String email, String code) {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found"));

        if (!user.isTwoFactorEnabled() || user.getTwoFactorSecret() == null) {
            throw new RuntimeException("2FA is not enabled");
        }

        if (!isValidTotpCode(user.getTwoFactorSecret(), code)) {
            throw new RuntimeException("Invalid 2FA code");
        }

        user.setTwoFactorEnabled(false);
        user.setTwoFactorSecret(null);
        user.setPendingTwoFactorSecret(null);

        userRepository.save(user);
        auditLogService.log(AuditAction.DISABLE_2FA, user.getEmail(), true,
"2FA disabled");
    }

    public void sendVerifyEmail(String email) {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found"));

        sendVerifyEmail(user);
    }

```

```

    }

    public void sendVerifyEmail(User user) {
        String code = String.valueOf((int) ((Math.random() * 900000) + 100000));

        user.setEmailVerificationCode(code);
        user.setEmailVerificationExpires(LocalDateTime.now().plusHours(24));

        userRepository.save(user);
        emailService.sendVerificationEmail(user.getEmail(), code);
    }

    public void verifyEmail(String email, String code) {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found"));

        if (user.getEmailVerificationCode() == null ||
            user.getEmailVerificationExpires() == null ||
            user.getEmailVerificationExpires().isBefore(LocalDateTime.now()) ||
            !user.getEmailVerificationCode().equals(code)) {
            throw new RuntimeException("Invalid or expired code");
        }

        user.setEmailVerified(true);
        user.setEmailVerificationCode(null);
        user.setEmailVerificationExpires(null);

        userRepository.save(user);
        auditLogService.log(AuditAction.EMAIL_VERIFIED, user.getEmail(), true,
            "Email verified");
    }

    public UserResponse getUserByEmail(String email) {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found"));

        return UserMapper.toResponse(user);
    }

    private boolean isValidTotpCode(String secret, String code) {
        if (secret == null || code == null || !code.matches("\\d{6}")) {
            return false;
        }

        CodeVerifier verifier = new DefaultCodeVerifier(

```

```

        new DefaultCodeGenerator(),
        new SystemTimeProvider()
    );

    return verifier.isValidCode(secret, code);
}

private void handleFailedLogin(User user) {
    user.setFailedLoginAttempts(user.getFailedLoginAttempts() + 1);

    if (user.getFailedLoginAttempts() >= 5) {
        int min = 15;
        user.setLockedUntil(LocalDateTime.now().plusMinutes(min));
        auditLogService.log(AuditAction.ACCOUNT_LOCKED, user.getEmail(),
            true, "Account locked in " + min + " minutes");
    }

    userRepository.save(user);
}

private void resetFailedLogins(User user) {
    user.setFailedLoginAttempts(0);
    user.setLockedUntil(null);
    userRepository.save(user);
}
}

```

Б.2 Текст файла SecurityConfig.java

```

package com.penguin.usersau.config;

import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;

```

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
```

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private final JwtAuthenticationFilter jwtAuthenticationFilter;
```

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .sessionManagement(session ->
```

```
    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        )
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(
                "/",
                "/login.html",
                "/register.html",
                "/dashboard.html",
                "/css/**",
                "/js/**"
            ).permitAll()

            .requestMatchers(
                "/swagger-ui/**",
                "/swagger-ui.html",
                "/v3/api-docs/**",
                "/v3/api-docs",
                "/v3/api-docs/swagger-config",
                "/auth/register",
                "/auth/login",
```

```

        "/auth/login/verify-2fa",
        "/auth/login/send-verify-email",
        "/auth/login/verify-email",
        "/auth/verify"
    ).permitAll()

    .requestMatchers("/auth/2fa/**").authenticated()
    .requestMatchers("/auth/profile").authenticated()

    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/tickets/**").authenticated()
    .requestMatchers("/support/**").hasAnyRole("SUPPORT",
"ADMIN")

    .anyRequest().authenticated()
)
.addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class);

return http.build();
}
}

```

Б.3 Текст файла User.java

```

package com.penguin.usersau.model;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.time.LocalDateTime;
import java.util.UUID;

@Setter
@Getter
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue
    private UUID uuid;

    @Column(unique = true, nullable = false)
    private String email;

```

```

@Column(nullable = false)
private String password;

@Enumerated(EnumType.STRING)
private Role role = Role.USER;

private int failedLoginAttempts = 0;
private LocalDateTime lockedUntil;

private boolean twoFactorEnabled = false;
private String twoFactorSecret;
private String pendingTwoFactorSecret;

private LocalDateTime createdAt = LocalDateTime.now();
private LocalDateTime lastLogin;

private String emailVerificationCode;
private LocalDateTime emailVerificationExpires;
private boolean emailVerified = false;
}

```

Б.4 Текст файла TicketRepository.java

```

package com.penguin.usersau.repository;

import com.penguin.usersau.model.Ticket;
import com.penguin.usersau.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface TicketRepository extends JpaRepository<Ticket, Long> {
    List<Ticket> findByUserOrderByCreatedAtDesc(User user);

    List<Ticket> findAllByOrderByCreatedAtDesc();
}

```

Б.5 Текст файла JwtUtils.java

```

package com.penguin.usersau.config;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;

```

```

import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.nio.charset.StandardCharsets;
import java.security.Key;
import java.util.Date;

@Component
public class JwtUtils {
    @Value("${jwt.secret}")
    private String secret;

    private final int ACCESS_EXP = 86400000; // 24h
    private final int TEMP_EXP = 300000; // 5m

    public String genToken(String email, String role) {
        return Jwts.builder()
            .setSubject(email)
            .claim("role", role)
            .claim("type", "ACCESS")
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + ACCESS_EXP))
            .signWith(getKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    public String genTempToken(String email) {
        return Jwts.builder()
            .setSubject(email)
            .claim("type", "2FA_PENDING")
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + TEMP_EXP))
            .signWith(getKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    public String extractEmail(String token) {
        return extractClaims(token).getSubject();
    }

    public String extractRole(String token) {
        return extractClaims(token).get("role", String.class);
    }
}

```

```

public boolean isAccessToken(String token) {
    return "ACCESS".equals(extractClaims(token).get("type", String.class));
}

public boolean isTempToken(String token) {
    return "2FA_PENDING".equals(extractClaims(token).get("type",
String.class));
}

public boolean isTokenValid(String token) {
    try {
        Claims claims = extractClaims(token);
        return claims.getExpiration().after(new Date());
    } catch (Exception e) {
        return false;
    }
}

private Claims extractClaims(String token) {
    return Jwts.parserBuilder()
        .setSigningKey(getKey())
        .build()
        .parseClaimsJws(token)
        .getBody();
}

private Key getKey() {
    return Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8));
}
}

```

Б.6 Текст файла AdminController.java

```

package com.penguin.usersau.controller;

import com.penguin.usersau.dto.response.UserResponse;
import com.penguin.usersau.mapper.AuditMapper;
import com.penguin.usersau.mapper.UserMapper;
import com.penguin.usersau.model.AuditAction;
import com.penguin.usersau.model.Role;
import com.penguin.usersau.model.User;
import com.penguin.usersau.repository.AuditLogRepository;
import com.penguin.usersau.repository.UserRepository;
import com.penguin.usersau.service.AuditLogService;

```

```

import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.time.LocalDateTime;
import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/admin")
@RequiredArgsConstructor
@PreAuthorize("hasRole('ADMIN')")
public class AdminController {
    private final UserRepository userRepository;
    private final AuditLogRepository auditLogRepository;
    private final AuditLogService auditLogService;

    @GetMapping("/users")
    public List<UserResponse> getAllUsers() {
        return userRepository.findAll().stream()
            .map(UserMapper::toResponse)
            .toList();
    }

    @GetMapping("/users/{uuid}")
    public ResponseEntity<UserResponse> getUser(@PathVariable UUID uuid) {
        return userRepository.findById(uuid)
            .map(user -> ResponseEntity.ok(UserMapper.toResponse(user)))
            .orElse(ResponseEntity.notFound().build());
    }

    @PatchMapping("/users/{uuid}/role")
    public ResponseEntity<?> changeRole(@PathVariable UUID uuid,
    @RequestBody Role role) {
        return userRepository.findById(uuid)
            .map(user -> {
                Role oldRole = user.getRole();
                user.setRole(role);
                userRepository.save(user);
                auditLogService.log(AuditAction.ROLE_CHANGED,
user.getEmail(), true, "Role changed from " + oldRole.name() + " to " +
role.name());
                return ResponseEntity.ok("Role updated");
            })
    }

```

```

        .orElse(ResponseEntity.notFound().build());
    }

    @PatchMapping("/users/{uuid}/lock")
    public ResponseEntity<?> lockUser(@PathVariable UUID uuid) {
        return userRepository.findById(uuid)
            .map(user -> {
                user.setLockedUntil(LocalDateTime.now().plusHours(1));
                userRepository.save(user);
                auditLogService.log(AuditAction.USER_LOCKED_BY_ADMIN,
user.getEmail(), true, "Locked by admin");
                return ResponseEntity.ok("User locked");
            })
            .orElse(ResponseEntity.notFound().build());
    }

    @PatchMapping("/users/{uuid}/unlock")
    public ResponseEntity<?> unlockUser(@PathVariable UUID uuid) {
        return userRepository.findById(uuid)
            .map(user -> {
                user.setLockedUntil(null);
                user.setFailedLoginAttempts(0);
                userRepository.save(user);
                auditLogService.log(AuditAction.USER_UNLOCKED_BY_ADMIN,
user.getEmail(), true, "Unlocked by admin");
                return ResponseEntity.ok("User unlocked");
            })
            .orElse(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/users/{uuid}")
    public ResponseEntity<?> deleteUser(@PathVariable UUID uuid) {
        if (!userRepository.existsById(uuid)) {
            return ResponseEntity.notFound().build();
        }

        User user = userRepository.findById(uuid).orElse(null);
        userRepository.deleteById(uuid);
        auditLogService.log(AuditAction.USER_DELETED_BY_ADMIN, user !=
null ? user.getEmail() : uuid.toString(), true, "Deleted by admin");
        return ResponseEntity.ok("User deleted");
    }

    @GetMapping("/audit")
    public ResponseEntity<?> getAuditLogs() {

```

```

        return ResponseEntity.ok(
            auditLogRepository.findTop50ByOrderByCreatedAtDesc()
                .stream()
                .map(AuditMapper::toResponse)
                .toList()
        );
    }

    @GetMapping("/audit/{email}")
    public ResponseEntity<?> getAuditLogsByEmail(@PathVariable String email)
    {
        return ResponseEntity.ok(
            auditLogRepository.findByEmailOrderByCreatedAtDesc(email)
                .stream()
                .map(AuditMapper::toResponse)
                .toList()
        );
    }
}

```

Б.7 Текст файла JwtAuthenticationFilter.java

```

package com.penguin.usersau.config;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;
import java.util.List;

@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private final JwtUtils jwtUtils;

```

```

@Override
protected void doFilterInternal(HttpServletRequest request,
    HttpServletResponse response, FilterChain filterChain) throws ServletException,
    IOException {
    String authHeader = request.getHeader("Authorization");

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

    String token = authHeader.substring(7);

    if (!jwtUtils.isTokenValid(token) || !jwtUtils.isAccessToken(token)) {
        filterChain.doFilter(request, response);
        return;
    }

    String path = request.getServletPath();

    if (path.startsWith("/swagger-ui")
        || path.startsWith("/v3/api-docs")
        || path.equals("/swagger-ui.html")) {
        filterChain.doFilter(request, response);
        return;
    }

    String email = jwtUtils.extractEmail(token);
    String role = jwtUtils.extractRole(token);

    UsernamePasswordAuthenticationToken authentication =
        new UsernamePasswordAuthenticationToken(
            email,
            null,
            List.of(new SimpleGrantedAuthority("ROLE_" + role))
        );

    SecurityContextHolder.getContext().setAuthentication(authentication);

    filterChain.doFilter(request, response);
}
}

```

Б.8 Текст файла AuthController.java

```
package com.penguin.usersau.controller;

import com.penguin.usersau.dto.request.LoginRequest;
import com.penguin.usersau.dto.request.RegisterRequest;
import com.penguin.usersau.dto.response.LoginResponse;
import com.penguin.usersau.dto.response.UserResponse;
import com.penguin.usersau.service.AuthService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

@RestController
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {
    private final AuthService authService;

    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody RegisterRequest request) {
        try {
            authService.register(request);
            return ResponseEntity.ok(Map.of(
                "message", "User registered successfully. Please verify your email."
            ));
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(
                "error", e.getMessage()
            ));
        }
    }

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {
        try {
            LoginResponse response = authService.login(request);
            return ResponseEntity.ok(response);
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(
                "error", e.getMessage()
            ));
        }
    }
}
```

```
    }
}
```

```
@PostMapping("/login/verify-2fa")
public ResponseEntity<?> verifyLogin2FA(@RequestBody Map<String,
String> request) {
    try {
        String tempToken = request.get("tempToken");
        String code = request.get("code");

        LoginResponse response = authService.verifyLogin2FA(tempToken,
code);
        return ResponseEntity.ok(response);
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(Map.of(
            "error", e.getMessage()
        ));
    }
}
```

```
@PostMapping("/login/send-verify-email")
public ResponseEntity<?> sendVerifyEmail(@RequestBody Map<String,
String> request) {
    try {
        authService.sendVerifyEmail(request.get("email"));
        return ResponseEntity.ok(Map.of(
            "message", "Verification code sent successfully"
        ));
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(Map.of(
            "error", e.getMessage()
        ));
    }
}
```

```
@PostMapping("/login/verify-email")
public ResponseEntity<?> verifyEmail(@RequestBody Map<String, String>
request) {
    try {
        authService.verifyEmail(request.get("email"), request.get("code"));
        return ResponseEntity.ok(Map.of(
            "message", "Email verified successfully"
        ));
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(Map.of(
```

```

        "error", e.getMessage()
    ));
}
}

```

```

@GetMapping("/verify")
public ResponseEntity<?> verifyEmailByLink(@RequestParam String email,
                                           @RequestParam String code) {
    try {
        authService.verifyEmail(email, code);
        return ResponseEntity.ok(Map.of(
            "message", "Email verified successfully"
        ));
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(Map.of(
            "error", e.getMessage()
        ));
    }
}

```

```

@GetMapping("/profile")
public ResponseEntity<?> profile(Authentication authentication) {
    try {
        UserResponse response =
authService.getUserByEmail(authentication.getName());
        return ResponseEntity.ok(response);
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(Map.of(
            "error", e.getMessage()
        ));
    }
}

```

```

@PostMapping("/2fa/generate")
public ResponseEntity<?> generate2FA(Authentication authentication) {
    try {
        String qrCode = authService.generate2FADData(authentication.getName());
        return ResponseEntity.ok(Map.of(
            "qrCode", qrCode
        ));
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(Map.of(
            "error", e.getMessage()
        ));
    }
}

```

```

    }

    @PostMapping("/2fa/confirm")
    public ResponseEntity<?> confirmEnable2FA(Authentication authentication,
                                              @RequestBody Map<String, String> request) {
        try {
            authService.confirmEnable2FA(authentication.getName(),
request.get("code"));
            return ResponseEntity.ok(Map.of(
                "message", "2FA enabled successfully"
            ));
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(
                "error", e.getMessage()
            ));
        }
    }

    @PostMapping("/2fa/disable")
    public ResponseEntity<?> disable2FA(Authentication authentication,
                                         @RequestBody Map<String, String> request) {
        try {
            authService.disable2FA(authentication.getName(), request.get("code"));
            return ResponseEntity.ok(Map.of(
                "message", "2FA disabled successfully"
            ));
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(
                "error", e.getMessage()
            ));
        }
    }
}

```

Б.9 Текст файла SupportTicketController.java

```

package com.penguin.usersau.controller;

import com.penguin.usersau.dto.request.UpdateTicketRequest;
import com.penguin.usersau.service.TicketService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

```

```

@RestController
@RequestMapping("/support/tickets")
@RequiredArgsConstructor
@PreAuthorize("hasAnyRole('SUPPORT', 'ADMIN')")
public class SupportTicketController {
    private final TicketService ticketService;

    @GetMapping
    public ResponseEntity<?> getAllTickets() {
        return ResponseEntity.ok(ticketService.getAllTickets());
    }

    @PatchMapping("/{id}")
    public ResponseEntity<?> updateTicket(@PathVariable Long id,
    @RequestBody UpdateTicketRequest request) {
        return ResponseEntity.ok(ticketService.updateTicket(id, request));
    }
}

```

Б.10 Текст файла TicketResponse.java

```

package com.penguin.usersau.dto.response;

import com.penguin.usersau.model.TicketStatus;
import lombok.AllArgsConstructor;
import lombok.Getter;

import java.time.LocalDateTime;

@Getter
@AllArgsConstructor
public class TicketResponse {
    private Long id;
    private String userEmail;
    private String subject;
    private String message;
    private TicketStatus status;
    private String supportAnswer;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}

```

Б.11 Текст файла TicketService.java

```
package com.penguin.usersau.service;

import com.penguin.usersau.dto.request.CreateTicketRequest;
import com.penguin.usersau.dto.request.UpdateTicketRequest;
import com.penguin.usersau.dto.response.TicketResponse;
import com.penguin.usersau.mapper.TicketMapper;
import com.penguin.usersau.model.Ticket;
import com.penguin.usersau.model.User;
import com.penguin.usersau.repository.TicketRepository;
import com.penguin.usersau.repository.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.List;

@Service
@RequiredArgsConstructor
public class TicketService {
    private final TicketRepository ticketRepository;
    private final UserRepository userRepository;

    public TicketResponse createTicket(String email, CreateTicketRequest request)
    {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found"));

        Ticket ticket = new Ticket();
        ticket.setUser(user);
        ticket.setSubject(request.getSubject());
        ticket.setMessage(request.getMessage());

        return TicketMapper.toResponse(ticketRepository.save(ticket));
    }

    public List<TicketResponse> getMyTickets(String email) {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new RuntimeException("User not found"));

        return ticketRepository.findByUserOrderByCreatedAtDesc(user)
            .stream()
            .map(TicketMapper::toResponse)
            .toList();
    }
}
```

```

    }

    public List<TicketResponse> getAllTickets() {
        return ticketRepository.findAllByOrderByCreatedAtDesc()
            .stream()
            .map(TicketMapper::toResponse)
            .toList();
    }

    public TicketResponse updateTicket(Long id, UpdateTicketRequest request) {
        Ticket ticket = ticketRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Ticket not found"));

        if (request.getStatus() != null) {
            ticket.setStatus(request.getStatus());
        }

        if (request.getSupportAnswer() != null &&
!request.getSupportAnswer().isBlank()) {
            ticket.setSupportAnswer(request.getSupportAnswer());
        }

        ticket.setUpdatedAt(LocalDateTime.now());

        return TicketMapper.toResponse(ticketRepository.save(ticket));
    }
}

```

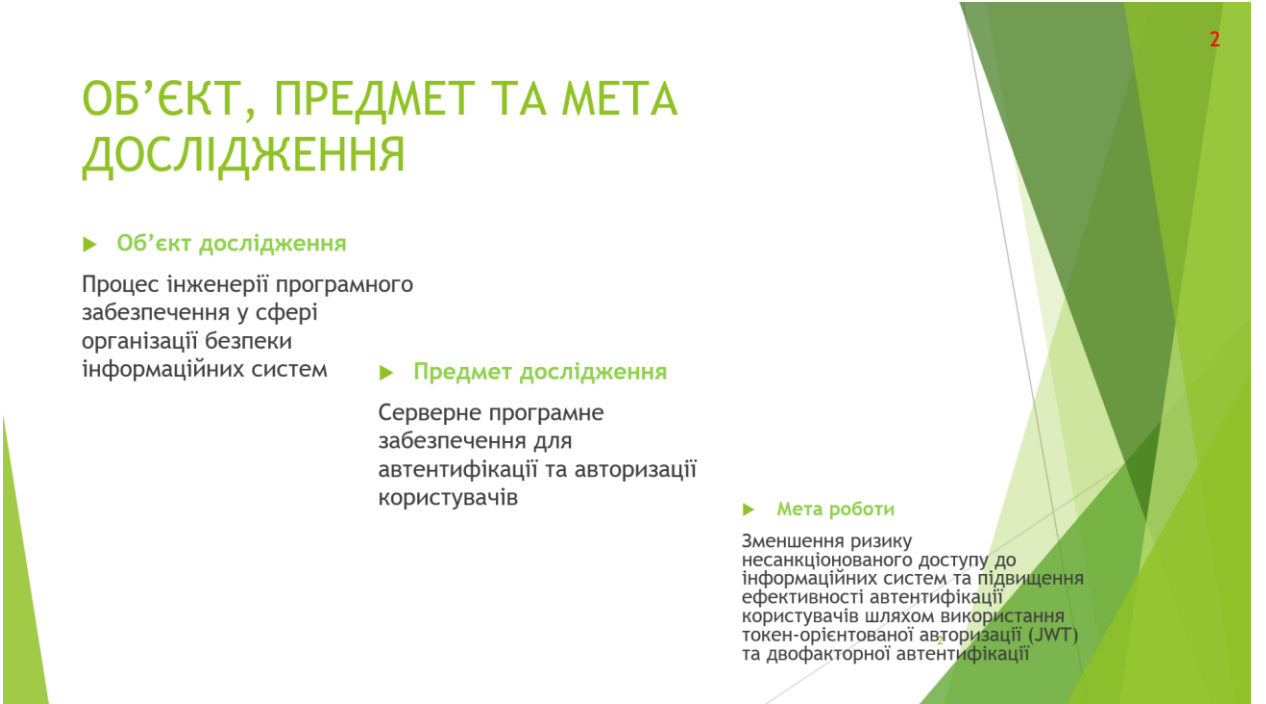
ДОДАТОК В
Слайди презентації

Дипломна кваліфікаційна робота бакалавра
Тема: “Розроблення серверного програмного
забезпечення для автентифікації
користувачів”

Виконав ст. гр. КНТ-132
Керівник к.т.н.

Іван Гостєв
Олена ШИТІКОВА

Рисунок В.1 – Слайд 1



ОБ'ЄКТ, ПРЕДМЕТ ТА МЕТА ДОСЛІДЖЕННЯ

► **Об'єкт дослідження**

Процес інженерії програмного
забезпечення у сфері
організації безпеки
інформаційних систем

► **Предмет дослідження**

Серверне програмне
забезпечення для
автентифікації та авторизації
користувачів

► **Мета роботи**

Зменшення ризику
несанкціонованого доступу до
інформаційних систем та підвищення
ефективності автентифікації
користувачів шляхом використання
токен-орієнтованої авторизації (JWT)
та двофакторної автентифікації

Рисунок В.2 – Слайд 2

АКТУАЛЬНІСТЬ РОБОТИ

Проблема	Наслідок
Слабкі механізми автентифікації	Компрометація акаунтів
Відсутність 2FA	Несанкціонований доступ
Session-based auth	Складність масштабування
Відсутність RBAC	Порушення контролю доступу

Рисунок В.3 – Слайд 3

АНАЛІЗ АНАЛОГІВ

Критерій	Auth0	Firebase	Власна система
JWT	+	+	+
TOTP 2FA	+	+	+
Open Source	-	-	+
Локальне розгортання	-	-	+
RBAC	+	Частково	
Кастомізація	Обмежена	Обмежена	Повна

Рисунок В.4 – Слайд 4

АРХИТЕКТУРА СИСТЕМИ

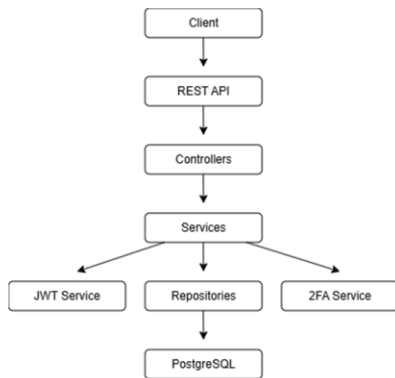
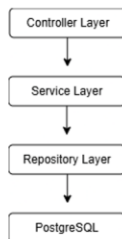


Рисунок В.5 – Слайд 5

АРХИТЕКТУРА CONTROLLER-SERVICE-REPOSITORY



Рівень	Призначення
Controller	HTTP requests
Service	Бізнес-логіка
Repository	Робота з БД

Рисунок В.6 – Слайд 6

МОДЕЛЬ БАЗИ ДАНИХ

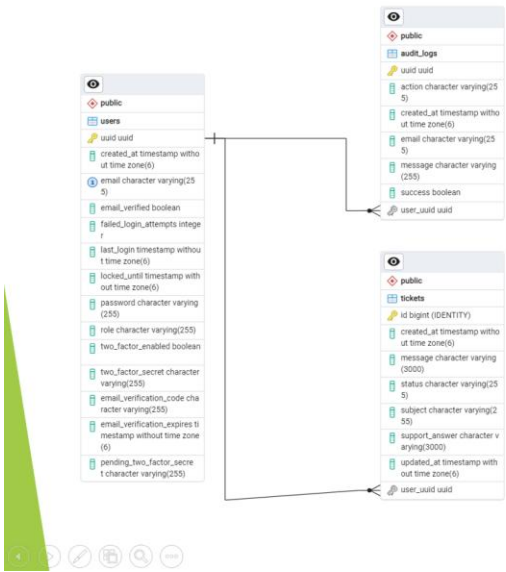
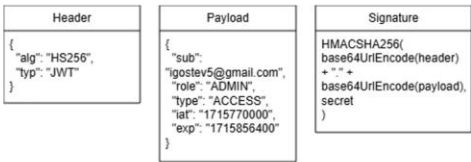


Рисунок В.7 – Слайд 7

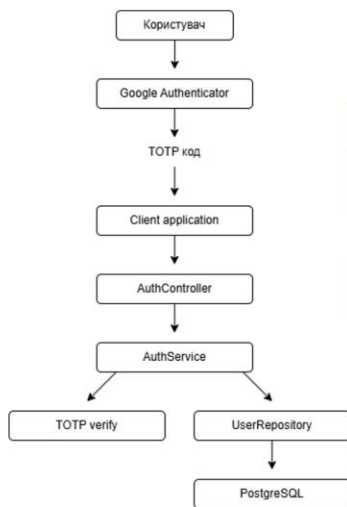
JWT-АВТОРИЗАЦІЯ



Claim	Призначення
sub	email
role	Роль користувача
type	ACCESS / 2FA_PENDING
exp	Час дії

Рисунок В.8 – Слайд 8

ДВОФАКТОРНА АВТЕНТИФІКАЦІЯ



Компонент	Призначення
Google Authenticator	Генерація TOTP
tempToken	Проміжна авторизація
JwtUtils	Генерація токенів

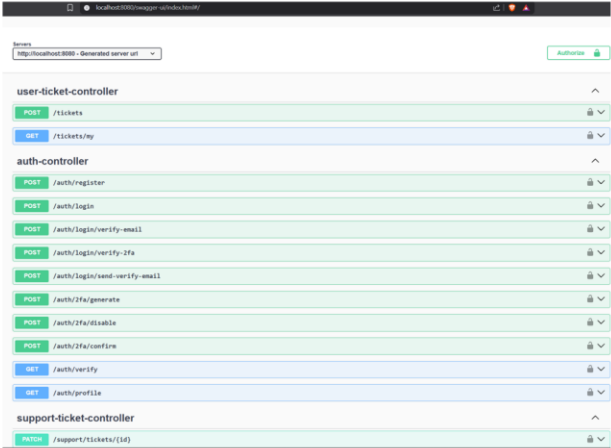
Рисунок В.9 – Слайд 9

ЗАХИСТ СИСТЕМИ

Механізм	Призначення
BCrypt	Захист паролів
JWT	Авторизація
2FA	Другий фактор
failedLoginAttempts	Захист brute-force
lockedUntil	Тимчасове блокування
RBAC	Контроль доступу

Рисунок В.10 – Слайд 10

REST API TA SWAGGER UI



Endpoint	Method	Доступ
/auth/login	POST	Public
/auth/profile	GET	Authorized
/support/tickets	GET	SUPPORT
/admin/**	GET	ADMIN

11

Рисунок В.11 – Слайд 11

РЕЗУЛЬТАТИ ТЕСТУВАННЯ

Тест	Результат
JWT validation	Успішно
Invalid password	401
Invalid role	403
Invalid TOTP	Access denied
tempToken restriction	Успішно
Ticket authorization	Успішно

12

Рисунок В.12 – Слайд 12

ВИСНОВКИ

- Проведено аналіз сучасних систем автентифікації та авторизації;
- Обґрунтовано вибір технологій Spring Boot, JWT, PostgreSQL та TOTP;
- Спроектовано архітектуру серверного застосунку на основі моделі Controller–Service–Repository;
- Реалізовано JWT-авторизацію та двофакторну автентифікацію систему рольового контролю доступу (RBAC), систему підтримки користувачів на основі тікетів;
- Проведено тестування REST API через Swagger UI;
- Підтверджено коректність роботи серверного застосунку.



Рисунок В.13 – Слайд 13