Міністерство освіти і науки України
Харківського національного університету імені В.Н. Каразіна
Навчально-наукового інституту комп’ютерних наук та штучного інтелекту
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

В.о. зав. кафедрою КІСМіТ
Марина ЄСІНА
“Допущено до захисту”
«   »__________2025р.

Пояснювальна записка

до кваліфікаційної роботи бакалавра
на тему: «Аналіз та розробка безпечних веб-додатків»

оцінка « ________ »

Голова ЕК
Мичуда Л.З.

Керівник: к.т.н. доцент
Мелкозьорова О.М.

Рецензент: д.т.н. с.н.с. Толстолузька О.Г.

Виконавець: студент групи КБ-42
Манцов М.К.

Харків - 2025

РЕФЕРАТ

Пояснювальна записка до проєкту бакалавра містить 65 сторінок, 8 рисунків, 4 таблиці, 3 додатки, 21 посилання на джерела.

Мета роботи полягає в дослідженні та розробці безпечного веб-додатку на платформі Java, що включає сучасні механізми автентифікації, авторизації та шифрування даних відповідно до актуальних вимог інформаційної безпеки.

Об’єкт дослідження – веб-додатки, їх архітектура та механізми захисту.

Предмет дослідження – методи, засоби та технології захисту інформації в Java-застосунках, що працюють за клієнт-серверною архітектурою.

Основними методами дослідження є аналіз типових загроз OWASP, класифікація вразливостей за етапами життєвого циклу ПЗ, огляд архітектурних підходів, криптографічних алгоритмів і моделей контролю доступу.

У роботі досліджено: поширені вразливості веб-додатків і методи їх нейтралізації; порівняно архітектурні рішення та моделі автентифікації, описано особливості реалізації захисту в середовищі Java, спроєктовано й реалізовано багаторівневу систему безпеки, що включає JWT, ролі доступу, асиметричне та симетричне шифрування, перевірку хешів, обмеження частоти запитів і логування дій; проведено тестування на відповідність безпековим вимогам і правильність реалізації криптографічних механізмів.

Результати роботи можуть бути використані у різних наукових виданнях, а також для побудови безпечних веб-сервісів, впровадження криптографічного захисту даних і кращого розуміння сучасних практик веб-розробки.

Ключові слова: JAVA, ВЕБ-ДОДАТОК, АВТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ, JWT, AES, RSA, ХЕШУВАННЯ, ІНФОРМАЦІЙНА БЕЗПЕКА, ШИФРУВАННЯ, SPRING SECURITY, REST API.

ABSTRACT

The explanatory note to the bachelor's project contains 65 pages, 8 figures, 4 tables, 3 appendices, 21 references.

The purpose of the work is to research and develop a secure web application on the Java platform, including modern mechanisms for authentication, authorisation and data encryption in accordance with current information security requirements.

The object of research is web applications, their architecture and security mechanisms.

The subject of the study is methods, tools and technologies for protecting information in Java applications that operate on a client-server architecture.

The main research methods are the analysis of typical OWASP threats, classification of vulnerabilities by stages of the software life cycle, review of architectural approaches, cryptographic algorithms and access control models.

The paper investigates: common vulnerabilities of web applications and methods of their neutralisation; compares architectural solutions and authentication models; describes the features of security implementation in the Java environment; designs and implements a multi-level security system that includes JWT, access roles, asymmetric and symmetric encryption, hash checking, limiting the frequency of requests and logging actions; tests for compliance with security requirements and correct implementation of cryptographic mechanisms.

The results of the work can be used in various scientific publications, as well as for building secure web services, implementing cryptographic data protection and better understanding of modern web development practices.

Keywords: JAVA, WEB APPLICATION, AUTHENTICATION, AUTHORISATION, JWT, AES, RSA, HASHING, INFORMATION SECURITY, ENCRYPTION, SPRING SECURITY, REST API.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 6
ВСТУП..... 7
1 АНАЛІЗ СУЧАСНИХ ЗАГРОЗ БЕЗПЕКИ У ВЕБ-ДОДАТКАХ..... 9
1.1 Огляд сучасних векторів атак на веб-додатки відповідно до OWASP Top 10 9
1.2 Класифікація вразливостей у веб- та клієнт-серверних застосунках 11
1.3 Методи нейтралізації загроз та побудови захищених веб-додатків 15
2 ПІДХОДИ ДО ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ У ВЕБ-ДОДАТКАХ..... 18
2.1 Архітектурні моделі веб-додатків і їхній вплив на безпеку 18
2.2 Моделі автентифікації, авторизації та управління доступом 24
2.3 Огляд криптографічних алгоритмів: симетричні й асиметричні 29
2.4 Принципи захисту даних у Java: шифрування, хешування, цілісність 32
3 ПРОЄКТУВАННЯ І РЕАЛІЗАЦІЯ БЕЗПЕЧНОГО ВЕБ-ДОДАТКА НА JAVA 36
3.1 Постановка задачі, технічні та безпекові вимоги 36
3.2 Архітектура веб-додатка..... 37
3.3 Реалізація автентифікації користувачів 39
3.4 Схема шифрування повідомлень та розподілу ключів 40
3.5 Проєктування перевірки цілісності повідомлень 41
3.6 Збереження облікових записів і хешування паролів 42
3.7 Логування подій і аудит безпеки 43
3.8 Засоби адміністративного контролю та безпеки..... 43
4 ТЕСТУВАННЯ І ОЦІНКА БЕЗПЕКИ РОЗРОБЛЕНОГО ВЕБ-ДОДАТКА 45
4.1 Методика тестування веб-додатку 45
4.2 Перевірка стійкості системи до атак 52
4.3 Тестування правильності роботи шифрування повідомлень 62
4.4 Перспективи подальшого розвитку веб-додатка 64
ВИСНОВКИ..... 65
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 67

ДОДАТОК А 70
ДОДАТОК Б 72
ДОДАТОК В 80

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

XSS – Cross-Site Scripting
SQL – Structured Query Language
HTTPS – HyperText Transfer Protocol Secure
XML – eXtensible Markup Language
CI/CD – Continuous Integration / Continuous Delivery
ПЗ – програмне забезпечення
IP – Internet Protocol
ОС – операційна система
БД – база даних
API – Application Programming Interface
HTML – HyperText Markup Language
CSS – Cascading Style Sheets
JWT – JSON Web Token
OIDC – OpenID Connect
IdP – Identity Provider
RSA – Rivest–Shamir–Adleman
TLS – Transport Layer Security
UML – Unified Modeling Language
SSL – Secure Sockets Layer
CSRF – Cross-Site Request Forgery
JPA – Java Persistence API
KMS – Key Management Service
SSE – Server-Sent Events

ВСТУП

У сучасному цифровому світі веб-додатки є основою роботи тисяч онлайн-сервісів, платформ і систем, без яких уже складно уявити повсякденне життя. Вони використовуються у всіх сферах діяльності: від електронного банкінгу та державних послуг до корпоративних інформаційних систем, онлайн-магазинів і платформ для спілкування. Саме через браузер або мобільний інтерфейс користувачі щоденно взаємодіють із веб-сервісами, залишаючи особисту інформацію, здійснюючи платежі або працюючи з конфіденційними даними.

Разом із розширенням функціональності веб-додатків стрімко зростає і кількість кіберзагроз, що на них спрямовані. Вразливості застосунків стали однією з основних причин витоків даних, несанкціонованих доступів, фінансових втрат та підриву довіри до онлайн-сервісів. Зловмисники все частіше використовують атаки на рівні застосунку, такі як SQL-ін’єкції, міжсайтове виконання скриптів (XSS), підміна сесій, атаки типу «людина посередині» та інші. Часто це відбувається не через відсутність інструментів захисту, а через неправильну або неповну їх реалізацію.

З огляду на це, проблема захисту веб-додатків виходить на перший план. Розробники мають не лише створювати зручні інтерфейси і стабільну бізнес-логіку, а й впроваджувати комплексні та перевірені механізми захисту даних користувачів. Особливого значення набуває правильна побудова архітектури безпеки, починаючи з автентифікації та шифрування, і закінчуючи логуванням подій та захистом від повторного використання сесій.

Java є однією з найпопулярніших мов програмування для створення надійних веб-додатків. Завдяки широкій підтримці безпекових бібліотек, таких як Spring Security, та наявності потужних криптографічних засобів, Java дозволяє розробникам реалізовувати сучасні підходи до захисту, включно з багаторівневою автентифікацією, шифруванням трафіку, хешуванням паролів та аудитом.

Актуальність цієї теми зумовлена постійною появою нових загроз, на які мають бути здатні реагувати як платформи, так і окремі розробники. Просте використання HTTPS або захист за допомогою паролю сьогодні вже не є достатніми — необхідно проєктувати додатки з урахуванням принципів «безпека за замовчуванням» та «мінімальні привілеї», а також проводити тестування захищеності у процесі розробки.

Метою даної дипломної роботи є дослідження сучасних підходів до безпеки веб-додатків, а також проєктування та реалізація власного клієнт-серверного веб-додатка з використанням ефективних механізмів захисту, таких як автентифікація користувачів, шифрування даних і контроль доступу.

Для досягнення цієї мети в межах роботи буде виконано:

- аналіз найпоширеніших загроз, пов'язаних з веб-додатками;
- класифікацію типів вразливостей за етапами розробки;
- аналіз ключових концепції, що впливають на рівень захисту системи;
- спроєктовано архітектуру захищеного веб-додатку;
- реалізовано механізми автентифікації, ролей, шифрування, перевірки хешів та захисту від перевантаження;
- проведено комплексне тестування на типові атаки, а також перевірку правильності реалізації криптографічної частини.

Дана робота спрямована на об'єднання теоретичних основ інформаційної безпеки із практичними інструментами сучасної Java-розробки для створення веб-додатку, який відповідає сучасним вимогам до захисту персональних даних та взаємодії клієнта із сервером.

1 АНАЛІЗ СУЧАСНИХ ЗАГРОЗ БЕЗПЕКИ У ВЕБ-ДОДАТКАХ

1.1 Огляд сучасних векторів атак на веб-додатки відповідно до OWASP Top 10

Для аналізу актуальних векторів атак розглянемо підхід, запропонований міжнародною спільнотою OWASP. На відміну від загальних класифікацій вразливостей, які розглядаються далі, OWASP Top 10 відображає практичну статистику найбільш критичних проблем безпеки, що фіксуються у реальних веб-додатках. Це дозволяє зосередитися саме на тих загрозах, які мають найбільший вплив на сучасні системи.

OWASP — це глобальна некомерційна організація, яка об'єднує експертів з безпеки та розробників, що прагнуть підвищити рівень захисту веб-додатків. Звіти OWASP стали орієнтиром для спільноти, зосередженої на питаннях безпеки. Вони формують тенденції на ринку захисту сучасних інтелектуальних веб-сервісів.

Перший звіт OWASP Top 10 був опублікований у 2003 році, а подальші оновлення виходили приблизно кожні 3 роки: у 2004, 2007, 2010, 2013 та 2017 роках. Остання редакція була оприлюднена у 2021 році – на яку ми і будемо орієнтуватись [2].

Оновлений список OWASP Top 10 за 2021 рік включає десять найкритичніших категорій вразливостей, що найчастіше зустрічаються у веб-додатках. До переліку входять як давно відомі проблеми, так і нові категорії, пов'язані зі змінами у підходах до розробки, впровадження та захисту вебсервісів:

1) A01:2021 — Порушення контролю доступу.

Описує ситуації, коли додаток не обмежує доступ до функціоналу або ресурсів належним чином. Це дозволяє неавторизованим користувачам отримувати доступ до даних або змінювати чужі ресурси. Така вразливість зустрічається дуже часто і має критичні наслідки, якщо її не усунуто [3].

2) A02:2021 — Криптографічні збої.

Ця категорія охоплює проблеми, пов’язані з використанням ненадійних або застарілих криптографічних алгоритмів, а також зберіганням даних без шифрування. Основна увага приділяється правильному використанню криптографії для захисту конфіденційної інформації [3].

3) A03:2021 — Ін’єкції.

Включає всі типи вразливостей, що виникають внаслідок введення несанкціонованих команд через незахищені поля. Прикладом є SQL-ін’єкції або XSS. Основна небезпека полягає у можливості виконання довільного коду або модифікації даних [3].

4) A04:2021 — Небезпечне проєктування.

Новий тип загроз, пов’язаних з помилками на ранніх етапах проєктування: відсутністю моделювання загроз, нехтуванням принципами захищеної архітектури або неправильним трактуванням вимог безпеки [3].

5) A05:2021 — Неправильна конфігурація.

Ця категорія охоплює найрізноманітніші помилки в налаштуваннях: відкриті порти, невимкнені налаштування розробника, наявність тестових функцій у продуктивному середовищі. До неї також віднесено загрози, що раніше стосувалися зовнішніх XML-об'єктів [3].

6) A06:2021 — Уразливі та застарілі компоненти.

Вразливості, пов’язані з використанням залежностей або сторонніх бібліотек, які більше не оновлюються або містять відомі помилки. Це поширена загроза, яку складно виявити автоматичними засобами без спеціальних перевірок ланцюжка залежностей [3].

7) A07:2021 — Помилки ідентифікації та автентифікації.

Категорія охоплює проблеми, пов’язані з автентифікацією користувачів: слабкі паролі, неправильна обробка сесій, відсутність багатофакторної автентифікації. Забезпечення надійного контролю доступу починається з правильної реалізації цих механізмів [3].

8) A08:2021 — Порушення цілісності програмного забезпечення та даних.

Фокусує увагу на небезпеці, що виникає при автоматичному оновленні ПЗ або обробці критичних даних без перевірки їхньої цілісності. До прикладів належать: недостовірні оновлення, небезпечна десеріалізація або скомпрометовані CI/CD-процеси [3].

9) A09:2021 — Недоліки журналювання та моніторингу безпеки.

Відсутність належного логування безпекових подій або механізмів сповіщення про інциденти значно ускладнює виявлення атак. У разі виникнення інциденту компанія може не мати жодного інструменту для аналізу та реагування [3].

10) A10:2021 — Підробка запитів на стороні сервера.

Дозволяє зловмиснику змусити сервер виконувати запити до внутрішніх ресурсів або сервісів. Це може призвести до доступу до приватної інфраструктури, яка зазвичай не доступна зовні [3].

Перелік вразливостей OWASP Top 10 дозволяє зосередити увагу на найнебезпечніших і найпоширеніших векторах атак, які загрожують сучасним веб-додаткам. Водночас для глибшого розуміння природи цих загроз доцільно розглядати їх у структурному контексті — з урахуванням того, на яких етапах створення програмного забезпечення вони виникають і як проявляються у клієнт-серверних архітектурах.

1.2 Класифікація вразливостей у веб- та клієнт-серверних застосунках

Вразливості веб-додатків можна класифікувати відповідно до фази життєвого циклу розробки програмного забезпечення наступним чином: аналіз вимог, проєктування, реалізація та впровадження.

1) Вразливості, пов’язані з аналізом вимог [1].

До цього класу належать загрози, які виникають через відсутність чітких вимог безпеки або помилки у формуванні доступу та перевірці користувача:

- Порушення контролю доступу: дозволяє отримати доступ до захищених функцій, наприклад до адміністративних інтерфейсів або чужих облікових записів.
- Зловживання функціональністю: використання штатних можливостей (наприклад, відновлення паролю) для отримання несанкціонованої інформації.
- Неналежна обробка помилок: система повертає повідомлення, що розкривають структуру бази даних, стеки викликів тощо.

2) Вразливості, пов'язані з етапом проєктування [1].

Ці вразливості пов'язані з логічними та архітектурними недоліками, коли система не враховує типові вектори атак:

- Атака методом перебору: атакуючий підбирає комбінації логінів/паролів без обмежень за кількістю спроб.
- Підробка міжсайтового запиту: зловмисник змушує автентифікованого користувача виконати дію без його відома.
- Витік інформації: чутлива інформація (наприклад, IP, логіка дій) може бути виявлена у вихідному коді або коментарях.
- Недостатня автентифікація: система дозволяє доступ без адекватної перевірки користувача.

3) Вразливості, пов'язані з етапом реалізації [1].

Найчастіше зустрічаються через помилки програмування, нехтування валідацією вводу або небезпечного використання сторонніх бібліотек:

- Переповнення буфера: дозволяє виконання довільного коду.
- Підміна вмісту: користувачу демонструється підроблений контент у довіреному середовищі.
- Прогнозування облікових/сесійних даних: сесійні ідентифікатори або паролі можна передбачити через слабкі генератори.
- Міжсайтове скриптування: впровадження зловмисного скрипту через поля введення.

- Відмова в обслуговуванні: систематичне перевантаження сервера або бази даних.
- Впровадження команд: використання вхідних даних для виконання команд операційної системи.
- Обхід шляхів: доступ до файлів за межами дозволеного каталогу (наприклад, ../../etc/passwd).
- SQL-ін'єкція: впровадження SQL-запитів через неконтрольовані параметри користувача.

4) Вразливості, пов'язані з етапом впровадження [1].

Цей тип вразливостей виникає через неправильну конфігурацію веб-сервера, недостатній контроль завершення сесій або надто слабкі налаштування безпеки за замовчуванням:

- Недостатнє завершення сесії: сесія залишається активною після виходу користувача з системи.
- Неправильна конфігурація застосунку: використання небезпечних налаштувань (наприклад, відкриті порти, активне повідомлення про версію PHP).
- Фіксація сесії: сесійний ідентифікатор встановлюється до автентифікації та залишається незмінним після входу.

Окрім описаної класифікації за етапами життєвого циклу веб-додатка, існують й інші підходи до поділу вразливостей. Наприклад, розрізняють вразливості за типом впливу (на конфіденційність, цілісність або доступність даних), або за тим, у якій частині системи вони виникають — на стороні клієнта чи сервера. Також можна зустріти класифікації за способом реалізації атак чи рівнем ризику. Проте у цій роботі було обрано саме підхід, що базується на фазах створення програмного забезпечення, оскільки він дозволяє логічно показати, де і як можуть виникати помилки ще на ранніх етапах розробки.

Для наочності, класифікацію вразливостей веб-застосунків, відповідно до фаз розробки програмного забезпечення, наведено нижче у таблиці 1.1, що відображає основні типи загроз та можливі наслідки їх реалізації.

Таблиця 1.1 – Класифікація вразливостей веб-додатків за фазою життєвого циклу розробки [1]

Фаза розробки	Тип вразливості	Можливі наслідки
Аналіз вимог	Порушення контролю доступу	Доступ до конфіденційних або обмежених даних; виконання несанкціонованих операцій; підміна даних
	Зловживання функціональністю	Повторне виконання дій поза передбаченою логікою; отримання несанкціонованого доступу; зміна стану системи
	Неналежна обробка помилок	Розкриття конфіденційної або службової інформації; обхід механізмів захисту; підвищення привілеїв
Проєктування	Атака методом перебору	Підбір облікових даних; компрометація облікового запису; несанкціонований доступ до системи
	Підробка міжсайтового запиту	Виконання дій від імені користувача без його згоди; зміна налаштувань; викрадення даних
	Витік інформації	Доступ до конфігурацій, внутрішніх API, IP-адрес або структури системи; збір метаданих для подальших атак
	Недостатня автентифікація	Вхід до системи без підтвердження особи; обхід захисту на основі сесій або токенів
Реалізація	Переповнення буфера	Пошкодження пам’яті; виконання довільного коду; зупинка сервера
	Підміна вмісту	Зміна збережених або переданих даних; фальсифікація повідомлень; порушення цілісності інформації
	Прогнозування облікових/сесійних даних	Викрадення сесії; повторне використання токена; отримання доступу до стороннього акаунта

Продовження таблиці 1.1

Реалізація	Міжсайтове скриптування	Викрадення сесій або облікових даних; підміна контенту; виконання шкідливого коду в браузері
	Відмова в обслуговуванні	Перевантаження сервера; порушення доступності системи; зупинка роботи додатку
	Впровадження команд	Виконання системних команд на сервері; повний контроль над ОС; викрадення даних
	Обхід шляхів	Доступ до службових або конфіденційних файлів; зчитування або зміна вмісту директорій
	SQL-ін’єкція	Виконання довільних SQL-запитів; доступ до БД; знищення або зміна даних; викрадення облікових записів
Впровадження	Недостатнє завершення сесії	Збереження активної сесії після виходу; несанкціонований доступ до профілю користувача
	Неправильна конфігурація застосунку	Відкритий доступ до службових інтерфейсів; використання небезпечних налаштувань за замовчуванням; витік інформації
	Фіксація сесії	Повторне використання токена; нав’язування стороннього ідентифікатора сесії; підміна користувача

1.3 Методи нейтралізації загроз та побудови захищених веб-додатків

У попередньому пункті було розглянуто, які типи вразливостей можуть виникати на різних етапах розробки веб-додатка. На практиці кожна загроза має свої способи протидії — технічні, архітектурні або організаційні. У деяких випадках достатньо правильно налаштувати систему, в інших — потрібне більш комплексне рішення, наприклад, використання багатофакторної автентифікації чи шифрування даних.

Нижче подано таблицю 1.2, в якій для кожного типу вразливості наведено приклади методів захисту [1],[4],[5].

Таблиця 1.2 – Методи нейтралізація різних типів вразливостей

Тип вразливості	Методи нейтралізації
Порушення контролю доступу	Розмежування доступу за ролями; перевірка прав на сервері; заборона доступу за замовчуванням
Зловживання функціональністю	Обмеження на повторне виконання дій; перевірка логіки сценаріїв; журналювання аномальної активності
Неналежна обробка помилок	Глобальна обробка винятків; приховування технічної інформації; повернення узагальнених повідомлень
Атака методом перебору	Обмеження кількості спроб входу; CAPTCHA; блокування після декількох невдалих спроб авторизації; двофакторна автентифікація (MFA)
Підробка міжсайтового запиту	CSRF-токени; використання SameSite cookies; перевірка заголовків запиту
Витік інформації	Шифрування даних; мінімізація відповідей; приховування структури системи; контроль доступу
Недостатня автентифікація	Використання складних паролів та MFA; обмеження тривалості сесій
Переповнення буфера	Перевірка розміру введених даних; використання безпечних API; контроль пам’яті
Підміна вмісту	Перевірка цілісності даних; цифрові підписи; валідація контенту
Прогнозування облікових/сесійних даних	Генерація токенів з використанням криптостійких алгоритмів; забезпечення випадковості значень; регулярне оновлення ідентифікаторів
Міжсайтове скриптування	Заміна небезпечних символів у виведенні; перевірка та очищення введених даних; обмеження виконання сторонніх скриптів у браузері

Продовження таблиці 1.2

Відмова в обслуговуванні	Лімітування запитів; захист від надмірної активності; обмеження ресурсів
Впровадження команд	Перевірка введених даних; заборона на виконання системних команд; дозвіл лише на перелік явно визначених дій
Обхід шляхів	Нормалізація введених шляхів; перевірка дозволів доступу; відкидання підозрілих шаблонів
SQL-ін'єкція	Параметризовані запити; використання ORM; валідація введених даних
Недостатнє завершення сесії	Автоматичне завершення сесії після неактивності; внесення недійсних токенів до списку заборонених; перевірка чинності активних сеансів
Неправильна конфігурація застосунку	Використання безпечних налаштувань за замовчуванням; вимкнення debug-режимів; аудит
Фіксація сесії	Генерація нового токена після логіну; заборона повторного використання старих токенів

На практиці ці методи використовуються не ізольовано, а як частини узгоджених політик безпеки, що враховують технічні, організаційні та процедурні аспекти. Їх впровадження дозволяє суттєво зменшити вразливість систем до поширених атак і забезпечити надійну роботу веб-додатка навіть за умов активного зовнішнього впливу.

2 ПІДХОДИ ДО ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ У ВЕБ-ДОДАТКАХ

2.1 Архітектурні моделі веб-додатків і їхній вплив на безпеку

Архітектура веб-додатку визначає загальну структуру системи, спосіб взаємодії її компонентів і розподіл функціональних обов'язків між ними. Від вибору архітектурної моделі значною мірою залежить не лише масштабованість та продуктивність системи, а й її здатність протистояти поширеним загрозам. Різні архітектурні підходи по-різному впливають на реалізацію контролю доступу, ізоляцію модулів, обробку даних та управління вразливостями. Далі розглянемо найбільш поширені моделі побудови веб-додатків, а також проаналізуємо їхні особливості з точки зору безпеки.

Однією з найдавніших і найпростіших моделей побудови веб-додатків є монолітна архітектура. Як випливає з назви, вона передбачає єдину цілісну структуру, у якій тісно пов'язані між собою всі основні компоненти: база даних, серверна логіка та клієнтська частина. Це означає, що будь-які зміни в одному компоненті неминуче впливають на інші, що може спричиняти додаткові труднощі під час супроводу або масштабування додатку [6].

Серед переваг монолітної архітектури варто відзначити [6]:

- Зручність управління міжфункціональними процесами, оскільки вся система працює як єдине ціле;
- швидке тестування, адже перевіряється вся система одночасно;
- простоту розгортання завдяки єдиній структурі;
- легке керування базою даних в умовах єдиного середовища.

Однак та сама простота породжує і низку недоліків [6]:

- обмежена масштабованість: неможливо масштабувати окрему частину системи без масштабування всієї архітектури;
- ускладнене внесення змін: модифікація одного модуля потребує перевірки сумісності з усією системою;

- знижена надійність: помилка в одному компоненті може призвести до збою всієї програми.

Сучасні вимоги до гнучкості та масштабованості систем роблять монолітну архітектуру менш привабливою для складних веб-додатків. Проте вона цілком придатна для невеликих проєктів або локальних внутрішніх систем, де простота та швидкість мають вищий пріоритет за масштабованість [6].

Водночас, з точки зору безпеки, а особливо тестування, монолітна архітектура має свої переваги. Насамперед, площа потенційної атаки зазвичай менша порівняно з більш розподіленими системами, де кожен мікросервіс є окремою ціллю. Менша кількість зовнішніх залежностей робить систему більш передбачуваною та спрощує аналіз вразливостей. У монолітному застосунку вся бізнес-логіка та внутрішня взаємодія між модулями відбувається локально, без потреби в численних API-запитах між окремими компонентами, що знижує ризики та полегшує налаштування захисту. Також моноліт легше описати й обмежити при впровадженні політик захисту даних, мережевої безпеки [7].

Втім, недоліком монолітної архітектури може бути повільніший процес усунення вразливостей. Через високу пов’язаність компонентів локалізувати та виправити проблему часто складніше, ніж у мікросервісній архітектурі, де окремі сервіси оновлюються незалежно, і їх можна перевіряти та відновлювати поодинці — за умови доступу до вихідного коду [7].

На відміну від монолітної, мікросервісна архітектура базується на розподілі системи на декілька незалежних, слабо пов’язаних між собою компонентів — мікросервісів. Кожен з них виконує окреме завдання, може бути реалізований на різних мовах програмування та взаємодіє з іншими компонентами через API. Такий підхід значно підвищує гнучкість системи та спрощує роботу розробників, даючи змогу працювати з кожним елементом окремо [6].

Серед основних переваг мікросервісної архітектури варто виділити [6]:

- можливість незалежного розгортання кожного мікросервісу;

- використання різноманітних технологій і простоту оновлення окремих компонентів;
- вищу надійність завдяки ізольованості кожної складової;
- гнучке масштабування, що дозволяє адаптувати лише ті частини системи, які цього потребують.

Водночас, ця архітектурна модель має і свої недоліки [6]:

- ускладнене управління через велику кількість окремих модулів;
- більш складне тестування, адже кожен мікросервіс потребує окремої перевірки;
- ризик зниження продуктивності через перевантаження каналів взаємодії між сервісами;
- наявність так званих «перехресних викликів», які складніше координувати.

Загалом, мікросервісна архітектура є доцільним вибором для великих, складних та динамічних систем, які обробляють великі обсяги трафіку й постійно розвиваються. Проте для ефективної роботи з такою архітектурою необхідна команда з відповідним досвідом, здатна забезпечити правильну організацію сервісів і підтримку стабільної взаємодії між ними [6].

Переваги мікросервісної архітектури необхідно співвідносити з численними потенційними проблемами безпеки, які виникають через поверхню атаки, розподілену між десятками, а іноді й сотнями незалежних сервісів. На рівні інфраструктурної безпеки кожен окремий сервіс зазвичай працює у власному хмарному контейнері, а конкретні екземпляри сервісів оркеструються за допомогою Kubernetes або аналогічного рішення, що робить безпеку хмари та спостережуваність критично важливими [7].

Оскільки всі компоненти застосунку взаємодіють через API-запити, забезпечення безпеки API також є першочерговим завданням як для експлуатації, так і для тестування, причому особливу увагу слід приділити автентифікації. Через те, що

кожен сервіс є окремою ціллю, моніторинг виступає ще однією слабкою ланкою, адже зловмисники можуть атакувати окремі сервіси, не викликаючи сигналів тривоги про те, що сам застосунок перебуває під атакою [7].

Як уже згадувалося раніше, однією з переваг мікросервісів є те, що ізолювати проблему безпеки в межах конкретного сервісу може бути простіше, ніж у монолітному застосунку. Навіть якщо немає змоги одразу усунути основну вразливість, легше налаштувати міжмережевий екран або повністю заблокувати сервіс до моменту усунення. Оскільки сервіси зазвичай функціонують автономно, ризик повної компрометації всієї системи у випадку зламу одного з компонентів суттєво нижчий [7]. Схематичне порівняння монолітної та мікросервісної архітектур наведено на рис. 2.1.

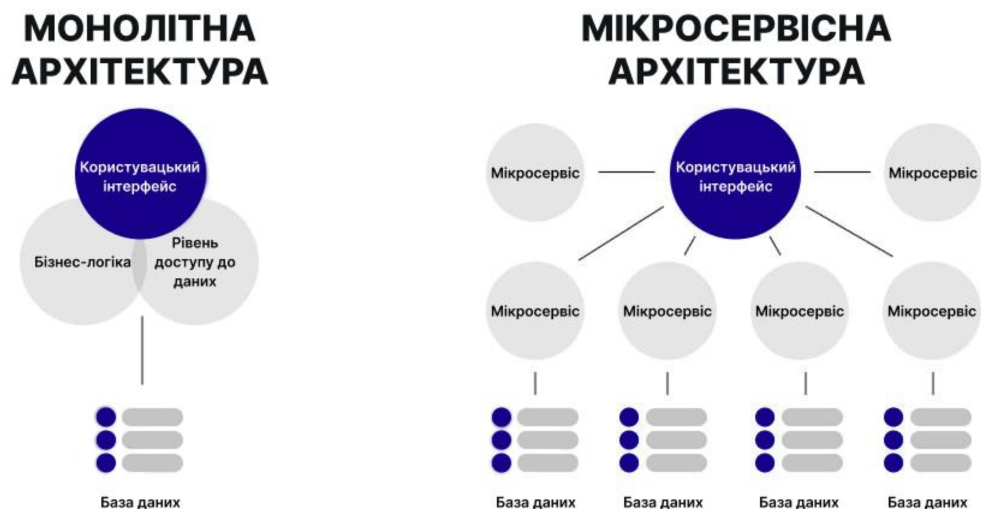

Рисунок 2.1 – Порівняльна схема монолітної та мікросервісної архітектур

Якщо архітектурні стилі, такі як моноліт або мікросервіси, описують спосіб організації компонентів на рівні системи загалом, то інша важлива перспектива — це рівнева модель побудови. Вона визначає, як саме структуровані функціональні частини застосунку: від інтерфейсу користувача до обробки даних. Найпоширенішими є дворівнева (2-tier) та трирівнева (3-tier) моделі, кожна з яких має свої переваги та недоліки для безпеки веб-додатків.

Дворівнева архітектура — це початкова форма клієнт-серверної архітектури, яка складається з презентаційного рівня та рівня даних (рис. 2.2). Бізнес-логіка знаходиться у або у презентаційному рівні або у рівні даних, або ж відразу в обох. У дворівневій архітектурі презентаційний рівень, а отже, і кінцевий користувач має прямий доступ до рівня даних, а бізнес-логіка часто є обмеженою. [8].

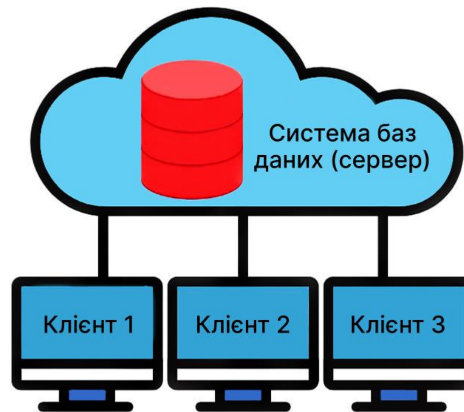

Рисунок 2.2 – Схема дворівневої архітектури

Проте цей підхід має кілька проблемних моментів. По-перше — безпека. Через пряму взаємодію між базою даних і пристроєм користувача зростає ймовірність порушень безпеки та зловмисних атак. По-друге — масштабованість. Якщо кількість користувачів збільшується, то продуктивність системи знижується, що негативно впливає на досвід користувача [7].

Саме тому все більше компаній переходять на сучаснішу архітектуру веб-додатків, відому як трирівнева (3-Tier) архітектура (рис. 2.3). На відміну від традиційної, ця модель містить три рівні [7]:

1) Презентаційний рівень виступає інтерфейсом користувача та слугує комунікаційним шаром, через який здійснюється взаємодія з системою. Його основне призначення — відображення інформації та збирання введених користувачем даних.

Цей рівень може реалізовуватись як веб-інтерфейс у браузері, графічний інтерфейс настільного застосунку або інша форма візуального представлення. Для створення веб-інтерфейсів зазвичай використовуються HTML, CSS і JavaScript, тоді

як настільні застосунки можуть бути розроблені на різних мовах програмування залежно від операційної системи та середовища виконання [8].

2) Рівень застосунку, або рівень логіки, є центральною складовою архітектури веб-додатку. На цьому етапі обробляється інформація, отримана з презентаційного рівня, з урахуванням наявних даних і визначених бізнес-правил.

Саме тут реалізується бізнес-логіка системи: відбувається прийняття рішень, виконання сценаріїв і взаємодія між різними компонентами. Застосунковий рівень також здійснює звернення до рівня зберігання даних для читання, оновлення або видалення інформації, забезпечуючи узгоджену роботу всієї системи [8].

3) Рівень даних відповідає за зберігання, доступ і управління всією інформацією, яку обробляє веб-додаток. Саме тут знаходяться джерела даних, до яких звертається рівень логіки для зчитування, зміни або видалення інформації. У залежності від потреб системи, на цьому рівні можуть використовуватись реляційні системи керування базами даних (наприклад, PostgreSQL або MySQL) чи нереляційні NoSQL-рішення, такі як MongoDB [8].

Рисунок 2.3 – Схема трирівневої архітектури [9]

Головною перевагою трирівневої архітектури є її логічне та фізичне розділення функціональності. Кожен рівень може працювати на окремій операційній системі та серверній платформі — наприклад, веб-сервер, сервер застосунків, сервер бази даних — що найкраще відповідає його функціональним вимогам. Кожен рівень функціонує принаймні на одному виділеному апаратному або віртуальному сервері, тож сервіси

кожного рівня можуть бути налаштовані та оптимізовані незалежно, без впливу на інші рівні [8].

Також з точки зору безпеки трирівнева архітектура має кращий захист, оскільки рівні презентації та даних не можуть взаємодіяти напряму, добре спроєктований рівень застосунку може виконувати функцію внутрішнього брандмауера, запобігаючи SQL-ін’єкціям та іншим зловмисним атакам [8].

2.2 Моделі автентифікації, авторизації та управління доступом

Автентифікація — це процес встановлення, чи є особа або об’єкт тим, за кого себе видає. Технології автентифікації контролюють доступ до систем, перевіряючи, чи є дані, введені користувачем (наприклад, логін і пароль), дійсними — тобто чи збігаються вони з тими, що зберігаються у базі даних або на сервері автентифікації [11].

Існує багато способів автентифікації веб-додатків. Розглянемо найпоширеніші з них:

1) Автентифікація на основі cookie [10].

Файли cookie зазвичай використовуються для обробки автентифікації користувачів у веб-додатках.

На рис. 2.4 наведено схему, яка показує, як це працює:

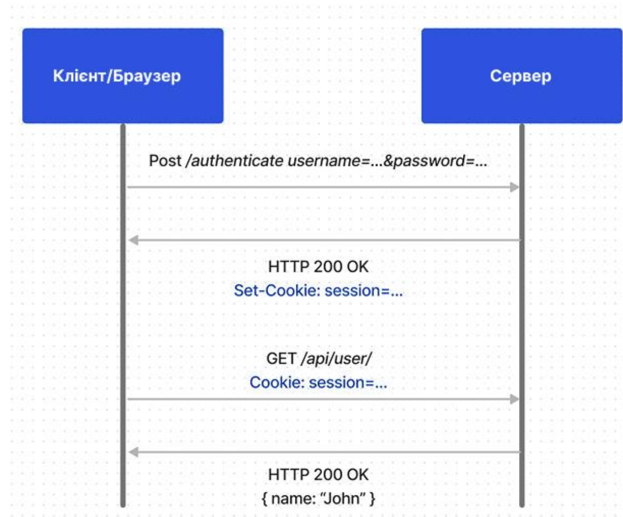

Рисунок 2.4 – Схема автентифікації на основі cookie [10]

Браузер користувача надсилає на сервер POST-запит із даними для входу. Сервер перевіряє ці облікові дані й, у разі успішної автентифікації, відповідає статусом HTTP 200 OK. Після цього сервер створює ідентифікатор сесії (session ID), зберігає його у себе й повертає клієнту у відповідь через заголовок Set-Cookie: session=….

Під час наступних запитів клієнт автоматично надсилає цей session ID разом із cookie, а сервер перевіряє його дійсність і відповідно обробляє запит. Коли користувач виходить із застосунку, ідентифікатор сесії видаляється як із клієнта, так і з сервера [10].

2) Автентифікація на основі токенів

Один із найпоширеніших способів реалізації токен-автентифікації — це використання JSON Web Token (JWT). JWT — це відкритий стандарт, що дозволяє безпечно передавати інформацію між сторонами у вигляді JSON-об'єктів, які містять усі необхідні дані в зашифрованому вигляді [10]. Схему роботи токен-автентифікації наведено на рис. 2.5.

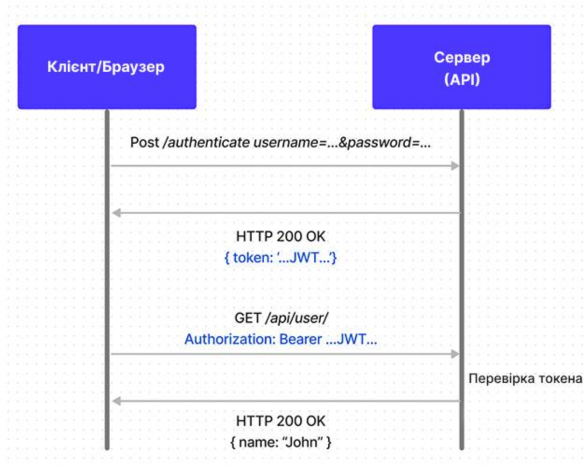

Рисунок 2.5 – Схема автентифікації на основі токенів [10]

Коли облікові дані надходять з браузера користувача, сервер їх перевіряє та генерує підписаний JWT, який містить усю необхідну інформацію про користувача. Токен є безстанним, тобто він не зберігається на сервері. У наступних запитах токен передається назад на сервер, де він декодується для перевірки достовірності [10].

3) Автентифікація через сторонній доступ (OAuth, API-токен) може працювати двома способами [10]:

- Через API-токен: зазвичай це працює аналогічно до JWT — токен передається в заголовку авторизації і обробляється на API-шлюзі для підтвердження користувача.
- Через Open Authentication (OAuth): як видно з назви, OAuth — це відкритий протокол, що дозволяє безпечно реалізовувати методи автентифікації для веб-, мобільних та десктопних додатків. Цей протокол виконує автентифікацію на сервері від імені користувача.

4) OpenID Connect (OIDC) — це надбудова над протоколом OAuth 2.0, яка спрощує процес автентифікації користувачів, додаючи до нього спеціальний авторизаційний шар. Завдяки цьому рівню застосунок може не лише перевірити особу користувача, а й отримати розширену інформацію про нього [10].

Основні переваги OIDC [10]:

- Єдиний вхід: користувачі можуть автентифікуватися через перевірених провайдерів ідентичності (IdP) (наприклад, Google або Facebook), без потреби створювати окремі облікові записи.
- Зниження ризиків безпеки: передача відповідальності за автентифікацію зовнішньому провайдеру зменшує необхідність зберігати паролі на стороні застосунку, що мінімізує ризик витоку облікових даних і несанкціонованого доступу.
- Розширена інформація про користувача: протокол дозволяє отримувати додаткові атрибути, такі як ім’я, адреса електронної пошти або налаштування профілю, що дає змогу краще персоналізувати взаємодію з користувачем.

Щоб узагальнити ключові характеристики розглянутих методів автентифікації, нижче наведено порівняльну таблицю 2.1.

Таблиця 2.1 – Порівняння методів автентифікації

Метод автентифікації	Принцип роботи	Переваги	Недоліки
На основі cookie	Ідентифікатор сесії зберігається на сервері та надсилається клієнту у cookie	Проста реалізація; підтримується всіма браузерами; автоматична відправка	Потребує захищеного зберігання сесій на сервері
На основі токенів (JWT)	Сервер генерує підписаний токен JWT, який передається клієнтом у заголовку	Безстанність; масштабованість; легка перевірка без доступу до БД	Ризик витоку токена; потреба в управлінні терміном дії
Сторонній доступ (OAuth / API-токен)	API-токен або OAuth використовується для доступу до ресурсів через API	Підходить для інтеграції з іншими сервісами; делегування прав доступу	Залежність від зовнішнього провайдера; складність налаштування
OpenID Connect (OIDC)	Надбудова над OAuth 2.0 з можливістю єдиного входу та розширеною інформацією	Покращений досвід користувача; мінімізація обробки облікових даних у системі	Потреба у зовнішньому постачальнику ідентичності; складна реалізація

Після автентифікації користувача наступним етапом є авторизація, тобто надання доступу лише до дозволених ресурсів. Правильне налаштування авторизації критично важливе для безпеки веб-додатків. Нижче описані найпоширеніші моделі авторизації [12]:

- Контроль доступу на основі ролей (RBAC – Role-Based Access Control): У цій моделі користувачам призначаються ролі (наприклад, USER, ADMIN), кожна з яких має заздалегідь визначений набір дозволів. RBAC добре підходить для систем із чіткими типами користувачів і дозволяє легко керувати правами доступу

- Контроль доступу на основі атрибутів (ABAC – Attribute-Based Access Control): У цій моделі рішення про доступ приймається на основі атрибутів користувача, запиту або середовища — наприклад, часу доби, місцезнаходження, типу пристрою чи рівня довіри. ABAC забезпечує високий рівень гнучкості та дає змогу враховувати контекст, що особливо корисно в системах із динамічними правилами доступу.
- Списки контролю доступу (ACL – Access Control Lists): ACL дозволяють задавати окремі правила доступу до кожного ресурсу (файлу, запису). Цей підхід забезпечує високий рівень деталізації, але може ускладнити адміністрування у великих системах.
- Політично орієнтоване управління доступом (PBAC – Policy-Based Access Control): PBAC використовує політики, які можуть враховувати як ролі, так і атрибути. Це дозволяє реалізовувати багаторівневу систему контролю доступу, що особливо корисно в мікросервісних архітектурах.

Щоб краще зрозуміти відмінності між цими моделями, систематизуємо їх переваги та недоліки у порівняльній таблиці 2.2.

Таблиця 2.2 – Порівняння моделей авторизації

Модель авторизації	Принцип роботи	Переваги	Недоліки
RBAC (Role-Based Access Control)	Надає доступ на основі призначених ролей (наприклад, ADMIN, USER)	Просте управління правами; зручне для ієрархічних систем	Обмежена гнучкість у разі складних або динамічних умов доступу
ABAC (Attribute-Based Access Control)	Використовує атрибути користувача або середовища (місце, час, пристрій)	Висока адаптивність; дозволи враховують контекст доступу	Складність налаштування, особливо при великій кількості атрибутів

Продовження таблиці 2.2

ACL (Access Control Lists)	Визначає дозволи окремо для кожного об’єкта або ресурсу	Детальний контроль доступу на рівні ресурсів	Погано масштабується у великих системах
PBAC (Policy-Based Access Control)	Використовує гнучкі політики, які враховують ролі, атрибути та інші умови	Багаторівневий контроль доступу, підходить для складних систем	Потребує складної реалізації і централізованого управління політиками

2.3 Огляд криптографічних алгоритмів: симетричні й асиметричні

У сучасних веб-додатках криптографія відіграє ключову роль у забезпеченні конфіденційності, цілісності та автентичності даних. Вона лежить в основі захищених з'єднань, зберігання чутливої інформації та безпечного обміну даними між клієнтами й серверами. Залежно від завдань, що стоять перед системою, застосовуються два основні типи криптографічних алгоритмів — симетричні та асиметричні.

Симетричне шифрування — це тип шифрування, при якому один і той самий ключ використовується як для шифрування, так і для дешифрування даних. Цей ключ спільно використовують відправник і одержувач, і він повинен залишатися конфіденційним для всіх інших. Симетричне шифрування відзначається високою швидкістю та ефективністю, однак має і певні недоліки. Наприклад, якщо ключ буде скомпрометовано, зловмисник зможе розшифрувати всі передані дані. Крім того, у веб-додатках виникає складність із безпечною передачею цього ключа між сторонами, що потребує додаткових механізмів захисту [13].

Щоб реалізувати симетричне шифрування у веб-додатках, найчастіше використовується алгоритм AES (Advanced Encryption Standard) - це надійний алгоритм симетричного шифрування, який широко використовується для захисту даних. Він підтримує ключі довжиною 128, 192 або 256 біт, забезпечуючи високий рівень стійкості до несанкціонованого доступу. У веб-додатках AES активно

використовується для захисту чутливих даних, шифрування файлів та безпечного обміну інформацією через інтернет [14].

Асиметричне шифрування, передбачає використання двох різних ключів — публічного та приватного. Під час передачі даних відправник шифрує повідомлення за допомогою публічного ключа одержувача, а для дешифрування ці дані може використовувати лише власник відповідного приватного ключа. Такий підхід дозволяє організувати безпечний обмін інформацією між сторонами без потреби у попередньому обміні спільним секретним ключем, що особливо важливо у розподілених середовищах і при взаємодії з багатьма користувачами одночасно [15].

Однією з головних переваг асиметричного шифрування є усунення потреби в безпечному каналі для передачі ключів, що суттєво знижує ризики перехоплення. Крім того, цей підхід дозволяє реалізовувати цифрові підписи, які забезпечують перевірку цілісності даних і підтвердження їх походження, — тобто гарантують, що повідомлення не було змінено та справді надійшло від заявленого відправника [15].

Алгоритм RSA є одним із найпоширеніших прикладів асиметричного шифрування, який широко використовується для захисту даних під час передачі через ненадійні або відкриті мережі. Його криптографічна стійкість базується на складності факторизації великих чисел, що є добутком двох простих чисел. Якщо множення таких чисел виконується швидко, то обернена операція — розклад на множники — потребує значних обчислювальних ресурсів, що й забезпечує надійність цього методу.

Пара ключів (публічний і приватний) в RSA генерується за допомогою математичних алгоритмів, які гарантують їх унікальність і криптографічну стійкість. У практичному застосуванні найчастіше використовуються ключі довжиною 1024 або 2048 біт. Зважаючи на розвиток обчислювальних потужностей, для забезпечення більшого рівня захисту рекомендовано використовувати ключі довжиною щонайменше 2048 біт [16].

Попри високу безпеку, асиметричне шифрування має низку недоліків. Воно є значно повільнішим за симетричне, складнішим у реалізації через потребу в

управлінні парами ключів, і неефективним для шифрування великих обсягів даних. Також існує загроза втрати стійкості до атак із розвитком квантових технологій.

Щоб поєднати переваги обох підходів, у сучасних веб-додатках широко застосовується гібридне шифрування — коли асиметричне шифрування використовується лише для передачі симетричного ключа, а безпосереднє шифрування даних виконується швидким симетричним алгоритмом [17].

Найпоширенішим прикладом використання гібридного шифрування в реальних системах є протокол HTTPS, що ґрунтується на стандарті TLS.

Більш детально розглянемо приклад роботи HTTPS за допомогою рис. 2.6.

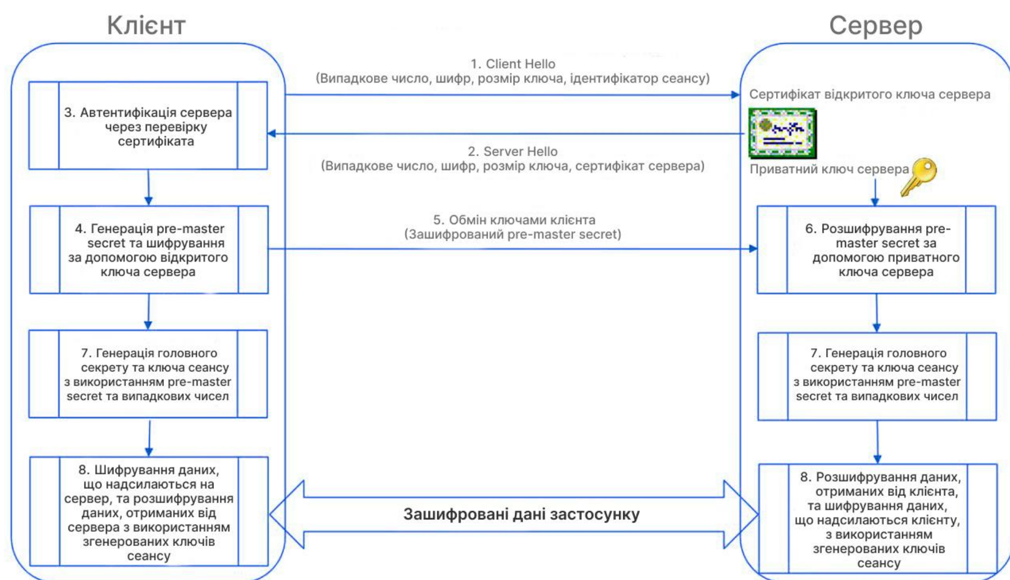

Рисунок 2.6 – Схема встановлення захищеного з’єднання в HTTPS (TLS) із використанням гібридного шифрування [18]

На рис. 2.6 зображено процес встановлення захищеного з’єднання між клієнтом і сервером у протоколі HTTPS (TLS), що ґрунтується на принципах гібридного шифрування. Коли клієнт ініціює з’єднання, він надсилає на сервер початкове повідомлення (Client Hello), яке містить випадкове число, набір підтримуваних алгоритмів шифрування та іншу службову інформацію. У відповідь сервер надсилає своє повідомлення (Server Hello), додаючи до нього цифровий сертифікат, що містить його публічний ключ. Цей сертифікат підписано довіреним центром сертифікації,

тому клієнт може перевірити його справжність і впевнитися в тому, що взаємодіє саме з тим сервером, з яким має.

Після валідації сертифіката клієнт генерує випадкове значення, яке називається pre-master secret. Це значення виступає заготовкою для подальшого обчислення спільного симетричного ключа. Щоб безпечно передати це значення серверу, клієнт шифрує його за допомогою публічного ключа сервера і відправляє назад. На сервері цей зашифрований pre-master secret розшифровується за допомогою приватного ключа, який зберігається тільки на стороні сервера і недоступний ззовні.

Маючи pre-master secret та випадкові числа, обидві сторони — клієнт і сервер — незалежно одна від одної генерують спільний симетричний сеансовий ключ. Саме цей ключ буде використовуватися для шифрування всього подальшого трафіку під час сесії. Таким чином, асиметричне шифрування використовується лише для безпечної ініціалізації, тобто для обміну ключем, тоді як основна передача даних відбувається вже з використанням симетричного шифрування, яке є значно швидшим і ефективнішим.

2.4 Принципи захисту даних у Java: шифрування, хешування, цілісність

Після розгляду архітектурних підходів, механізмів автентифікації та основ криптографії перейдемо до того, як ці принципи реалізуються на практиці. в одному з найпоширеніших середовищ для створення веб-додатків, яким є Java.

Для цього в Java передбачено Java Cryptography Architecture (JCA) — набір програмних інтерфейсів (API) і концепцій, який дає змогу розробникам інтегрувати сучасні механізми безпеки без потреби самостійно реалізовувати алгоритми [19].

У межах JCA передбачено підтримку цифрових підписів, хеш-функцій, симетричних і асиметричних алгоритмів шифрування, генерації та управління ключами, а також створення криптографічно стійких випадкових значень. Усі ці можливості реалізуються через гнучку модульну систему так званих провайдерів — окремих компонентів, які надають конкретні реалізації криптографічних сервісів. Ці

провайдери можна динамічно підключати до платформи Java через стандартизований інтерфейс, що дозволяє використовувати різні реалізації одного й того ж алгоритму без необхідності змінювати логіку самої програми [19].

Серед основних принципів, на яких ґрунтується JCA — незалежність реалізацій, взаємна сумісність провайдерів і можливість їхнього розширення. Це означає, що застосунок не обмежений конкретною реалізацією алгоритмів, а може працювати з будь-яким сумісним провайдером — як вбудованим, так і користувацьким. Завдяки цьому Java дає змогу легко застосовувати вже доступні стандартизовані алгоритми або, за потреби, підключати нові, наприклад ті, що розроблені в межах внутрішніх проєктів чи для підтримки нових криптографічних стандартів [19].

У Java всі криптографічні сервіси базуються на класі java.security.Provider. Кожен з них містить інформацію про власне ім’я, а також про перелік алгоритмів та сервісів, які він підтримує. Коли застосунок створює об’єкт для виконання криптографічної операції, наприклад для хешування чи шифрування, платформа звертається до зареєстрованих провайдерів. Якщо один із них містить реалізацію потрібного алгоритму, система створює відповідний об’єкт і передає його програмі [19].

Зазвичай кожен провайдер охоплює одну або кілька груп криптографічних алгоритмів — наприклад, AES, RSA чи SHA-256. Після встановлення JDK середовище вже містить низку стандартних провайдерів, зокрема SUN, SunJCE, SunRsaSign та інші. При цьому Java дозволяє розширити цей список: сторонні або власні провайдери можна додавати як статично — через файл java.security, — так і динамічно, під час виконання програми. У разі, якщо не зазначено конкретного провайдера, система здійснює пошук у порядку, визначеному конфігурацією — саме цей порядок можна змінювати, якщо виникає потреба у виборі специфічної реалізації [19].

З практичної точки зору, для використання певного алгоритму достатньо викликати відповідний метод з Java API. Наприклад, щоб створити об’єкт для обчислення хешу за алгоритмом SHA-256, застосовується команда:

MessageDigest md = MessageDigest.getInstance(«SHA-256»);

У цьому прикладі об'єкт буде створений з використанням реалізації одного з наявних провайдерів, які підтримують SHA-256 [19].

Аналогічним чином працює реалізація шифрування в Java. Універсальним інтерфейсом для цього є клас javax.crypto.Cipher, який дозволяє працювати як із симетричними, так і з асиметричними алгоритмами. Він підтримує великий набір стандартів, зокрема AES, DES, RSA, а також різні режими — CBC, GCM, ECB тощо. Наприклад, для створення об'єкта шифрування можна використати такий виклик:

Cipher.getInstance(«RSA/ECB/PKCS1Padding»);

Для реалізації симетричних алгоритмів, таких як AES Java підтримує ключі довжиною 128, 192 або 256 біт. Ключ можна згенерувати автоматично за допомогою класу KeyGenerator або задати вручну через SecretKeySpec. У режимах, що використовують ініціалізаційний вектор (IV), наприклад, додатково забезпечується захист від повторного використання одного й того ж шифрувального шаблону, що підвищує криптостійкість [19].

Для реалізації асиметричного шифрування в Java передбачено використання стандартних інструментів, таких як класи KeyPairGenerator для створення пари відкритого та закритого ключів і Cipher для виконання операцій шифрування та дешифрування. Після ініціалізації відповідного алгоритму (наприклад, RSA), ці компоненти дозволяють забезпечити конфіденційність переданих даних або виконати перевірку автентичності. Реалізація підтримує різні режими роботи та формати заповнення, що дозволяє адаптувати алгоритм до конкретних потреб системи [19].

Завдяки єдиному інтерфейсу Cipher, Java дозволяє застосовувати однаковий програмний підхід як для симетричного, так і для асиметричного шифрування. Це спрощує реалізацію багаторівневих схем захисту даних, у яких, наприклад, може комбінуватись шифрування вмісту за допомогою AES і захист ключів за допомогою RSA.

Для забезпечення цілісності даних, а також для створення цифрових відбитків повідомлень, Java надає стандартний механізм хешування через клас java.security.MessageDigest. Цей клас дозволяє обчислювати хеші за допомогою алгоритмів SHA-1, SHA-256, SHA-512 та інших. Найчастіше використовується SHA-256 — криптографічно стійкий алгоритм, який формує 256-бітний хеш незалежно від розміру вхідних даних [20].

Створення об'єкта хешування здійснюється через MessageDigest.getInstance(«SHA-256»). У подальшому дані, що підлягають перевірці, передаються до об'єкта через метод update(), після чого результат обчислюється методом digest(). Отриманий хеш може бути збережений разом з повідомленням або використаний для верифікації його автентичності при передачі [20].

У випадках, коли необхідно перевірити не лише цілісність даних, а й їхнє джерело, доцільно використовувати HMAC (Hash-based Message Authentication Code). Це криптографічна конструкція, яка поєднує хеш-функцію з секретним ключем, забезпечуючи захист від підміни повідомлення. Для реалізації HMAC в Java використовується клас javax.crypto.Mac, що підтримує алгоритми типу HmacSHA256 або HmacSHA512. Такий підхід дозволяє забезпечити перевірку автентичності та цілісності повідомлень за допомогою спільного симетричного ключа, без необхідності використовувати повноцінну інфраструктуру відкритих ключів [20].

Для збереження паролів у Java рекомендовано використовувати адаптивні хеш-функції, наприклад BCrypt. Вони додають до пароля випадкове значення, щоб кожне хеш-представлення було унікальним, а також дозволяють налаштувати складність обчислень. Це ускладнює масовий підбір паролів, навіть якщо зловмисник має заздалегідь підготовлену базу хешів. У Java підтримка BCrypt реалізована через бібліотеку Spring Security (BCryptPasswordEncoder) [21].

3 ПРОЄКТУВАННЯ І РЕАЛІЗАЦІЯ БЕЗПЕЧНОГО ВЕБ-ДОДАТКА НА JAVA

3.1 Постановка задачі, технічні та безпекові вимоги

На основі теоретичного аналізу архітектур веб-додатків, механізмів автентифікації та сучасних криптографічних підходів, у межах цієї роботи необхідно розробити захищений веб-застосунок, що виконує функцію загального чату. Основна мета полягає в реалізації прикладної системи, яка не лише забезпечує обмін повідомленнями, а й демонструє інтеграцію ключових принципів безпечної розробки: контроль доступу, шифрування даних, автентифікацію користувачів, захист переданої та збереженої інформації, а також аудит подій, пов’язаних із безпекою.

Застосунок має бути реалізований у вигляді RESTful API з підтримкою шифрування трафіку за допомогою протоколу HTTPS. Для автентифікації користувачів необхідно використовувати JWT, з цифровим підписом, терміном дії та перевіркою дійсності при кожному запиті. Доступ до функціоналу повинен бути чітко розмежований на основі ролей, зокрема USER та ADMIN.

Функціональність користувачів із роллю USER має бути обмежена можливістю надсилати текстові повідомлення в загальний чат і отримувати повну історію повідомлень. Водночас користувачі з роллю ADMIN повинні мати змогу виконувати адміністративні дії, а саме: переглядати список усіх зареєстрованих користувачів, блокувати або видаляти користувачів, переглядати журнал безпекових подій, а також видаляти повідомлення, що порушують правила взаємодії. Усі адміністративні дії мають бути зафіксовані в журналі подій.

Вміст повідомлень, що надсилаються користувачами, повинен бути зашифрований перед збереженням у базі даних. Шифрування має відбуватися на прикладному рівні з використанням гібридного підходу: повідомлення шифрується симетричним алгоритмом AES, а ключ шифрується за допомогою асиметричного алгоритму RSA. Приватний ключ для дешифрування має зберігатися виключно на сервері.

У результаті реалізації поставленої задачі має бути створено веб-застосунок, який задовольняє наступні вимоги:

- підтримка реєстрації, автентифікації та авторизації користувачів;
- наявність двох типів ролей з різними правами доступу;
- обмін повідомленнями через REST API;
- збереження зашифрованих повідомлень у базі даних;
- використання JWT з цифровим підписом для автентифікації;
- застосування HTTPS для шифрування трафіку;
- ведення журналу подій безпеки;
- реалізація інтерфейсів для керування користувачами й повідомленнями з боку адміністратора.

3.2 Архітектура веб-додатка

Розроблений веб-застосунок побудований за класичною багаторівневою архітектурою типу MVC (Model–View–Controller) із використанням фреймворку Spring Boot. У структурі системи виділено окремі рівні: представлення (контролери), бізнес-логіка (сервіси) та доступ до даних (репозиторії). Оскільки застосунок реалізовано у вигляді RESTful API, рівень «View» фактично представлений у вигляді HTTP-відповідей у форматі JSON. Уся система реалізована як монолітний веб-додаток, тобто всі її компоненти працюють у межах єдиного застосунку й розгортаються спільно.

Усі зовнішні запити до сервера надходять через контролери, які обробляють HTTP-методи, виконують валідацію вхідних даних і делегують виконання бізнес-логіки відповідним сервісам. Сервіси, у свою чергу, взаємодіють із базою даних через репозиторії, реалізовані за допомогою Spring Data JPA.

Архітектура передбачає чітке розділення відповідальностей:

1) Контролери (AuthController, UserController, ChatController, AdminController) обробляють вхідні HTTP-запити, виконують базову перевірку даних, викликають відповідні сервіси й формують відповіді у форматі JSON;

2) Сервіси (LoginAttemptService, TokenBlacklistService) реалізують логіку безпеки, зокрема обмеження кількості спроб входу та перевірку JWT-токенів на відкликання;

3) Допоміжні сервіси (EncryptionService, AuditLogService, ServerKeyService) відповідають за шифрування і дешифрування повідомлень, запис подій у журнал безпеки та управління криптографічними ключами;

4) Репозиторії (UserRepository, ChatMessageRepository, AuditLogRepository) забезпечують доступ до бази даних і оперують сутностями користувачів, повідомлень та логів дій.

Уся система побудована як безстанна: після успішної автентифікації користувач отримує підписаний JWT-токен, який передається в заголовку Authorization при кожному подальшому запиті. На сервері не зберігається інформація про активну сесію користувача.

Для наочності взаємодії між контролерами, сервісами, репозиторіями та базою даних, нижче подано узагальнену архітектурну схему системи (рис. 3.1).

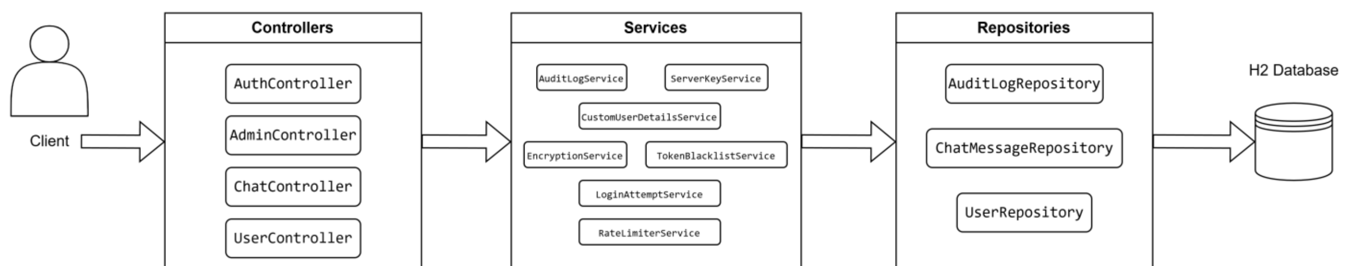

Рисунок 3.1 – Логічна схема застосунку

Щоб краще зрозуміти внутрішню структуру застосунку на рівні класів і взаємозв’язків між ними, нижче наведено UML-діаграму класів (рис. 3.2), згенеровану на основі реалізованих компонентів системи.

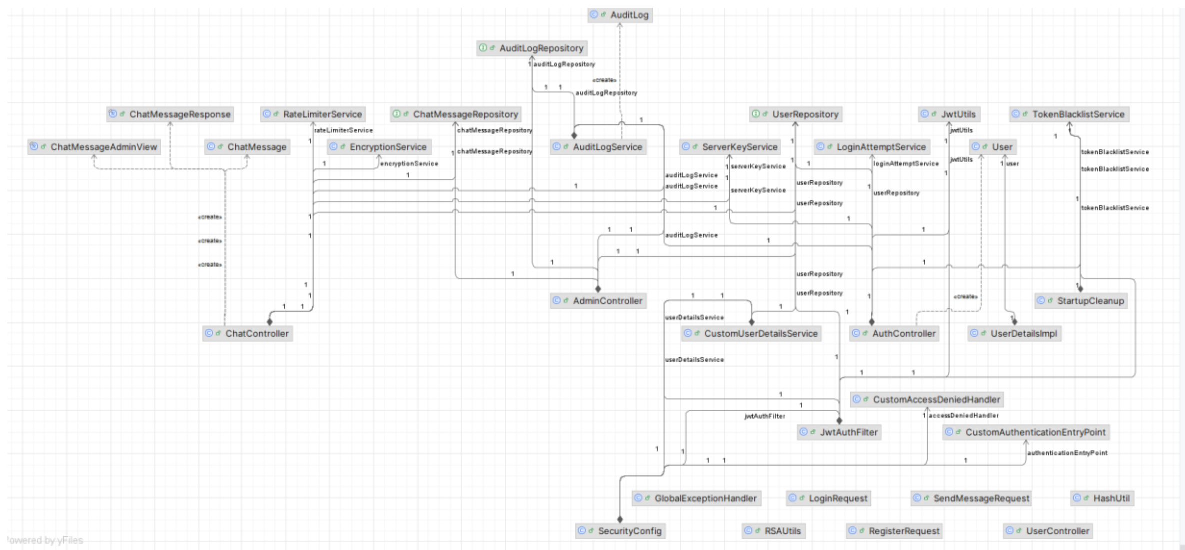

Рисунок 3.2 – UML діаграма реалізованих класів

UML-діаграма на рисунку 3.2 ілюструє взаємозв’язки між ключовими компонентами системи. Вона демонструє, що кожен контролер взаємодіє лише з тими сервісами, які безпосередньо реалізують його функціональність. Наприклад, ChatController використовує EncryptionService для шифрування повідомлень і RateLimiterService для контролю частоти запитів. AdminController звертається до AuditLogService, UserRepository та інших компонентів для виконання адміністративних операцій.

3.3 Реалізація автентифікації користувачів

Процес автентифікації у реалізованому веб-застосунку починається з реєстрації нового користувача. Клієнт надсилає POST-запит на відповідний ендпоінт контролера AuthController, передаючи логін, пароль, та, для вибору ролі адміністратора, спеціальний пароль. Система перевіряє, чи не існує вже користувача з таким логіном. Якщо користувач ще не зареєстрований і роль — USER, обліковий запис створюється одразу. Якщо ж вказано роль ADMIN, перевіряється відповідність наданого адміністративного пароля збереженому значенню в конфігурації застосунку (application.properties). У разі успішної перевірки система створює обліковий запис із роллю адміністратора. Приклад реєстрації користувача наведений у Додатку Б.

Після реєстрації користувач надсилає запит на авторизацію, передаючи логін і пароль. Контролер AuthController перевіряє облікові дані за допомогою CustomUserDetailsService. Якщо автентифікація успішна, система генерує підписаний JWT-токен за допомогою JwtUtils і повертає його клієнту у вигляді текстового рядка. Приклад успішної авторизації наведений у Додатку Б.

Подальші запити до захищених ресурсів повинні містити токен у заголовку Authorization. Усі такі запити перехоплюються JwtAuthFilter (Додаток А), який перевіряє валідність і термін дії токена, а також переконується, що токен не перебуває у чорному списку (TokenBlacklistService). Після успішної перевірки користувач вважається автентифікованим, а доступ до ресурсів надається згідно з його роллю.

У разі виходу користувача з системи (logout), токен, який було йому видано, додається до чорного списку. Результат виходу користувача із системи наведений у Додатку Б. Це гарантує, що навіть якщо термін дії токена ще не завершився, його не можна буде використати повторно для доступу до захищених ресурсів.

Крім того, під час запуску застосунку виконується автоматичне очищення чорного списку токенів за допомогою компонента StartupCleanup. Завдяки цьому жоден токен, створений до перезапуску сервера, не буде вважатися чинним, що підвищує загальний рівень безпеки системи. Спроба зайти за старим токеном наведена у Додатку Б.

3.4 Схема шифрування повідомлень та розподілу ключів

Процес шифрування реалізовано наступним чином: на стороні клієнта для кожного повідомлення генерується новий випадковий AES-ключ, яким шифрується текст повідомлення. Після цього сам AES-ключ шифрується за допомогою публічного RSA-ключа сервера, що гарантує, що лише сервер, який володіє відповідним приватним ключем, зможе його розшифрувати. Клієнт отримує публічний ключ від сервера під час реєстрації. Згенеровані приватний та публічний ключ, а також отримання адміністратором публічного ключа при реєстрації наведено у Додатку Б.

Обидва зашифровані елементи: повідомлення та AES-ключ надсилаються на сервер. Повідомлення зберігається у базі у вигляді об’єкта ChatMessage, що містить поля encryptedText та encryptedKey. Приклад збереженого зашифрованого повідомлення наведено у Додатку Б.

Під час отримання повідомлень сервер спочатку розшифровує AES-ключ за допомогою свого приватного RSA-ключа, а потім використовує цей ключ для дешифрування самого тексту повідомлення. Цей процес реалізовано у класі EncryptionService, який використовує утилітний клас RSAUtils для роботи з ключами. Збереження та доступ до пари ключів на сервері забезпечує ServerKeyService.

Перевага такого підходу полягає в тому, що навіть у випадку компрометації бази даних сторонні особи не зможуть прочитати вміст повідомлень без доступу до приватного RSA-ключа, що зберігається лише на сервері.

Окрім прикладного шифрування повідомлень, для захисту мережевого трафіку було додатково згенеровано самопідписаний SSL-сертифікат. Завдяки цьому весь обмін даними між клієнтом і сервером відбувається через захищене з'єднання по HTTPS, що виключає можливість перехоплення повідомлень під час передачі. Команда, яка була виконана для генерації самопідписаного сертифікату та результат її виконання наведено у Додатку Б.

Після створення сертифіката необхідно налаштувати сервер для використання HTTPS. Всі необхідні параметри конфігурації, зокрема шлях до keystore, пароль доступу та порт, на якому працює застосунок, задаються у файлі application.properties, вміст якого наведено у Додатку Б.

3.5 Проєктування перевірки цілісності повідомлень

У системах обміну повідомленнями важливо не лише шифрувати вміст, а й бути впевненим у тому, що цей вміст не був змінений або пошкоджений після збереження. Саме тому в розробленому веб-застосунку реалізовано механізм перевірки цілісності кожного повідомлення.

Після того, як користувач надсилає повідомлення, система не просто зберігає його у зашифрованому вигляді, а ще й формує — хеш, який створюється за допомогою SHA-256 на основі трьох складових: імені відправника, самого тексту повідомлення (до шифрування) та форматованої мітки часу. В результаті кожне повідомлення має унікальний хеш, який записується в базу разом із іншими полями. Приклад відправлення повідомлення наведено у Додатку Б.

Коли інший користувач (або той самий) отримує історію повідомлень, сервер автоматично перевіряє цілісність кожного з них. Для цього він заново обчислює хеш із тексту повідомлення, імені відправника та часу надсилання, після чого порівнює отримане значення з тим, що зберігається в базі. Приклад зберігання хешу повідомлення у БД наведено у Додатку Б.

Якщо обидва хеші збігаються — повідомлення вважається автентичним і передається користувачу. Результат отримання повідомлення наведено у Додатку Б.

Якщо ж виявляється розбіжність — це ознака того, що повідомлення було змінено, і воно не потрапляє в результат. У таких випадках система фіксує попередження у журналі подій.

3.6 Збереження облікових записів і хешування паролів

У реалізованому веб-застосунку облікові записи користувачів зберігаються в базі даних у вигляді об’єктів сутності User. Під час реєстрації користувача введений пароль не зберігається у відкритому вигляді, а одразу шифрується за допомогою алгоритму хешування BCrypt. Для цього використовується компонент BCryptPasswordEncoder, який створює криптографічно стійкий хеш, що включає унікальну «сіль» та параметр складності. Згенерований хеш зберігається у полі password у базі даних. Приклад збереження користувача у БД із зашифрований паролем наведено у Додатку Б.

Під час подальшої авторизації пароль, що надходить від клієнта, не порівнюється напряму, а передається до методу перевірки, який порівнює його хеш із

уже збереженим у базі. Якщо хеші збігаються, користувача автентифіковано успішно. Уся перевірка здійснюється без необхідності знати або зберігати сам пароль, що виключає ризик компрометації у випадку витоку даних.

3.7 Логування подій і аудит безпеки

Для аудиту безпеки використовується спеціальний сервіс AuditLogService, який відповідає за створення та збереження записів про події у базі даних. Усі важливі дії, такі як блокування або видалення користувачів, видалення повідомлень з чату, успішні входи з адміністративною роллю — фіксуються автоматично. Відповідні методи викликають createLog(...), що створює об'єкт AuditLog, у якому зберігається тип події, ім'я користувача, час виконання та супровідний опис.

Збереження цих записів здійснюється через AuditLogRepository, який оперує таблицею журналу дій у базі, приклад якої наведено у Додатку Б. Ці дані доступні лише адміністраторам через окремий ендпоінт AdminController, що дозволяє переглядати список подій у системі. Таким чином, адміністратор має змогу контролювати дії користувачів і швидко реагувати на потенційно шкідливу активність.

Крім того, загальне логування у застосунку реалізовано за допомогою бібліотеки log4j. Вона використовується для виводу системних повідомлень у консоль або у файл логів, зокрема для фіксації запуску програми, очищення чорного списку токенів (StartupCleanup), помилок доступу та винятків. Загальне логування продемонстровано у Додатку Б.

3.8 Засоби адміністративного контролю та безпеки

Адміністративний контроль у веб-застосунку реалізовано як окремий функціональний блок, доступ до якого мають лише користувачі з роллю ADMIN. Після автентифікації з відповідними правами адміністратор отримує доступ до

спеціального ендпоінта AdminController, який надає можливості для керування системою й користувачами.

Одна з ключових функцій адміністратора — це блокування користувачів. Заблокувати користувача можна як повністю, так і тимчасово. У випадку тимчасового блокування в обліковому записі зберігається дата завершення блокування, і під час автентифікації система перевіряє, чи минув цей час. Якщо ні — користувач не допускається до системи, навіть із валідним токеном. Повне блокування здійснюється шляхом встановлення спеціального прапорця, який повністю забороняє доступ до будь-яких функцій системи. Приклад блокування користувача в системі та спроба повторного входу надано у Додатку Б.

Крім ручного блокування, у системі реалізовано автоматичне тимчасове блокування користувача при несанкціонованих спробах входу. Якщо користувач декілька разів поспіль вводить неправильні облікові дані, LoginAttemptService блокує доступ на певний період, запобігаючи brute-force-атакам. Інформація про кількість невдалих спроб та час блокування зберігається й обробляється автоматично. Приклад блокування після 5-ти невдалих спроб вводу пароля наведено у Додатку Б.

Також адміністратор має змогу видаляти користувачів із системи. Видалення облікового запису призводить до повного очищення відповідних записів у базі. Приклад видалення користувача із системи наведено у Додатку Б.

Крім того, передбачена можливість видалення окремих повідомлень у чаті, наприклад, якщо вони порушують правила взаємодії. Приклад видалення повідомлення наведено у Додатку Б.

Усі ці дії — блокування, розблокування, видалення користувачів чи повідомлень — фіксуються в журналі подій через AuditLogService, що гарантує повну прозорість адміністративних втручань.

4 ТЕСТУВАННЯ І ОЦІНКА БЕЗПЕКИ РОЗРОБЛЕНОГО ВЕБ-ДОДАТКА

4.1 Методика тестування веб-додатку

Щоб переконатися в правильності реалізації, стійкості до загроз і відповідності сучасним вимогам безпеки, для створеного веб-додатку було проведено комплексне тестування, яке включає як юніт-тести, так і інтеграційні тести. Тестування охоплює ключові компоненти системи — від конфігурації та контролерів до сервісів шифрування, перевірки прав доступу та логування подій.

Юніт-тести використовувалися для перевірки окремих частин бізнес-логіки та допоміжних класів: наприклад, функцій генерації JWT, дешифрування повідомлень або ведення журналу дій. Вони дозволяють ізольовано перевірити, чи кожен компонент поводиться очікувано при різних вхідних даних.

Інтеграційні тести, у свою чергу, перевіряють взаємодію між компонентами системи — наприклад, автентифікацію через JWT, доступ до захищених ендпоінтів, реакцію на неправильні запити, фіксацію адміністративних дій або обробку повідомлень із шифруванням і перевіркою хешу.

Усі тести згруповані відповідно до архітектурної структури проєкту — за пакетами: config, controller, exception, security, service. У кожному пакеті реалізовано один або кілька тестових класів, кожен із яких містить набір методів, що перевіряють ключові функціональні й безпекові сценарії.

Детально розглянемо всі реалізовані тестові класи:

Пакет config містить конфігураційні компоненти, що визначають правила безпеки, механізми фільтрації запитів та поведінку застосунку під час запуску. Тестування в цьому пакеті спрямоване на перевірку коректності створення необхідних бінів, а також виконання критичних дій, які впливають на безпеку системи ще до початку її повноцінної роботи.

SecurityConfigTest: Цей тестовий клас перевіряє, що всі необхідні для функціонування системи безпеки Spring Beans коректно ініціалізуються і доступні в контексті.

- contextLoadsAndBeansAreNotNull() — перевіряє, що основні компоненти, створюються без помилок і не є null.

StartupCleanupTest: Під час запуску застосунку виконується ініціалізація компонента StartupCleanup, завданням якого є очищення чорного списку токенів, створеного під час попереднього сеансу роботи.

- testClearBlacklistOnStartupIsCalled() — перевіряє, що при старті застосунку дійсно викликається метод clearBlacklist() із сервісу TokenBlacklistService. Для цього використовується мок-об’єкт, який перевіряє факт виклику.

Результат успішного виконання тестів пакету config наведено у Додатку В.

Пакет controller: контролери є ключовими компонентами веб-додатку, адже саме вони приймають HTTP-запити, виконують валідацію, ініціюють бізнес-логіку через сервіси, перевіряють права доступу та формують відповіді. Для кожного контролера створено окремий інтеграційний тестовий клас, що охоплює як успішні сценарії, так і перевірки захисту, валідації, та обробки помилок.

AuthControllerTest: призначений для перевірки процесу реєстрації, автентифікації та виходу з системи, охоплюючи основні сценарії взаємодії користувача з механізмами безпеки:

- testRegisterUserSuccessfully() — перевіряє успішну реєстрацію звичайного користувача через /api/auth/register-user. Очікується код 200 та повідомлення про успішну реєстрацію.
- testRegisterAdminSuccessfully() — перевіряє створення адміністратора при вказанні правильного секретного коду.
- testRegisterDuplicateUser() — імітує повторну реєстрацію користувача з існуючим ім’ям. Очікується код 400 та відповідне повідомлення.

- testRegisterAdminWithInvalidSecret() — перевіряє спробу реєстрації адміністратора з некоректним секретом. Очікується статус 403.
- testRegisterWithEmptyUsernameOrPassword() — перевіряє валідацію реєстраційного запиту з порожніми полями. Очікується код 400 Bad Request.
- testLoginWithValidCredentials() — перевірка, що користувач з валідними обліковими даними отримує JWT токен у відповіді.
- testLoginWithInvalidPassword() — логін із неправильним паролем, очікується статус 401 та повідомлення про невірні облікові дані.
- testLoginWithNonexistentUser() — логін з неіснуючим іменем користувача. Очікується 401 Unauthorized та стандартне повідомлення про помилку автентифікації.

UserControllerTest: призначений для перевірки доступу до профілю автентифікованого користувача:

- testGetProfileWithValidToken() — перевірка, що користувач з валідним JWT може отримати свій профіль через /api/user/profile.
- testGetProfileWithoutToken() — запит без токена призводить до 401 Unauthorized.
- testGetProfileWithInvalidToken() — надання невалідного або підробленого токена також блокується.

ChatControllerTest: Цей контролер реалізує логіку обміну повідомленнями, включаючи шифрування, перевірку цілісності та обмеження частоти запитів:

- testSendMessageSuccessfully() — перевіряє успішне надсилання повідомлення з валідним токеном та дозволеним доступом згідно з RateLimiter. Очікується код 200 та повідомлення про успішну відправку.
- testSendMessageTooFast() — моделює ситуацію, коли користувач надсилає повідомлення надто часто. Другий запит повертає статус 429 (Too Many Requests).

- testGetMessagesWithValidToken() — тестує успішне отримання зашифрованих повідомлень за наявності дійсного токена. Очікується статус 200.
- testGetMessagesWithoutToken() — перевіряє, що доступ до повідомлень без токена блокується. Очікується статус 401 Unauthorized.
- testSendMessageWithEmptyBody() — перевірка валідації порожнього тіла запиту при надсиланні повідомлення. Очікується статус 400 Bad Request.

AdminControllerTest: Цей контролер доступний лише адміністраторам і реалізує повний спектр функцій управління користувачами, повідомленнями та логами:

- testGetAllUsersAsAdmin() — перевірка, що адміністратор успішно отримує список усіх користувачів.
- testGetSpecificUser() — перевірка можливості отримання детальної інформації про конкретного користувача.
- testDeleteUser() — перевірка видалення користувача з системи.
- testBlockAndUnblockUser() — послідовне блокування та розблокування користувача через відповідні ендпоінти.
- testTemporaryBlockUser() — перевірка тимчасового блокування користувача (без параметрів часу, використовується дефолтна тривалість).
- testDeleteMessage() — перевірка успішного видалення повідомлення з чату.
- testAuditLogRecorded() — підтвердження, що після адміністративної дії (наприклад, блокування) відповідна подія фіксується в журналі аудиту (через ендпоінт /api/admin/logs).

Результат успішного виконання тестів пакету controller наведено у Додатку В.

Пакет security: цей пакет реалізує основні механізми захисту: генерацію та перевірку JWT, фільтрацію запитів, обмеження за кількістю спроб входу та перевірку цілісності даних через хешування. Тестові класи охоплюють усі критичні компоненти безпеки.

JwtUtilsTest: Тестує генерацію та перевірку JWT-токенів:

- testGenerateAndValidateToken() — перевіряє повний цикл створення токена, його дійсність і правильне вилучення імені користувача.
- testTokenValidationAgainstUserDetails() — перевіряє, що токен визнається дійсним лише для користувача, для якого його було створено.
- testTokenValidationFailsWithWrongUser() — переконується, що токен недійсний, якщо його використовує інший користувач.
- testInvalidToken() — імітує повністю некоректний токен і перевіряє, що валідація не проходить і при спробі отримання імені користувача виникає виняток.

JwtAuthFilterTest: Фільтр JWT відповідає за обробку вхідних запитів та визначення, чи варто допускати користувача до ресурсу. Перевірено:

- shouldSkipFilterForAuthEndpoint() — фільтр не застосовується до маршрутів /api/auth/**.
- shouldRejectBlacklistedToken() — токени, які було відкликано та додано до чорного списку, не пропускаються до обробки, повертається статус 401.
- shouldRejectIfUserIsBlocked() — користувач, який має валідний токен, але перебуває у статусі блокування, не проходить перевірку, доступ заборонено.
- shouldAuthenticateValidToken() — для валідного токена встановлюється об'єкт аутентифікації в SecurityContext, і запит проходить далі.
- shouldReturnUnauthorizedOnInvalidToken() — при обробці підробленого або пошкодженого токена система повертає 401 з відповідним повідомленням.

LoginAttemptServiceTest: Цей клас реалізує захист від brute-force атак:

- testUserNotBlockedInitially() — новий користувач не заблокований за замовчуванням.
- testBlockingAfterFailedAttempts() — після п’яти невдалих входів користувача блокує.

- testUnblockAfterSuccess() — успішна авторизація скидає статус блокування.
- testSeparateUsers() — блокування одного користувача не впливає на інших.

HashUtilTest: Відповідає за перевірку коректності хешування повідомлень методом SHA-256:

- testHashIsConsistent() — для одного й того самого рядка повертається однаковий хеш.
- testHashChangesWhenInputChanges() — якщо змінені вхідні дані, то змінюється і хеш.
- testHashIsNotNullOrEmpty() — хеш не може бути null або порожнім, і має точну довжину 64 символи (hex).

Результат успішного виконання тестів пакету security наведено у Додатку В.

Пакет service: сервісний рівень відповідає за бізнес-логіку та критичні функції: шифрування повідомлень, логування дій користувачів, обробку RSA-ключів, обмеження частоти запитів та управління чорним списком токенів. Усі тести є юніт-тестами й перевіряють поведінку окремих сервісів в ізоляції, використовуючи моки, або перевіряючи внутрішню логіку без залежності від контексту Spring.

EncryptionServiceTest: Тестує симетричне шифрування повідомлень (AES) та їх розшифрування:

- testEncryptAndDecryptSuccess() — перевіряє, що після шифрування і подальшого розшифрування повідомлення повертається в первісному вигляді.
- testDecryptWithWrongKeyFailure() — спроба розшифрувати повідомлення іншим ключем призводить до винятку.
- testGenerateAESKeyUniqueKeys() — перевіряє, що метод generateAESKey() щоразу повертає унікальний ключ.

AuditLogServiceTest: Сервіс логування дій, що взаємодіє з AuditLogRepository. Тест реалізовано з використанням Mockito.

- testLogActionSavesCorrectLog() — перевірка, що під час логування зберігається правильний об’єкт із вказаними користувачем, дією та часовою міткою.

ServerKeyServiceTest: Тестує генерацію та доступ до RSA-ключів, які використовуються для шифрування AES-ключів:

- testPrivateKeyNotNull() — перевірка, що приватний ключ згенеровано.
- testPublicKeyBase64NotEmpty() — публічний ключ доступний у форматі Base64.
- testPublicKeyIsValid() — перевіряє, що публічний ключ можна коректно відновити з Base64-рядка через KeyFactory.

RateLimiterServiceTest: Забезпечує захист від надмірної кількості запитів (антиспам):

- testIsAllowedInitially() — перші кілька запитів дозволяються.
- testIsBlockedAfterThreshold() — після перевищення ліміту користувача блокує.
- testDifferentUsersIndependentLimits() — перевірка, що обмеження діють незалежно для кожного користувача.
- TokenBlacklistServiceTest: Тестує сервіс, який керує чорним списком токенів:
- testTokenNotBlacklistedInitially() — новий токен за замовчуванням дозволений.
- testTokenIsBlacklistedAfterAdding() — токен потрапляє до blacklist після додавання.
- testMultipleTokensHandledSeparately() — переконання, що blacklist обробляє токени незалежно.
- testClearBlacklist() — метод clearBlacklist() очищає весь список токенів.

Результат успішного виконання тестів пакету service надано у Додатку В.

Пакет exception: Пакет exception містить глобальний механізм обробки помилок, що реалізований через клас GlobalExceptionHandler. Він дозволяє централізовано реагувати на різні типи винятків, що виникають під час обробки запитів, та повертати зрозумілі повідомлення користувачеві.

GlobalExceptionHandlerTest: Тест перевіряє реакцію системи на ключові винятки, які можуть виникати під час взаємодії користувача з додатком:

- testValidationErrorHandled() — симуляція валідаційної помилки (реєстрація з порожнім логіном і паролем). Очікується код 400 та відповідне повідомлення.
- testBadCredentialsHandled() — логін із неправильним ім'ям користувача або паролем. Очікується статус 401 і повідомлення "Invalid username or password."
- testAccessDeniedHandled() — авторизований користувач із роллю USER намагається отримати доступ до адміністративного ендпоінта /api/admin/users. Очікується 403 та повідомлення про заборону доступу.
- testUnexpectedExceptionHandled() — викликається штучний ендпоінт /api/trigger-error, який імітує RuntimeException. Очікується помилка 500 з повідомленням «Unexpected server error.»

Результат успішного виконання тестів пакету exception наведено у Додатку В.

4.2 Перевірка стійкості системи до атак

Щоб переконатися, що розроблений веб-додаток захищений не лише на рівні окремих функцій, а й у цілому — як система, було проведено повноцінне тестування на стійкість до поширених загроз. Для цього було використано класифікацію вразливостей, яку було детально описано у розділі 1.2 цієї дипломної роботи. Цей підхід передбачає перевірку безпеки на кожному етапі життєвого циклу створення програмного забезпечення: від аналізу вимог і проєктування до реалізації та впровадження.

1) Порушення контролю доступу.

У багатьох веб-додатках ця вразливість виникає тоді, коли система не перевіряє, до яких саме маршрутів звертається користувач, або виконує це поверхнево. В результаті користувач із базовими правами може отримати доступ до функцій адміністратора — наприклад, переглядати або змінювати інших користувачів, видаляти повідомлення чи бачити лог дій.

У моєму додатку контроль доступу реалізовано через механізм ролей (USER та ADMIN), і перевірка здійснюється на рівні ендпоінтів за допомогою Spring Security. Зокрема, усі маршрути /api/admin/** відкриті виключно для ролі ADMIN, що налаштовується у класі конфігурації безпеки.

Щоб перевірити стійкість системи до цієї вразливості, я авторизувався як звичайний користувач (роль USER) і спробував звернутись до ендпоінта GET /api/admin/users (Додаток В).

У тілі запиту використовувався валідний JWT токен, який належить не адміну, а простому користувачу. У результаті система повернула статус 403 Forbidden — тобто доступ був успішно заблокований. Це підтверджує, що роль користувача дійсно перевіряється, і навіть з валідним токеном він не має доступу до захищеної частини системи.

2) Зловживання функціональністю.

При даній вразливості зловмисник не намагається обійти систему напряму, натомість він користується тим, що вже дозволено, — але у спосіб, якого не передбачено логікою додатку. Наприклад, можна багато разів викликати одну й ту саму дію, навіть якщо вона вже була виконана.

У моєму додатку я перевірив, чи можливо таким чином «зламати» або дестабілізувати поведінку системи. Зокрема, я увійшов як адміністратор, надіслав запит DELETE /api/admin/users/{id} для видалення користувача, а потім повторив цей запит ще раз — уже до користувача, якого фактично не існує (Додаток В). У відповідь система повернула коректне повідомлення з поясненням, що такого користувача

більше немає. Тобто — видалити вже видалене неможливо, і це не викликає ніякої помилки.

Аналогічно було протестовано POST /api/auth/logout: навіть якщо користувач натисне кнопку «вийти» кілька разів поспіль — додаток все одно поводиться адекватно (Додаток В). Токен, який вже був відкликаний, більше не працює, і повторні запити просто ігноруються або повертають статус 401.

3) Неналежна обробка помилок.

При вразливості неналежної обробки помилок система у відповідь на некоректний запит може розкрити технічні деталі, які користувач не має бачити: назви класів, stack trace, внутрішню структуру БД тощо.

Щоб переконатися, що в моєму додатку таких проблем немає, я навмисно створив кілька ситуацій, які викликають помилки. Зокрема, намагався:

- надіслати запит на реєстрацію з порожніми полями (username і password) (Додаток В);
- авторизуватись із неіснуючим користувачем або з неправильним паролем (Додаток В);
- звернутись до захищеного ресурсу без токена (Додаток В).

У всіх випадках система відповідала лаконічними, узагальненими повідомленнями, наприклад: «Username is required», «Invalid username or password», та статусом 400 Bad Request або 401 Unauthorized. Жодної технічної інформації, стеків або імен внутрішніх класів не поверталось.

4) Атака методом перебору.

У розробленому додатку реалізовано захист від надмірної кількості невдалих спроб входу в систему. Для цього використовується сервіс LoginAttemptService, який після п’яти поспіль неправильних спроб авторизації тимчасово блокує можливість повторного входу для конкретного користувача. Тривалість блокування становить 1 хвилину, протягом якої будь-які спроби входу завершуються відповіддю зі статусом

429 Too Many Requests. Результат тестування захисту від атаки методом перебору наведено у Додатку В.

Для перевірки цього механізму я кілька разів поспіль вводив неправильний пароль. Система повертала 401 Unauthorized до п’ятої спроби включно. На шосту спробу вона вже відповіла статусом 429 із повідомленням про перевищення ліміту. Це свідчить про те, що захист від перебору працює коректно й блокує несанкціоновані дії ще до етапу видачі токена.

5) CSRF-атака.

Дана атака полягає в тому, що зловмисник змушує автентифікованого користувача виконати дію, якої він сам не ініціював — наприклад, надіслати запит на зміну пароля або видалення даних. Це можливо, коли додаток використовує cookies для зберігання сесії і не перевіряє джерело запиту, дозволяючи браузеру автоматично підставити облікові дані в будь-який запит.

У моєму додатку така вразливість архітектурно виключена завдяки використанню токенів JWT, які не зберігаються у cookies, а передаються у заголовку Authorization. Це означає, що жоден запит, ініційований з іншого сайту або сторінки, не може автоматично містити токен. Без дійсного JWT доступ до захищених маршрутів неможливий, оскільки фільтр JwtAuthFilter блокує всі запити без токена або з токеном, якого немає в системі, або якщо він відкликаний. Перевірку захисту від CSRF-атаки наведено у Додатку В.

Для підтвердження цієї поведінки я надіслав запит POST /api/chat/send без заголовка Authorization. Система повернула статус 401 Unauthorized, вказавши, що для доступу необхідно пройти автентифікацію. Це підтверджує, що автоматично відправити запит без токена неможливо, а значить — додаток захищений від CSRF-атак за своєю структурою.

6) Витік інформації.

Для перевірки стійкості додатку до витоку інформації було проаналізовано вміст відповідей на запити до API. Зокрема, при виконанні запиту GET

/api/admin/users, доступного лише для користувачів з роллю ADMIN, система повертає лише необхідні поля: id, username, role, blocked, blockExpiresAt. Поле password не включається у відповідь, навіть попри те, що воно зберігається у моделі користувача. Це запобігає передачі чутливої інформації навіть адміністраторам. Приклад інформації про користувачів, яка доступна адміну надано у Додатку В.

Також було протестовано відповідь на запит GET /api/user/profile, який звичайний користувач може виконати лише щодо власного облікового запису. У цьому випадку повертаються лише обмежені дані: username, role та blocked. Отже, користувач не має доступу до інформації про інших користувачів і не може бачити жодних внутрішніх або службових полів. Інформація, яку користувач може отримати про себе наведено у Додатку В.

7) Недостатня автентифікація.

Дана вразливість виникає тоді, коли додаток дозволяє доступ до захищених ресурсів без повної перевірки користувача — наприклад, у випадку відсутності токена або якщо він підроблений. У моєму додатку автентифікація реалізована через JWT, і кожен запит до захищених маршрутів обробляється фільтром JwtAuthFilter. Цей фільтр перевіряє, чи є токен у заголовку, чи він не відкликаний, і чи дійсний підпис.

Для перевірки я виконав лише два ключові запити:

- запит до /api/user/profile взагалі без токена (Додаток В);
- запит до цього ж маршруту з навмисно зміненим JWT (Додаток В).

У першому випадку система повернула 401 Unauthorized із повідомленням про відсутність токена. У другому — також 401, але з іншим повідомленням про недійсний токен. Це чітко демонструє, що додаток не допускає жодної взаємодії із захищеними ресурсами без повної, успішної автентифікації.

8) Переповнення буфера.

Це класична вразливість, яка найчастіше характерна для низькорівневих мов програмування, таких як C або C++, де пам’ять обробляється вручну. У високорівневих мовах, таких як Java, ця проблема практично не зустрічається завдяки

вбудованому контролю меж масивів і автоматичному керуванню пам'яттю. Проте вразливість може проявлятись у вигляді зависань або падінь системи, якщо не передбачити обмежень на обсяг введених даних.

Для перевірки, наскільки система стійка до надмірного навантаження через великі обсяги введення, я виконав запит на авторизацію (POST /api/auth/login), вказавши у полі password дуже довгий рядок — понад 10 000 символів (Додаток В).

Система коректно обробила запит, повернувши відповідь про неправильний логін або пароль, без жодних винятків, зависань або помилок з боку сервера.

9) Підміна вмісту.

Виникає тоді, коли зловмисник намагається змінити або сфальсифікувати інформацію, що зберігається у системі, аби вона була прийнята як справжня. У веб-додатках це може проявлятися у вигляді змінених повідомлень, які виглядають достовірними, але насправді були підроблені або змінені вручну.

У моєму додатку особливу увагу приділено захисту текстових повідомлень у чаті. Кожне повідомлення після надсилання зберігається у базі даних у зашифрованому вигляді (AES), а разом із ним зберігається також SHA-256 хеш, обчислений від відкритого тексту повідомлення, ідентифікатора відправника та мітки часу. Під час кожного запиту на отримання повідомлень система спочатку виконує розшифрування, після чого обчислює хеш заново й порівнює його з тим, що був збережений у базі. Якщо хеши не збігаються — повідомлення вважається пошкодженим або потенційно фальсифікованим.

Щоб перевірити, як система реагує на спробу підміни, я вручну змінив хеш одного з повідомлень у базі, не змінюючи сам зашифрований текст повідомлення (оскільки він недоступний для дешифрування без ключа) (Додаток В). Таким чином було змодельовано ситуацію, коли зловмисник намагається змінити хеш і змусити систему прийняти підроблене повідомлення за справжнє.

Після цього при виконанні запиту GET /api/chat/messages (Додаток В) система коректно обробила повідомлення, але у відповіді повернула його з міткою

[CORRUPTED MESSAGE] — тобто система розшифрувала текст, але не прийняла його як достовірний через невідповідність контрольної суми.

Це свідчить про те, що навіть у разі втручання в базу даних, повідомлення не буде показане користувачам як справжнє, якщо хоча б частково порушено його цілісність.

10) Прогнозування облікових/сесійних даних.

Ця вразливість полягає в тому, що ідентифікатори користувачів, токени сесій або інші чутливі значення генеруються передбачуваним чином. Якщо, наприклад, JWT або ID користувача легко вгадати — зловмисник може отримати доступ до чужого акаунта або даних.

У моєму додатку всі сесійні механізми реалізовано через JWT, який підписується за допомогою алгоритму HMAC SHA-512 з використанням надійного, довгого секретного ключа, заданого у файлі конфігурації.

Токени не зберігаються на сервері — вони повністю самодостатні. При кожному запиті фільтр JwtAuthFilter перевіряє їхню валідність, підпис, дату завершення дії та наявність у чорному списку. Якщо токен підроблено вручну або навіть незначно змінено — система одразу повертає статус 401 Unauthorized.

Щоб переконатися в стійкості до прогнозування, я навмисно згенерував токен з валідною структурою, але неправильним підписом. Приклад генерації підробленого токену наведено у Додатку В.

При спробі виконати будь-який запит з таким токеном (GET /api/chat/messages) (Додаток В) система одразу повернула 401 — тобто жодного ризику використання передбачуваного або підробленого токена немає.

11) Міжсайтове скриптування.

У додатку передбачено захист від XSS-повідомлень, які можуть містити HTML або скрипти. Повідомлення, що надсилаються у чат, зберігаються у зашифрованому вигляді. Після розшифрування вони автоматично обробляються методом

StringEscapeUtils.escapeHtml4(...), який замінює потенційно небезпечні HTML-символи на безпечні еквіваленти.

Щоб перевірити стійкість до XSS, я надіслав JavaScript код у полі текст (Додаток В).

Після дешифрування на клієнт надходив неактивний код, який не виконувався у браузері, а виводився як звичайний текст (Додаток В). Це підтверджує, що екранування працює правильно і шкідливі вставки не можуть бути відтворені.

12) Відмова в обслуговуванні.

У додатку реалізовано захист від надмірної активності з боку одного користувача через сервіс RateLimiterService. Він обмежує кількість запитів, які може надіслати користувач протягом певного інтервалу часу. Якщо ліміт перевищено, система блокує наступні запити на деякий час.

У моєму випадку встановлено обмеження: не більше 20 запитів на хвилину на користувача. Якщо цей поріг перевищено, кожен наступний запит завершується зі статусом 429 Too Many Requests і відповідним повідомленням.

Щоб перевірити цей механізм, я виконав 21 запит на надсилання повідомлень (POST /api/chat/send) протягом однієї хвилини. Налаштування спаму наведено у Додатку В.

Після двадцятої спроби система повернула помилку 429 (Додаток В) — це означає, що захист від DoS-атаки через часте надсилання запитів працює коректно.

Це підтверджує, що система стійка до надмірного навантаження з боку окремих користувачів і здатна захищатися від спроб перевантаження шляхом повторних звернень.

13) Впровадження команд.

У моєму додатку не використовується прямий виклик системних команд або виконання shell-інструкцій на стороні сервера, що є основною передумовою для появи цієї вразливості. Вся логіка побудована на основі Java-коду, а обробка введення

користувача проходить через стандартні API без використання Runtime.exec(), ProcessBuilder або аналогів.

Для перевірки на наявність потенційної вразливості я здійснив запити, у яких у поля повідомлення (text) вставлялись рядки, схожі на shell-команди (Додаток В).

Система приймала такі запити як звичайний текст, зберігала їх у зашифрованому вигляді й не виконувала жодних дій. Під час отримання ці повідомлення розшифровувались і повертались у відповідь без будь-якого виконання або зміни поведінки системи.

Це підтверджує, що додаток не містить вразливості до впровадження команд, оскільки не обробляє введення як інструкцію до виконання на сервері.

14) Обхід шляхів.

У додатку відсутня функціональність, що передбачає роботу з файловою системою на основі введених користувачем шляхів. Система не дозволяє завантажувати, зчитувати або передавати файли за вказаним шляхом, тому можливість обходу директорій за допомогою конструкцій типу ../ архітектурно виключена.

Для перевірки цієї вразливості було виконано тестовий запит із передачею підозрілого рядка, що імітує спробу доступу до системного файлу (Додаток В).

Цей рядок було передано як звичайне повідомлення у чаті. Після шифрування і збереження повідомлення не викликало жодної реакції з боку системи, і при спробі його отримання відображалось як звичайний текст.

15) SQL-ін’єкція.

У моєму додатку усі взаємодії з базою даних реалізовані через Spring Data JPA, що виключає використання сирих SQL-запитів. Замість ручного формування запитів використовуються методи репозиторіїв з параметрами, які автоматично обробляються фреймворком. Це запобігає можливості вставки шкідливого SQL-коду у запити.

Для перевірки на вразливість до SQL-ін’єкції я виконав кілька запитів із введенням ін’єкції у полі username:

Жоден із запитів не призвів до помилок, винятків або небажаної поведінки. Система не авторизувала користувача та повертала стандартну відповідь про недійсні облікові дані. Це свідчить про те, що фільтрація введення виконується на рівні JPA, а дані не вставляються напряму у SQL-запити.

Таким чином, застосунок захищений від SQL-ін’єкцій завдяки використанню параметризованих запитів і високорівневих абстракцій JPA.

16) Недостатнє завершення сесії.

У моєму додатку реалізовано явне завершення сесії через ендпоінт POST /api/auth/logout. Після його виклику токен користувача додається до чорного списку (TokenBlacklistService) і вважається недійсним. Подальші запити з цим токеном блокуються фільтром JwtAuthFilter — навіть якщо сам токен ще дійсний за строком та підписом.

Щоб перевірити правильність роботи механізму, я виконав логін, отримав токен, а потім здійснив POST /api/auth/logout з ним. Після цього при спробі доступу до захищеного ресурсу (/api/user/profile) система повертала 401 Unauthorized, підтверджуючи, що токен було успішно відкликано. Вихід користувача та спроба доступу до захищеного ресурсу наведено у Додатку В.

Крім того, у додатку реалізовано автоматичне очищення чорного списку токенів при запуску сервера (через компонент StartupCleanup). Це означає, що після перезапуску застосунку всі користувачі, навіть ті, хто не виконав logout, будуть примусово розлогінені, і жоден старий токен не зможе бути використаний. Приклад завершення роботи сервера та спроби доступу до ресурсів після перезапуску наведено у Додатку В.

17) Неправильна конфігурація застосунку.

У моєму додатку було приділено окрему увагу налаштуванням безпеки, щоб уникнути ситуацій, коли через недбалі конфігурації система стає вразливою до атак. Додаток не надає доступу до конфіденційних маршрутів без автентифікації, не

розкриває службову інформацію в логах, а також використовує HTTPS для всіх з'єднань, завдяки вбудованому SSL-сертифікату.

У файлі конфігурації application.properties:

- H2 Console доступна лише в режимі розробки, за шляхом /h2-console, і захищена авторизацією (Додаток В);
- Ввімкнено HTTPS (порт 8443) із власним .p12-сертифікатом (Додаток В);
- Для JWT використовується криптостійкий секрет, що зберігається не у відкритому коді, а у конфігурації (Додаток В).

Також не використовуються налаштування на кшталт spring.jpa.hibernate.ddl-auto=create, які могли б автоматично скидати базу або створювати нові таблиці при кожному запуску.

18) Фіксація сесії.

У додатку автентифікація реалізована повністю на основі JWT, який створюється лише після успішного входу в систему. Жоден токен не видається заздалегідь або до моменту авторизації. Це означає, що в принципі неможливо «закріпити» сесію наперед, як це часто буває при атаках фіксації сесії, коли зловмисник намагається нав'язати жертві вже відомий ідентифікатор сесії.

4.3 Тестування правильності роботи шифрування повідомлень

Окрім перевірки стійкості до атак, було окремо протестовано правильність реалізації шифрування повідомлень, яке відбувається при обміні у чаті. Повідомлення шифруються за схемою AES + RSA: текст шифрується унікальним AES-ключем, а цей ключ — публічним RSA-ключем сервера. Розшифрування відбувається лише у випадку, якщо всі елементи — шифротекст, зашифрований ключ та хеш — є валідними.

Для підтвердження правильності реалізації було проведено такі тести:

1) Перевірка збереження зашифрованого тексту у базі

Було надіслано звичайне повідомлення через POST /api/chat/send. Після цього у базі даних було перевірено вміст поля text у таблиці chat_message (Додаток В). Повідомлення зберігалося не у відкритому вигляді, а як строка у форматі Base64. Це свідчить про те, що текст дійсно був зашифрований перед збереженням.

2) Перевірка унікальності AES-ключів

Було двічі надіслано одне й те саме повідомлення від одного користувача (Додаток В). При порівнянні записів у базі з'ясувалось, що значення поля encrypted_key відрізняється, хоча текст був однаковий. Це означає, що AES-ключ для кожного повідомлення генерується динамічно, що відповідає вимогам до симетричного шифрування.

3) Перевірка правильності дешифрування повідомлення

Після надсилання повідомлення користувач звернувся до GET /api/chat/messages (Додаток В). Якщо дані в базі не були змінені вручну, повідомлення успішно дешифрувалося й повернулося у вихідному вигляді. Це підтверджує, що дешифрування працює коректно при наявності валідного AES-ключа, зашифрованого RSA.

4) Перевірка поведінки при спотворенні зашифрованого AES-ключа

У базі вручну було змінено поле encrypted_key одного з повідомлень. Після цього при зверненні до /api/chat/messages повідомлення не вдалося розшифрувати — система повернула [DECRYPTION ERROR] (Додаток В). Це свідчить про те, що без правильного ключа розшифрування є неможливим.

5) Перевірка поведінки при спотворенні RSA-приватного ключа сервера

У тестовому середовищі сервер було перезапущено з іншим RSA-ключем. Усі повідомлення, які були збережені раніше, більше не підлягали розшифруванню — система повертала [DECRYPTION ERROR] (Додаток В). Це доводить, що AES-ключ дійсно був зашифрований RSA-приватним ключем, і без відповідного ключа його неможливо відновити.

4.4 Перспективи подальшого розвитку веб-додатка

На поточному етапі розроблений веб-додаток забезпечує базову функціональність захищеного обміну повідомленнями з повноцінною підтримкою авторизації, контролю доступу, шифрування повідомлень, ведення журналу дій та стійкості до більшості поширених загроз інформаційної безпеки. Проте в перспективі його можна суттєво розширити як з функціонального, так і з архітектурного боку.

Насамперед доцільно розглянути реалізацію двофакторної автентифікації (2FA) — наприклад, за допомогою одноразових кодів або мобільного застосунку. Це підвищить рівень захисту облікових записів.

Ще одним напрямком є перехід від концепції загального чату до підтримки приватного спілкування між користувачами. Це дозволить реалізувати індивідуальні діалоги, групи, а згодом — систему запитів на дружбу, списків контактів або навіть тимчасових кімнат. З технічної точки зору, для цього потрібно зберігати інформацію про відправника й одержувача у кожному повідомленні, а також адаптувати логіку фільтрації при отриманні історії чату.

З погляду захисту даних, система вже використовує гібридне шифрування, але у перспективі можлива інтеграція з апаратними криптомодулями (HSM) або зовнішніми KMS-сервісами.

Функціональні покращення можуть включати:

- пошук по історії повідомлень;
- підтримку вкладень із шифруванням;
- реактивний інтерфейс (WebSocket, SSE) для миттєвого оновлення.

На рівні адміністрування доцільно реалізувати розширене управління журналами дій — з фільтрами, візуалізацією та можливістю експорту.

У довгостроковій перспективі можливе поступове впровадження мікросервісної архітектури, де автентифікація, чат, шифрування й аудит функціонують як незалежні сервіси.

ВИСНОВКИ

У процесі виконання дипломної роботи було проведено всебічний аналіз загроз, які виникають при створенні сучасних веб-додатків, з урахуванням міжнародних стандартів безпеки, зокрема моделі OWASP Top 10. Основна увага приділялась не лише теоретичному узагальненню типових вразливостей, а й практичному втіленню захисних механізмів у реальному програмному рішенні.

На основі виявлених ризиків і класифікації вразливостей за фазами життєвого циклу програмного забезпечення було побудовано архітектуру веб-серверного застосунку, який реалізує сучасні підходи до захисту інформації. У розробленій системі було реалізовано надійний механізм автентифікації з використанням токенів JWT, які мають цифровий підпис, термін дії та можливість відкликання. Всі маршрути захищено на рівні контролерів відповідно до ролей користувачів, а також запроваджено механізм повного й тимчасового блокування облікових записів із веденням журналу дій. Повідомлення у чаті зберігаються виключно у зашифрованому вигляді, а шифрування реалізовано за гібридною схемою AES + RSA, що дозволяє забезпечити конфіденційність як тексту повідомлення, так і ключа. Для контролю цілісності використано хешування SHA-256. З'єднання з сервером захищено через HTTPS, а обробка запитів обмежена на рівні частоти, щоб запобігти перевантаженням і спаму.

Особливу увагу в рамках роботи було приділено тестуванню. Система пройшла перевірку як на рівні окремих компонентів (через юніт-тести), так і в цілому — через інтеграційне тестування на предмет стійкості до найпоширеніших атак. Було доведено, що додаток не допускає SQL-ін’єкцій, XSS, CSRF, brute-force, прогнозування токенів, несанкціонованого доступу, помилок конфігурації або фіксації сесій. Окремо було протестовано коректність реалізації шифрування: повідомлення не розшифровуються без приватного ключа сервера, AES-ключі є унікальними, а всі спроби спотворити текст або ключ виявляються на етапі перевірки

хешу. Така поведінка доводить надійність і логічну цілісність криптографічної частини системи.

Результати роботи свідчать про те, що розроблений веб-додаток демонструє реальну стійкість до типових загроз і може слугувати прикладом безпечної архітектури веб-рішень на базі Java. При цьому система залишає простір для подальшого розвитку. У майбутньому доцільно реалізувати приватні чати між користувачами, двофакторну автентифікацію, підтримку вкладень із шифруванням, а також перейти до реактивної моделі роботи за допомогою WebSocket або Server-Sent Events. Архітектура проєкту дозволяє масштабувати його до мікросервісної моделі, у якій сервіси автентифікації, обробки повідомлень, журналювання та шифрування можуть функціонувати незалежно.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. International Journal of Engineering Science and Innovative Technology (IJESIT) Volume 2, Issue 2, March 2013 [Електронний ресурс]. – Режим доступу: https://ijesit.com/Volume%202/Issue%202/IJESIT201302_35.pdf

2. M. Nawrocki, J. Kołodziej, “Vulnerabilities of web applications: Good practices and new trends,” ACIG, vol. 3, no. 2, 2024, pp. 122– 143. DOI: 10.60097/ ACIG/199521 [Електронний ресурс]. – Режим доступу: https://www.acigjournal.com/pdf-199521-123641?filename=Vulnerabilities%20of%20Web.pdf

3. OWASP Top 10 – 2021 [Електронний ресурс]. – Режим доступу: https://owasp.org/Top10/

4. Mitigating the OWASP Top 10 Vulnerabilities: Strategies for Protecting Your Systems [Електронний ресурс]. – Режим доступу: https://certera.com/blog/mitigating-the-owasp-top-10-vulnerabilities/

5. 10 Web Application Security Threats and How to Mitigate Them. StackHawk [Електронний ресурс]. – Режим доступу: https://www.stackhawk.com/blog/10-web-application-security-threats-and-how-to-mitigate-them/

6. Understanding the Science Behind a High-Performing Web Application Architecture [Електронний ресурс]. – Режим доступу: https://softteco.com/blog/web-application-archtecture-explained

7. Monolithic vs microservices architecture: Which is better for security? [Електронний ресурс]. – Режим доступу: https://www.invicti.com/blog/web-security/monolithic-vs-microservices-architecture-which-is-better-for-security/

8. What is three-tier architecture? [Електронний ресурс]. – Режим доступу: https://www.ibm.com/think/topics/three-tier-architecture

9. Three-Tier Client Server Architecture in Distributed System [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/three-tier-client-server-architecture-in-distributed-system/

10. From Login to Lockdown: Building Secure Authenticated Applications [Електронний ресурс]. – Режим доступу: https://www.authgear.com/post/web-application-authentication-guide

11. Definition authentication [Електронний ресурс]. – Режим доступу: https://www.techtarget.com/searchsecurity/definition/authentication

12. Authentication and Authorization Techniques in Modern Web Applications [Електронний ресурс]. – Режим доступу: https://dev.to/divine_nnanna2/authentication-and-authorization-techniques-in-modern-web-applications-2okl

13. What is the difference between symmetric and asymmetric encryption in web application security? [Електронний ресурс]. – Режим доступу: https://www.linkedin.com/advice/1/what-difference-between-symmetric asymmetric-zadde

14. Advanced Encryption Standard (AES) [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/advanced-encryption-standard-aes/

15. What is an Asymmetric Encryption? [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/what-is-asymmetric-encryption/

16. 5 Super Asymmetric Encryption Example Use Cases [Електронний ресурс]. – Режим доступу: https://cyberexperts.com/asymmetric-encryption-example/

17. What are the disadvantages of asymmetric encryption? [Електронний ресурс]. – Режим доступу: https://www.tencentcloud.com/techpedia/105344

18. SSL/TLS - A Hybrid Crypto System [Електронний ресурс]. – Режим доступу: https://www.binderror.com/blog/cryptography_101/

19. 2 Java Cryptography Architecture (JCA) Reference Guide [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/en/java/javase/17/security/java-cryptography-architecture-jca-reference-guide.html#GUID-2BCFDD85-D533-4E6C-8CE9-29990DEB0190

20. Encoding, Encryption, Hashing, and Obfuscation in Java [Електронний ресурс]. – Режим доступу: https://www.cesarsotovalero.net/blog/encoding-encryption-hashing-and-obfuscation-in-java.html

21. Password Storage [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-security/reference/features/authentication/password-storage.html

ДОДАТОК А

ВИХІДНИЙ КОД КЛАСУ JWTAUTHFILTER

```
@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    private final JwtUtils jwtUtils;
    private final CustomUserDetailsService userDetailsService;
    private final TokenBlacklistService tokenBlacklistService;
    private final UserRepository userRepository;

    public JwtAuthFilter(JwtUtils jwtUtils,
                         CustomUserDetailsService userDetailsService,
                         TokenBlacklistService tokenBlacklistService,
                         UserRepository userRepository) {
        this.jwtUtils = jwtUtils;
        this.userDetailsService = userDetailsService;
        this.tokenBlacklistService = tokenBlacklistService;
        this.userRepository = userRepository;
    }

    @Override
    protected boolean shouldNotFilter(HttpServletRequest request) {
        String uri = request.getRequestURI();
        return uri.startsWith("/api/auth/register") ||
uri.startsWith("/api/auth/login");
    }


    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException,
IOException {

        String authHeader = request.getHeader("Authorization");

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String jwt = authHeader.substring(7).trim();

            if (tokenBlacklistService.isTokenBlacklisted(jwt)) {
                response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                response.setContentType("text/plain");
                response.getWriter().write("Token has been revoked. Please log in
again.");
                return;
            }

            String username;
            try {
                username = jwtUtils.extractUsername(jwt);
            } catch (RuntimeException e) {
                if ("Invalid JWT token.".equals(e.getMessage())) {
                    response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                    response.setContentType("text/plain");
                    response.getWriter().write("Invalid JWT token.");
                    return;
                }
```

```
                throw e;
            }

            if (username != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
                UserDetails userDetails =
userDetailsService.loadUserByUsername(username);

                User user = userRepository.findByUsername(username).orElse(null);
                if (user != null && user.isBlocked()) {
                    response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                    response.setContentType("text/plain");
                    response.getWriter().write("User account is blocked.");
                    return;
                }

                if (jwtUtils.validateToken(jwt, userDetails)) {
                    UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(
                            userDetails, null, userDetails.getAuthorities()
                    );
                    authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                    SecurityContextHolder.getContext().setAuthentication(authToken);
                }
            }
        }

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

ДОДАТОК Б

СКРІНШОТИ ПРАКТИЧНОЇ РЕАЛІЗАЦІЇ ДОДАТКУ

POST https://localhost:8443/api/auth/register-user Send

Params Authorization Headers (10) Body Scripts Settings Cookies

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

```
1 {
2   "username": "user3",
3   "password": "user123"
4 }
```

Body Cookies (1) Headers (15) Test Results 200 OK 160 ms 517 B Save Response

Raw Preview Visualize

```
1 User registered successfully.
```

Рисунок Б.1 – Приклад запиту для реєстрації користувача

POST https://localhost:8443/api/auth/login Send

Params Authorization Headers (10) Body Scripts Settings Cookies

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

```
1 {
2   "username": "user3",
3   "password": "user123"
4 }
```

Body Cookies (1) Headers (15) Test Results 200 OK 111 ms 663 B Save Response

Raw Preview Visualize

1 eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ1c2VyMyIsImlhdCI6MTc0NzU5NDE4OCwiZXhwIjoxNzQ3NjgwNTg4fQ.
_yHAUjowM_RcLeXLikzcQX2EpkH7dy1F2GLx7ON3SM4bA6eDw6rktmn95PeGjfuiM-0fOsoErTWHjllHHLYMYQ

Рисунок Б.2 – Приклад отримання токена після успішної авторизації

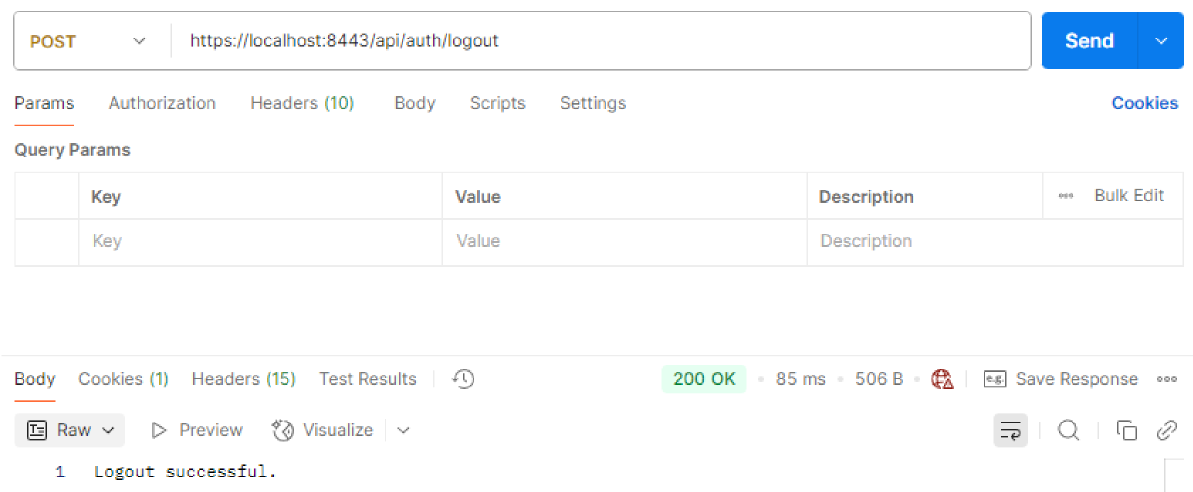

Рисунок Б.3 – Вихід користувача із системи

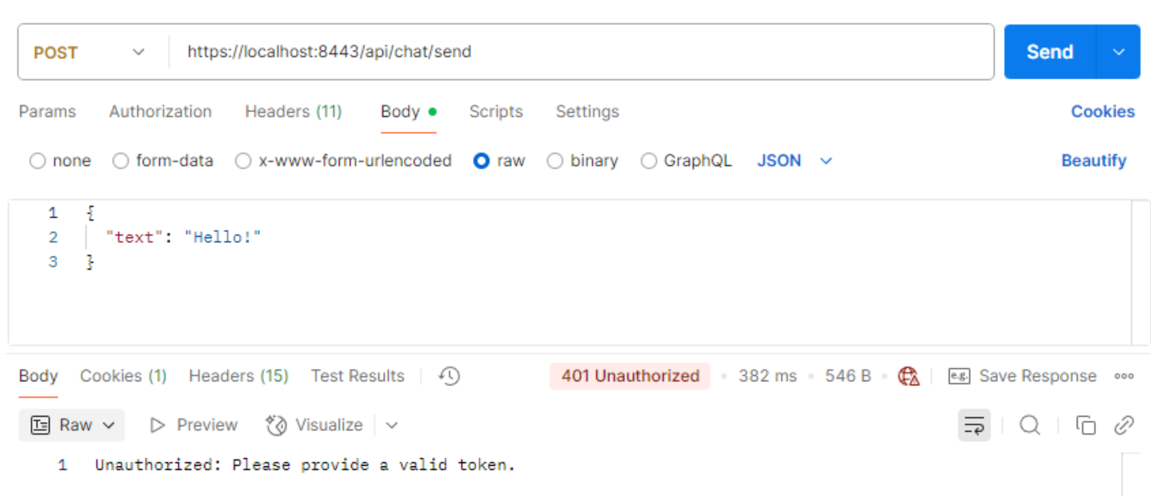

Рисунок Б.4 – Спроба зайти за старим токеном

Рисунок Б.5 – Приватний та публічний ключі сервера

POST https://localhost:8443/api/auth/register-admin

```
{
  "username": "admin3",
  "password": "admin123",
  "secret": "SECRET123"
}
```

200 OK · 148 ms · 931 B

```
Admin registered successfully.
Server public key: MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2Faawe+yDZD/
rN5c3hIqMnP6EyDNBNo7TwZ82rFEmBxPXUAAhjRdskqckEC3ie+sjqvLXCDqQAODJrcu8I3SWR8CJ2PHsfj0+kzDv3WuOQ/
xfBsRpdchClw/DJ+JIaZdCRcjCJg4AOhVQtsPrj+RrBXpnuDPpNm6
+RLPUJE52zmuGGcEfmzmqPu444djxBHaZ6DM4CcmLcWo7t1tG6+2063aWZI/H9D
+xr4peqAr5O01H0SUl3ChR9zClpvp5uiXAEKZW1xkhpNy5wKUKvA2SKNs7emFWBS3rJ9pnXWNlm
+5FXSgh5huk0l8f6meGq2UdaixCce7QAOVRNEruuf1oQIDAQAB
```

Рисунок Б.6 – Отримання публічного ключа сервера при реєстрації

CHAT_MESSAGE

ID	ENCRYPTED_KEY	HASH	SENDER	TEXT	TIMESTAMP
1	Z4hfi7V4KPlDRFEzqmm013…	d887e3939377bd38747eb0…	user3	JKZW6lri1k11e/pUlk4gL4…	2025-05-18 22:12:58.86…

Рисунок Б.7 – Приклад збереженого зашифрованого повідомлення

```
PS C:\Users\mants\diploma> keytool -genkeypair -alias secureapp -keyalg RSA -keystore keystore.p12 -storetype PKCS12 -validity 365 -storepass yourPassword
```

keystore.p12

Visual layout of bidirectional text can depend on the base direction … Choose direction Hide notification Don't show again

This file was explicitly re-assigned to plain text Remove association Edit File Types

Рисунок Б.8 – Команда для створення сертифікату та її результат

keystore.p12 | application.properties ×

```
1  # ========================
2  # SERVER CONFIGURATION (HTTPS)
3  # ========================
4  server.port=8443
5  server.ssl.enabled=true
6  server.ssl.key-store=classpath:keystore.p12
7  server.ssl.key-store-password=password
8  server.ssl.key-store-type=PKCS12
9  server.ssl.key-alias=secureapp
10 admin.secret=SECRET123
11
```

Рисунок Б.9 – Налаштування HTTPS з'єднання

POST https://localhost:8443/api/chat/send Send

Params Authorization Headers (11) Body Scripts Settings Cookies

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

```
1 {
2   "text": "Hello!"
3 }
```

Body Cookies (1) Headers (15) Test Results 200 OK 82 ms 515 B Save Response

Raw Preview Visualize

```
1  Message encrypted and sent.
```

Рисунок Б.10 – Приклад відправлення повідомлення

CHAT_MESSAGE ×

Tx: Auto DDL CSV

WHERE ORDER BY

	ID	ENCRYPTED_KEY	HASH	SENDER	TEXT
1	1	Z4hfi7V4KPLDRFEzqmm013Jf46…	d887e3939377bd38747eb0cbc8…	user3	JKZW6lri1k11e/pUlk4gL4cd0h…
2	2	eC+MRHqcHbxhlkB87DybcjksYJ…	0e718b2ef68eed9895cddfd4b6f533b49ea36e288c436aa625cdb17f380b082f		gIVtwle…

Рисунок Б.11 – Приклад зберігання хешу у БД

GET https://localhost:8443/api/chat/messages Send

Params | Authorization | Headers (8) | Body | Scripts | Settings | Cookies

Headers 7 hidden

Key	Value	Description
Authorization	Bearer {{user_token}}	
Key	Value	

eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ1c2VyMyIsImlhdCI6MTc0NzU5Nzg2MSwiZXhwIjoxNzQ3Njg0MjYxfQ.J8zwcSIT2q2DLpxY5hze74UQed07LyBReyDiC-Yfp3Y91Xa-7aL9__BNjAankL-8o0NvJ6MvsXaJ8d0eYDbwuQ

Environment Variables in request →

Body | Cookies (1) | Headers (15) | Test Results | Save Response

JSON | Preview | Visualize

```
1  [
2      {
3          "sender": "user3",
4          "text": "Hello!",
5          "timestamp": "2025-05-18T22:12:58.864772"
6      },
7      {
8          "sender": "user3",
9          "text": "Hello!",
10         "timestamp": "2025-05-18T22:51:35.582607"
11     }
12 ]
```

Рисунок Б.12 – Результат отримання повідомлення

CHAT_MESSAGE | USERS

Tx: Auto | DDL | CSV

WHERE | ORDER BY

	ID	BLOCK_EXPIRES_AT	BLOCKED	PASSWORD	ROLE	USERNAME
1	1	<null>	false	$2a$10$sTBqziI58mkokD4QbHu0TOIf4fDnhdBxv.…	ROLE_USER	user3
2	2	<null>	false	$2a$10$WsC8Vl.WqJdwFNZldBPuJuqCElbwus4Cch…	ROLE_ADMIN	admin3

Рисунок Б.13 – Приклад збереження користувача у БД

GET https://localhost:8443/api/admin/logs Send

Params | Authorization | Headers (8) | Body | Scripts | Settings | Cookies

Body | Cookies (1) | Headers (15) | Test Results | 200 OK • 51 ms • 1.33 KB | Save Response

JSON | Preview | Visualize

```
43     },
44     {
45         "id": 8,
46         "username": "user3",
47         "action": "User logout",
48         "timestamp": "2025-05-18T23:14:44.038885"
49     },
50     {
51         "id": 9,
52         "username": "admin3",
53         "action": "Successful login",
54         "timestamp": "2025-05-18T23:20:52.267918"
55     }
56 ]
```

Рисунок Б.14 – Перегляд таблиці логування у режимі адміністратора

```
Hibernate: insert into audit_log (action,timestamp,username,id) values (?,?,?,default)
2025-05-18T23:20:47.512+03:00 DEBUG 12672 --- [nio-8443-exec-4] o.s.security.web.FilterChainProxy        : Securing GET /api/admin/logs
Hibernate: select u1_0.id,u1_0.block_expires_at,u1_0.blocked,u1_0.password,u1_0.role,u1_0.username from users u1_0 where u1_0.username=?
Hibernate: select u1_0.id,u1_0.block_expires_at,u1_0.blocked,u1_0.password,u1_0.role,u1_0.username from users u1_0 where u1_0.username=?
2025-05-18T23:20:47.533+03:00 DEBUG 12672 --- [nio-8443-exec-4] o.s.s.w.a.AnonymousAuthenticationFilter  : Set SecurityContextHolder to anonymous SecurityContext
2025-05-18T23:20:47.533+03:00 DEBUG 12672 --- [nio-8443-exec-4] o.s.s.w.session.SessionManagementFilter  : Request requested invalid session id AA9CBAE8A4D1CD56B7FC71042
2025-05-18T23:20:52.021+03:00 DEBUG 12672 --- [nio-8443-exec-5] o.s.security.web.FilterChainProxy        : Securing POST /api/auth/login
2025-05-18T23:20:52.021+03:00 DEBUG 12672 --- [nio-8443-exec-5] o.s.s.w.a.AnonymousAuthenticationFilter  : Set SecurityContextHolder to anonymous SecurityContext
2025-05-18T23:20:52.021+03:00 DEBUG 12672 --- [nio-8443-exec-5] o.s.s.w.session.SessionManagementFilter  : Request requested invalid session id AA9CBAE8A4D1CD56B7FC71042
2025-05-18T23:20:52.021+03:00 DEBUG 12672 --- [nio-8443-exec-5] o.s.security.web.FilterChainProxy        : Secured POST /api/auth/login
Hibernate: select u1_0.id,u1_0.block_expires_at,u1_0.blocked,u1_0.password,u1_0.role,u1_0.username from users u1_0 where u1_0.username=?
Hibernate: select u1_0.id,u1_0.block_expires_at,u1_0.blocked,u1_0.password,u1_0.role,u1_0.username from users u1_0 where u1_0.username=?
2025-05-18T23:20:52.244+03:00 DEBUG 12672 --- [nio-8443-exec-5] o.s.s.a.dao.DaoAuthenticationProvider    : Authenticated user
Hibernate: insert into audit_log (action,timestamp,username,id) values (?,?,?,default)
2025-05-18T23:20:59.687+03:00 DEBUG 12672 --- [nio-8443-exec-6] o.s.security.web.FilterChainProxy        : Securing GET /api/admin/logs
Hibernate: select u1_0.id,u1_0.block_expires_at,u1_0.blocked,u1_0.password,u1_0.role,u1_0.username from users u1_0 where u1_0.username=?
Hibernate: select u1_0.id,u1_0.block_expires_at,u1_0.blocked,u1_0.password,u1_0.role,u1_0.username from users u1_0 where u1_0.username=?
2025-05-18T23:20:59.699+03:00 DEBUG 12672 --- [nio-8443-exec-6] o.s.security.web.FilterChainProxy        : Secured GET /api/admin/logs
Hibernate: select al1_0.id,al1_0.action,al1_0.timestamp,al1_0.username from audit_log al1_0
```

Рисунок Б.15 – Загальне логування системи

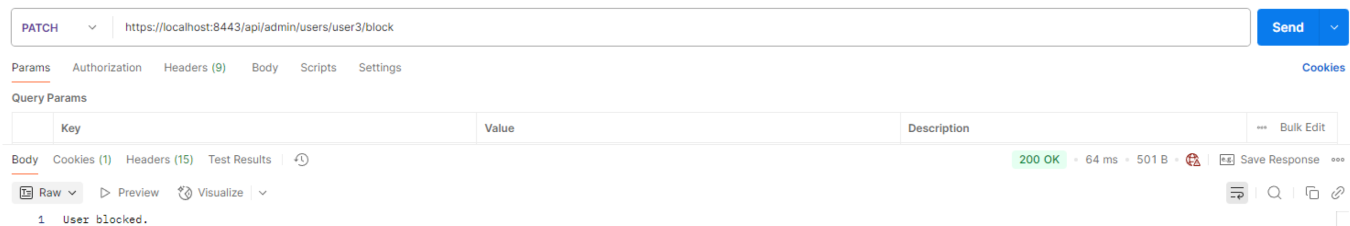

Рисунок Б.16 – Блокування користувача

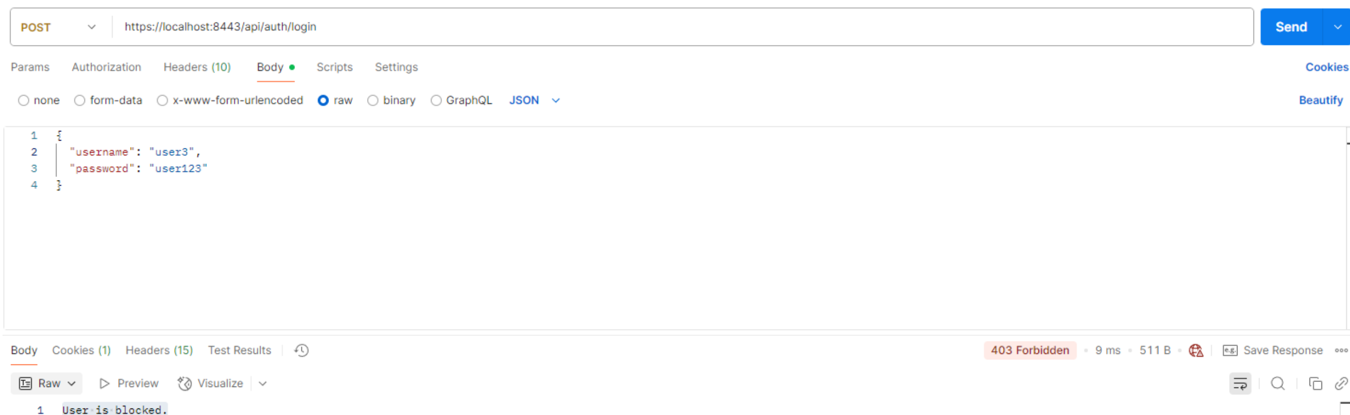

Рисунок Б.17 – Спроба увійти після блокування

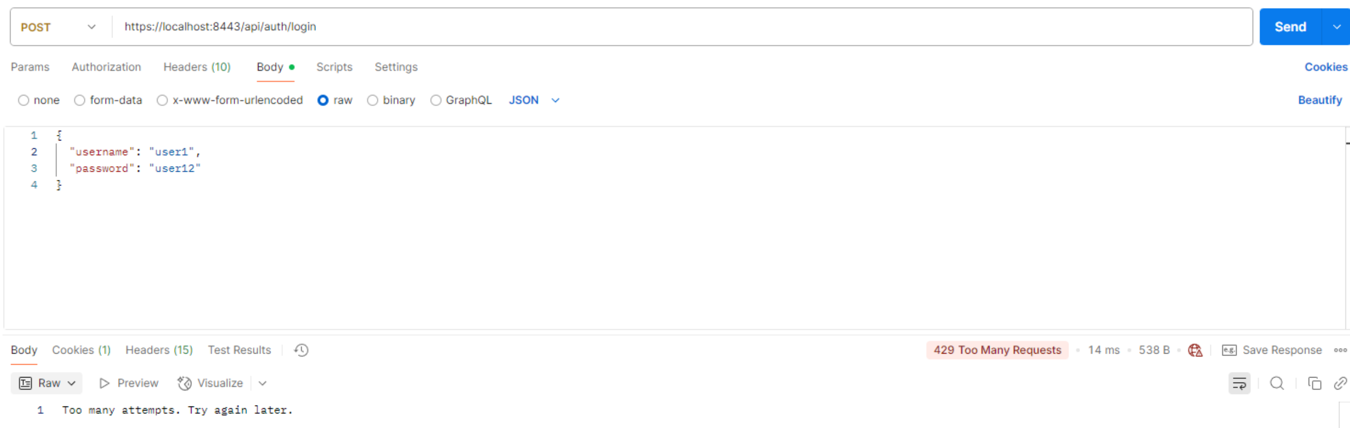

Рисунок Б.18 – Блокування після 5-ти невдалих спроб вводу пароля

{} JSON Preview Visualize


```json
[
    {
        "id": 2630,
        "username": "admin",
        "role": "ROLE_ADMIN",
        "blocked": false,
        "blockExpiresAt": null
    },
    {
        "id": 2631,
        "username": "user3",
        "role": "ROLE_USER",
        "blocked": false,
        "blockExpiresAt": null
    },
    {
        "id": 2663,
        "username": "admin3",
        "role": "ROLE_ADMIN",
        "blocked": false,
        "blockExpiresAt": null
    }
]
```


Body Cookies Headers (15) Test Results

{} JSON Preview Visualize


```json
[
    {
        "id": 2630,
        "username": "admin",
        "role": "ROLE_ADMIN",
        "blocked": false,
        "blockExpiresAt": null
    },
    {
        "id": 2663,
        "username": "admin3",
        "role": "ROLE_ADMIN",
        "blocked": false,
        "blockExpiresAt": null
    }
]
```


DELETE https://localhost:8443/api/admin/users/user3 Send

Params Authorization Headers (7) Body Scripts Settings Cookies

Headers 6 hidden

Key	Value	Description
Authorization	Bearer {{admin_token}}	
Key	Value	Description

Body Cookies Headers (15) Test Results 200 OK 53 ms 501 B Save Response

Raw Preview Visualize

```
1  User deleted.
```

Рисунок Б.19 – Приклад видалення користувача в системі

DELETE https://localhost:8443/api/admin/messages/1 Send

Params Authorization Headers (8) Body Scripts Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies (1) Headers (15) Test Results 200 OK 318 ms 504 B Save Response

Raw Preview Visualize

```
1  Message deleted.
```

Рисунок Б.20 – Видалення повідомлення з id = 1

GET https://localhost:8443/api/chat/messages Send

Params Authorization Headers (8) Body Scripts Settings Cookies

Headers 7 hidden

Key	Value	Description
Authorization	Bearer {{user_token}}	
Key	Value	Description

Body Cookies (1) Headers (15) Test Results 200 OK 34 ms 1008 B Save Response

{} JSON Preview Visualize


```
1  [
2      {
3          "id": 2,
4          "sender": "user3",
5          "text": "Hello!",
6          "encryptedKey": "eC+MRHqcHbxhlkB87DybcjksYJtwSRifzeVnS1HWI6BkGghfMVU4lbtk8EHpBMSy9C+sY1b9baR5/ZPjkmFW2Qj8D3xxVNny7c2iKpfMUCougefJaUJkGnfp4W9MoK20/mOvS2UF85cVKc
             +AgaJpqBPQYpQq5PthV97rIMi91trkxg+GfvE+9uo84tCQhJHAS4DtyVBFzHyvBNBGoKXZaisuFnQad9bTZ9QD3fduiYlVRIgxaHLuLrRE0fDnCI8NdVpTgf6RrAYNlkvPLyyfsvvUytZb0qTiHRsQhecqRgBBiZys53/
             Q7Gr9ok33yzHFSCEozewQLSJ8y8KpFM018A==",
7          "hash": "0e718b2ef68eed9895cddfd4b6f533b49ea36e288c436aa625cdb17f380b082f",
8          "timestamp": "2025-05-18T22:51:35.582607"
9      }
10 ]
```



```
68     {
69         "id": 12,
70         "username": "ADMIN",
71         "action": "Deleted message ID: 1",
72         "timestamp": "2025-05-18T23:46:19.557957"
73     }
74 ]
```


Рисунок Б.21 – Результат видалення повідомлення

ДОДАТОК В

СКРІНШОТИ ТЕСТУВАННЯ РОЗРОБЛЕНОГО ДОДАТКУ

Test	Time
config (com.diploma.secureapp)	769 ms
SecurityConfigTest	598 ms
contextLoadsAndBeansAreNotNull()	598 ms
StartupCleanupTest	171 ms
testClearBlacklistOnStartupIsCalled()	171 ms

Рисунок В.1 – Результат виконання тестів пакету config

Test	Time
controller (com.diploma.secureapp)	5 sec 924 ms
UserControllerTest	1 sec 133 ms
AdminControllerTest	1 sec 728 ms
ChatControllerTest	2 sec 276 ms
AuthControllerTest	787 ms

Рисунок В.2 – Результат виконання тестів пакету controller

Test	Time
security (com.diploma.secureapp)	1 sec 801 ms
HashUtilTest	42 ms
LoginAttemptServiceTest	9 ms
JwtAuthFilterTest	1 sec 453 ms
JwtUtilsTest	297 ms

Рисунок В.3 – Результат виконання тестів пакету security

service (com.diploma.secureapp)	1 sec 184 ms
ServerKeyServiceTest	43 ms
AuditLogServiceTest	1 sec 105 ms
EncryptionServiceTest	35 ms
TokenBlacklistServiceTest	1 ms
RateLimiterServiceTest	

Рисунок В.4 – Результат виконання тестів пакету service

exception (com.diploma.secureapp)	1 sec 194 ms
GlobalExceptionHandlerTest	1 sec 194 ms
testAccessDeniedHandled()	870 ms
testUnexpectedExceptionHandled()	64 ms
testBadCredentialsHandled()	218 ms
testValidationErrorHandled()	42 ms

Рисунок В.5 – Результат виконання тестів пакету exception

Рисунок В.6 – Спроба доступу до захищених даних за допомогою користувача

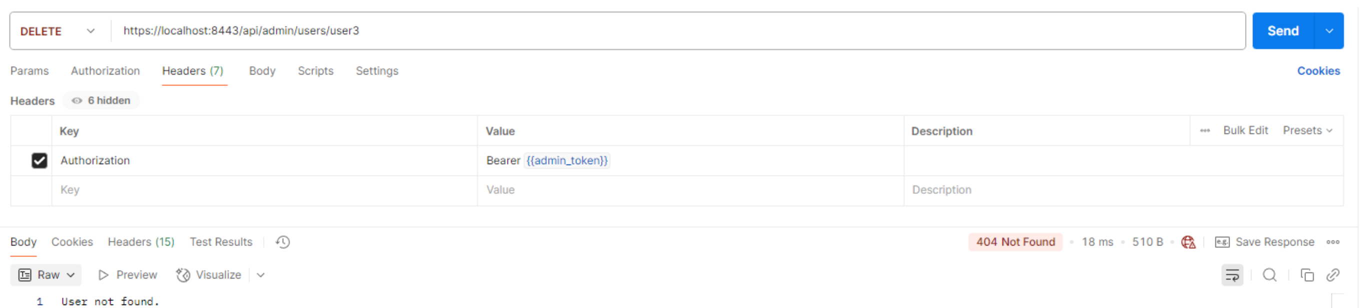

Рисунок В.7 – Спроба повторного видалення користувача

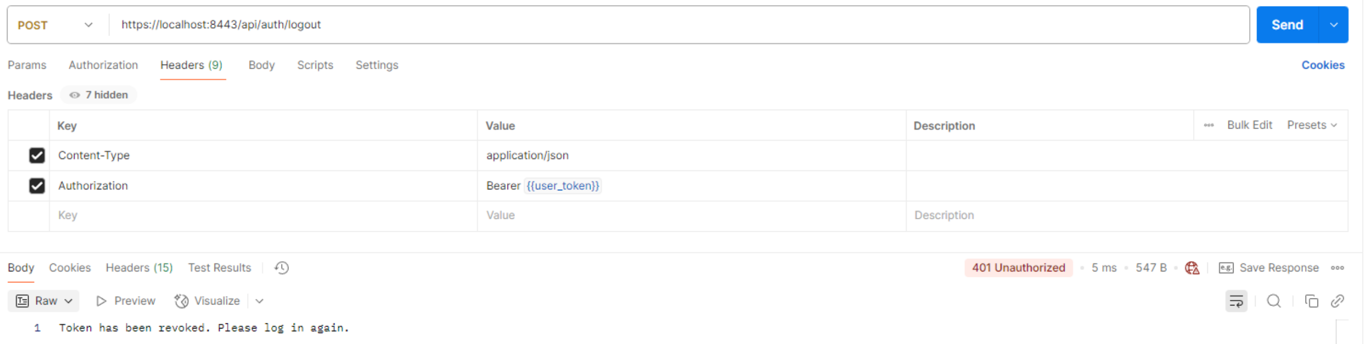

Рисунок В.8 – Спроба повторного виходу із системи

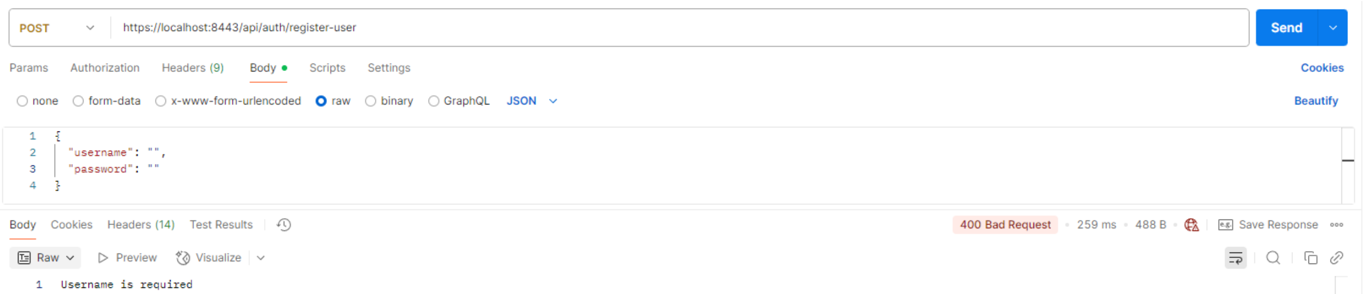

Рисунок В.9 – Спроба отримати стек помилок при некоректній реєстрації

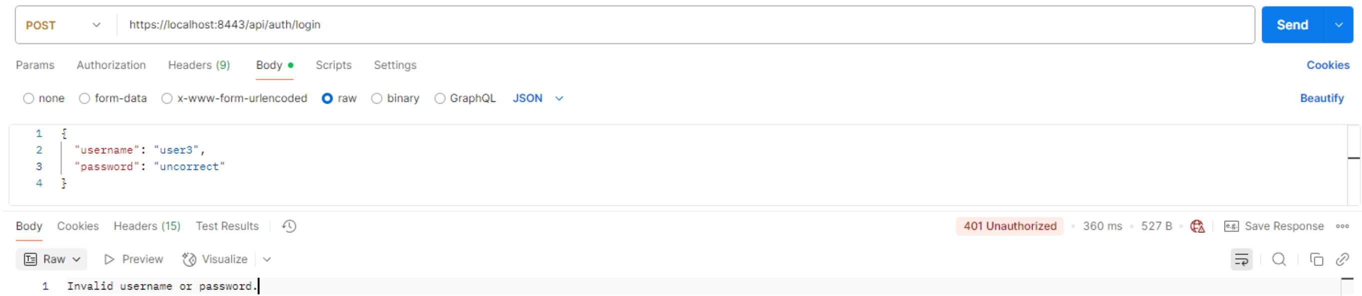

Рисунок В.10 – Спроба отримати стек помилок при некоректній авторизації

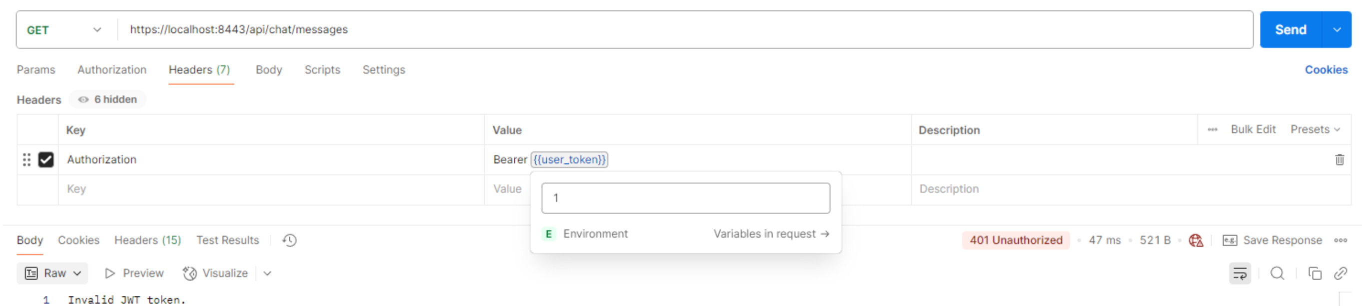

Рисунок В.11 – Спроба отримати стек помилок при некоректному токені

POST https://localhost:8443/api/auth/login Send

Params Authorization Headers (9) Body Scripts Settings Cookies

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

```
1 {
2   "username": "user3",
3   "password": "uncorrect"
4 }
```

Body Cookies Headers (15) Test Results 401 Unauthorized 360 ms 527 B Save Response

Raw Preview Visualize

```
1 Invalid username or password.
```

POST https://localhost:8443/api/auth/login Send

Params Authorization Headers (9) Body Scripts Settings Cookies

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

```
1 {
2   "username": "user3",
3   "password": "uncorrect"
4 }
```

Body Cookies Headers (15) Test Results 429 Too Many Requests 9 ms 538 B Save Response

Raw Preview Visualize

```
1 Too many attempts. Try again later.
```

Рисунок В.12 – Перевірка захисту від перебору

POST https://localhost:8443/api/chat/send Send

Params Authorization Headers (10) Body Scripts Settings Cookies

Headers 8 hidden

	Key	Value	Description
☑	Content-Type	application/json	
☐	Authorization	Bearer {{user_token}}	
	Key	Value	Description

Body Cookies Headers (15) Test Results 401 Unauthorized 29 ms 546 B Save Response

Raw Preview Visualize

```
1 Unauthorized: Please provide a valid token.
```

Рисунок В.13 – Перевірка захисту від CSRF-атаки

GET https://localhost:8443/api/admin/users Send

Params Authorization Headers (7) Body Scripts Settings Cookies

Headers 6 hidden

Key	Value	Description

Body Cookies Headers (15) Test Results 200 OK 97 ms 756 B Save Response

{} JSON Preview Visualize

```
1  [
2    {
3      "id": 2597,
4      "username": "admin",
5      "role": "ROLE_ADMIN",
6      "blocked": false,
7      "blockExpiresAt": null
8    },
9    {
10     "id": 2598,
11     "username": "user3",
12     "role": "ROLE_USER",
13     "blocked": false,
14     "blockExpiresAt": null
15   },
16   {
17     "id": 2599,
18     "username": "admin3",
19     "role": "ROLE_ADMIN",
20     "blocked": false,
21     "blockExpiresAt": null
22   }
23 ]
```

Рисунок В.14 – Отримання інформації про користувачів(адмін)

Рисунок В.15 – Отримання інформації про себе(користувач)

Рисунок В.16 – Спроба автентифікації без токена

Рисунок В.17 – Спроба автентифікації з недійсним токеном

Рисунок В.18 – Спроба переповнити буфер

	ID	ENCRYPTED_KEY	HASH	SENDER	TE
1	727	ytyoLbx3/fhk20zS4YwwiRR7mQppfMGUttxCyqHzEIRjE1lxeskfB0…	56bf59f0601c950b1b1cb6fee567d92241dcd86f560886bd7a2144…	user3	kfdJb
2	728	cAfcnbRkUKzjr382NUr+EEbmM0/0tjziC9F0li6V5SVAcAsWYLFlqK…	change209d74c173670188d1eaa67dec609f7df45ef7c81a8c4a4218d6		ZjoMF

Рисунок В.19 – Зміна хешу повідомлення

GET https://localhost:8443/api/chat/messages

Headers: Authorization — Bearer {{user_token}}

200 OK · 137 ms · 653 B

```
[
  {
    "sender": "user3",
    "text": "Hello!",
    "timestamp": "2025-05-20T14:24:05.72178"
  },
  {
    "sender": "user3",
    "text": "[CORRUPTED MESSAGE]",
    "timestamp": "2025-05-20T14:24:39.994254"
  }
]
```

Рисунок В.20 – Спроба отримати змінене повідомлення

JWT Decoder JWT Encoder

Fill in the fields below to generate a signed JWT.

HEADER: ALGORITHM & TOKEN TYPE — Valid header

```
{
  "alg": "HS512",
  "typ": "JWT"
}
```

PAYLOAD: DATA — Valid payload

```
{
  "sub": "user3",
  "role": "ROLE_USER",
  "iat": 1716200000,
  "exp": 1926200000
}
```

SIGN JWT: SECRET — Valid secret

ThisIsAReallyLongFakeSecretKeyThatLooksLegitButIsUsedForTestingOnly!1234567890ABCDE

JSON WEB TOKEN

eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMyIsInJvbGUiOiJST0xFX1VTRVIiLCJpYXQiOjE3MTYyMDAwMDAsImV4cCI6MTkyNjIwMDAwMH0.kuoSlz6qT4Q9XLf4TqGBSpLJXEhkSN856xSJY2G7ZQikjMmA_-BPknDMCkM2T9oAUSBGW2nd9HCH4QKWAHBF6Q

Рисунок В.21 – Генерація підробленого токену

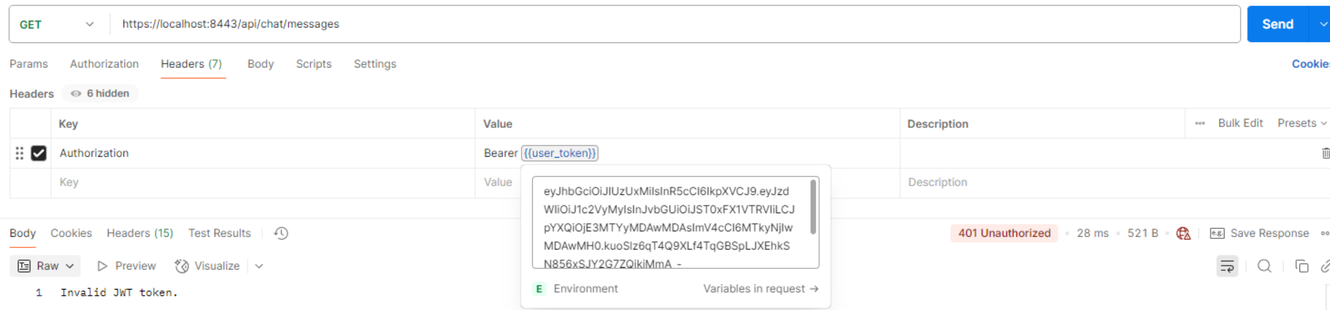

Рисунок В.22 – Спроба входу з підробленим токеном

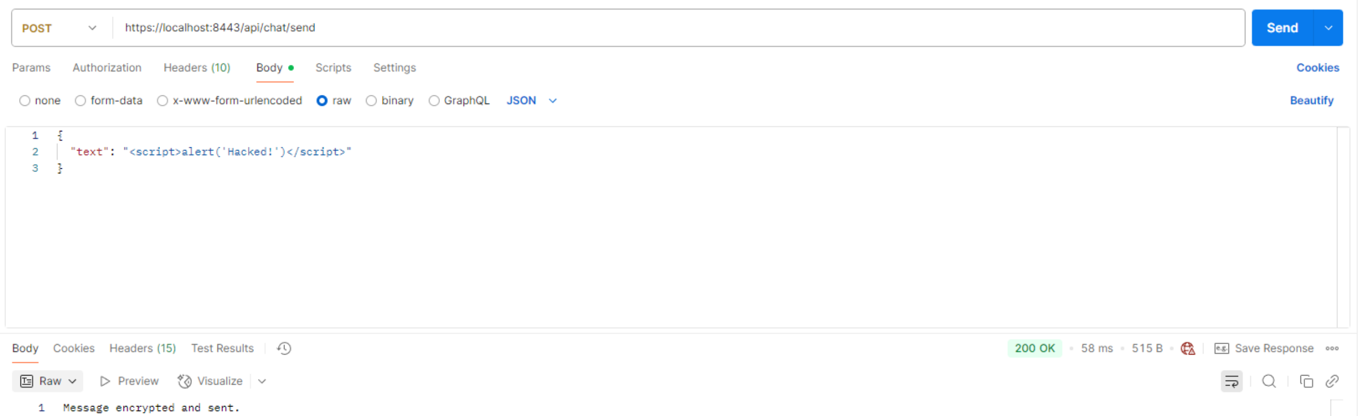

Рисунок В.23 – Введення повідомлення зі вставкою JavaScript

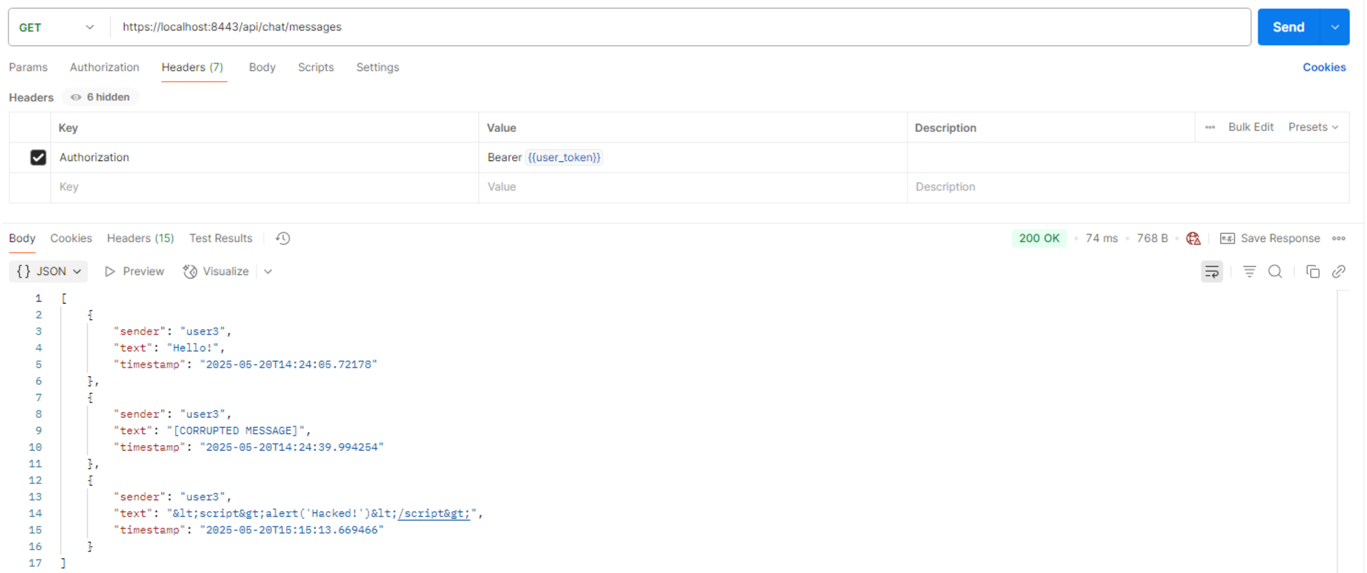

Рисунок В.24 – Виведення повідомлення після обробки: XSS не виконується

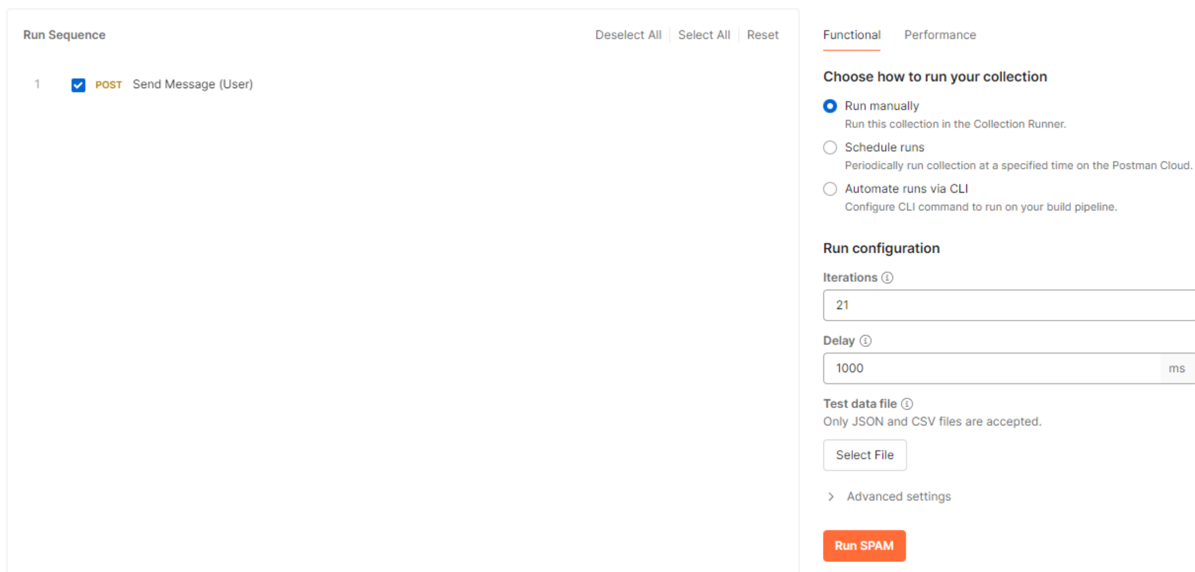

Рисунок В.25 – Налаштування спаму повідомлень

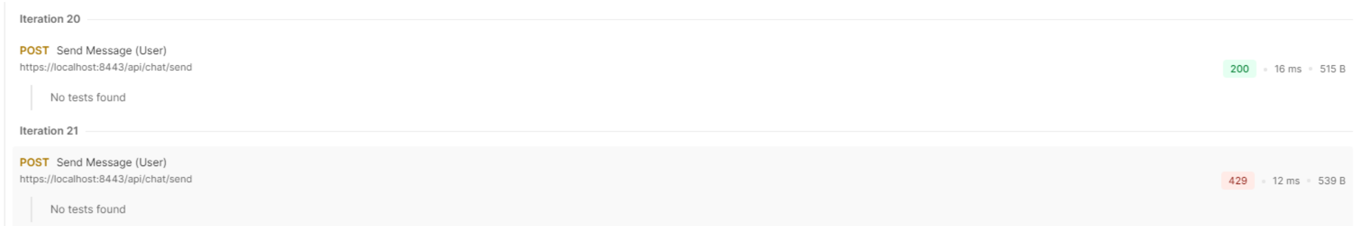

Рисунок В.26 – Результат виконання останніх двох запитів

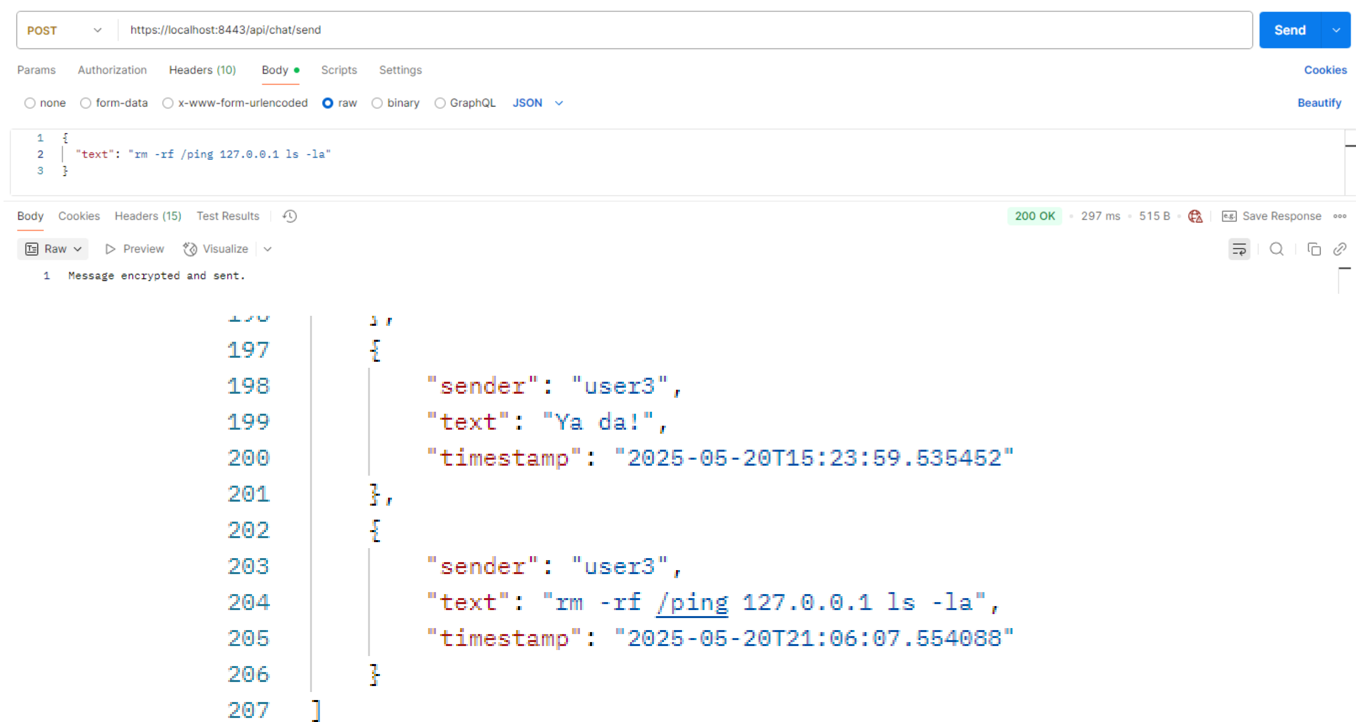

Рисунок В.27 – Результат відправлення повідомлення з shell-командою

POST https://localhost:8443/api/chat/send

Params Authorization Headers (10) Body Scripts Settings Cookies

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

```
1 {
2   "text": "../../etc/passwd"
3 }
```

Body Cookies Headers (15) Test Results 200 OK 297 ms 515 B Save Response

Raw Preview Visualize

```
1 Message encrypted and sent.
```

```
206     },
207     {
208         "sender": "user3",
209         "text": "../../etc/passwd",
210         "timestamp": "2025-05-20T21:10:34.661973"
211     }
212 ]
```

Рисунок В.28 – Результат введення повідомлення зі спробою обходу шляху

POST https://localhost:8443/api/auth/login

Params Authorization Headers (9) Body Scripts Settings Cookies

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

```
1 {
2   "username": "' OR 1=1 --",
3   "password": "user123"
4 }
```

Body Cookies Headers (15) Test Results 401 Unauthorized 150 ms 527 B Save Response

Raw Preview Visualize

```
1 Invalid username or password.
```

POST https://localhost:8443/api/auth/login

Params Authorization Headers (9) Body Scripts Settings Cookies

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

```
1 {
2   "username": "' DROP TABLE users --",
3   "password": "user123"
4 }
```

Body Cookies Headers (15) Test Results 401 Unauthorized 164 ms 527 B Save Response

Raw Preview Visualize

```
1 Invalid username or password.
```

POST https://localhost:8443/api/auth/login

Params Authorization Headers (9) Body Scripts Settings Cookies

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

```
1 {
2   "username": "admin' --",
3   "password": "user123"
4 }
```

Body Cookies Headers (15) Test Results 401 Unauthorized 98 ms 527 B Save Response

Raw Preview Visualize

```
1 Invalid username or password.
```

Рисунок В.29 – Спроби авторизації з ін’єкцією у полі username

POST https://localhost:8443/api/auth/logout Send

Params Authorization Headers (9) Body Scripts Settings Cookies

Headers 7 hidden

Key	Value	Description
Content-Type	application/json	
Authorization	Bearer {{user_token}}	
Key	Value	Description

Body Cookies Headers (15) Test Results — 200 OK · 40 ms · 506 B — Save Response

Raw Preview Visualize

```
1   Logout successful.
```

GET https://localhost:8443/api/user/profile Send

Params Authorization Headers (8) Body Scripts Settings Cookies

Headers 6 hidden

Key	Value	Description
Content-Type	application/json	
Authorization	Bearer {{user_token}}	
Key	Value	Description

Body Cookies Headers (15) Test Results — 401 Unauthorized · 5 ms · 547 B — Save Response

Raw Preview Visualize

```
1   Token has been revoked. Please log in again.
```

Рисунок В.30 – Вихід користувача та спроба доступу до захищеного ресурсу

```
Hibernate: insert into audit_log (action,timestamp,username,id) values (?,?,?,default)
2025-05-20T21:22:04.062+03:00 DEBUG 26388 --- [nio-8443-exec-8] o.s.security.web.FilterChainProxy        : Securing GET /api/user/profile
2025-05-20T21:23:49.545+03:00  INFO 26388 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for pers
2025-05-20T21:23:49.557+03:00  INFO 26388 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2025-05-20T21:23:49.561+03:00  INFO 26388 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.

Process finished with exit code 130
```

Рисунок В.31 – Завершення роботи сервера

GET https://localhost:8443/api/admin/users/user3 Send

Params Authorization Headers (8) Body Scripts Settings Cookies

Headers 6 hidden

Key	Value	Description
Content-Type	application/json	
Authorization	Bearer {{admin_token}}	
Key	Value	Description

Body Cookies Headers (15) Test Results — 401 Unauthorized · 18 ms · 546 B — Save Response

Raw Preview Visualize

```
1   Unauthorized: Please provide a valid token.
```

Рисунок В.32 – Спроба доступу до захищеного ресурсу після перезапуску

```
# ========================
# H2 DATABASE CONFIGURATION
# ========================
spring.datasource.url=jdbc:h2:file:./data/secureappdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
```

```
# ==========================
# H2 WEB CONSOLE
# ==========================
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Рисунок В.33 – Налаштування H2 бази даних

```
application.properties    EncryptionServiceTest.java
1  # ========================
2  # SERVER CONFIGURATION (HTTPS)
3  # ========================
4  server.port=8443
5  server.ssl.enabled=true
6  server.ssl.key-store=classpath:keystore.p12
7  server.ssl.key-store-password=password
8  server.ssl.key-store-type=PKCS12
```

Рисунок В.34 – Конфігурація SSL у файлі application.properties

```
POST  https://localhost:8443/api/chat/send
1 {
2   "text": "Test text"
3 }

200 OK · 81 ms · 515 B
1  Message encrypted and sent.
```

ID	ENCRYPTED_KEY	HASH	SENDER	TEXT
779	lpoB/a/DkPShMuMRKWK4zUUOBueHiCGTg2oat5T…	caef00700b723863eb7a43590ab9a791229be32…	user3	FQRmyou9yR5kDdNOlEIWP2jBCMCiKwFPDSuVSAs

Рисунок В.35 – Перевірка збереження повідомлення у зашифрованому вигляді

```json
[
    {
        "sender": "user3",
        "text": "Test text",
        "timestamp": "2025-05-20T22:22:58.785151"
    },
    {
        "sender": "user3",
        "text": "Test text",
        "timestamp": "2025-05-20T22:25:02.414845"
    }
]
```

ID	ENCRYPTED_KEY	HASH	SENDER	TEXT
779	lpoB/a/DkPShMuMRKWK4zUUOBueHiCGTg2oat5TH4og62cIcJYQ5+ji…	caef00700b723863eb7a43590ab9a791229be32a558aa069386696d…	user3	FQRmyou9yR5kDdNOlEIWP2jI
811	umw7XS9YbJ4ZBBvjbAQ+X4D7E08WyQtfF9Yh+TkeygpGWZGEDHatRHb…	aedd563acd4eca184cb5d052d397b8789d00b6500b5588d8b5c94af…	user3	0j4SerI+C8sDGycR9F/1U4j

Рисунок В.36 – Перевірка унікальності шифрування кожного повідомлення

Рисунок В.37 – Перевірка розшифрування повідомлень

Tx: Auto DDL CSV

WHERE ORDER BY

	ID	ENCRYPTE…	HASH	SENDER	TEXT	TIME
1	779	lpoB/a/nocorrect+jiYNXRBgyqJAbx5Em66qXug4XU46P/psV42CFdQmkCehcxFVl9cUnJpP				2025-05
2	811	umw7XS9YbJ4ZBBvjb…	aedd563acd4eca184cb5d052d397b8789d00b6500b5588d8b5c94af6c63968f2			

GET https://localhost:8443/api/chat/messages Send

Params Authorization Headers (7) Body Scripts Settings Cookies

Headers 6 hidden

Key	Value	Description
Authorization	Bearer {{admin_token}}	
Key	Value	Description

Body Cookies Headers (15) Test Results 200 OK 130 ms 1.47 KB Save Response


```
[
    {
        "id": 779,
        "sender": "user3",
        "text": "[DECRYPTION ERROR]",
        "hash": "caef00700b723863eb7a43590ab9a791229be32a558aa069386696d542cd4e25",
        "timestamp": "2025-05-20T22:22:58.785151"
    },
    {
        "id": 811,
        "sender": "user3",
        "text": "Test text",
        "hash": "aedd563acd4eca184cb5d052d397b8789d00b6500b5588d8b5c94af6c63968f2",
        "timestamp": "2025-05-20T22:25:02.414845"
    }
]
```


Рисунок В.38 – Результат спотворення ключа

application.properties | server-private.key × | server-public.key | Encrypti

1 MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQC+i/jLKAS2aQzZGa9JnF2

Body Cookies Headers (15) Test Results | 200 OK · 64 ms · 1.48 KB · Save Response

{} JSON ▷ Preview Visualize


```
[
    {
        "id": 779,
        "sender": "user3",
        "text": "[DECRYPTION ERROR]",
        "encryptedKey": "lpoB/a/nocorrect+jiYNXRBgyqJAbx5Em66qXug4XU46P/psV42CFdQmkCehcxFVl9cUnJpPCbcswI95WkWORCWuVbL+4Dtz3/cA0sa4QfmtLasKofmEmeJDfZT4bbXaXPoxXzSjcVrbAZ3G1GS7nZZvUF30BEJC0IOWVAqrYaVoXhyv/S48UxYbyDXE/2pE0nBTvuLm6sU88e05g505/2lbMwINNay6BV6OfPO3I3Ts+0mJOWeli5AnX5GuEhWg1Z6HDhgv/Z9VDlnPLvNJU+ieWO4WrWWIrxlzF2t2N+TDwP2rw==",
        "hash": "caef00700b723863eb7a43590ab9a791229be32a558aa069386696d542cd4e25",
        "timestamp": "2025-05-20T22:22:58.785151"
    },
    {
        "id": 811,
        "sender": "user3",
        "text": "[DECRYPTION ERROR]",
        "encryptedKey": "umw7XS9YbJ4ZBBvjbAQ+X4D7EO8WyQtfF9Yh+TkeygpGWZGEDHatRHbW8jbWSneQGaBzjjGkwCv5y1yYpdCbiYthjHOSZM/hjwFAst8ilkbVT1JgfqC9oLvJjxJnrSU11Hph9nYeVzJGMFqQqiSZMmxJj9mZ4lOWMgVnmMBCHsuToTEdrEoSM1HYTGTf23E5q8RkcuQfsnhav+QUKI5qtgWTHSq5A4yuAo7VU3U1MPPAxb/YwS+7nBUsq8sAXGA9wjvEiLcS37h4VYCGKMsoxJVnTVQciEHHWpucUw1eG2XopDOdD3nNnLYhnksl+ooQGrrMr+aFVQ1dAE51kE5KzA==",
        "hash": "aedd563acd4eca184cb5d052d397b8789d00b6500b5588d8b5c94af6c63968f2",
        "timestamp": "2025-05-20T22:25:02.414845"
    }
]
```


Рисунок В.39 – Зміна приватного ключа та спроба розшифрувати повідомлення