**Міністерство освіти і науки України**
**Київський національний університет імені Тараса Шевченка**

**Факультет інформаційних технологій**
**Кафедра кібербезпеки та захисту інформації**

**ДОПУСТИТИ ДО ЗАХИСТУ:**
В.о. завідувача кафедри
кібербезпеки та захисту
інформації
________ Іван ПАРХОМЕНКО
«___» червня 2025 р.

# ПОЯСНЮВАЛЬНА ЗАПИСКА
## кваліфікаційної роботи

**галузь знань** 12 Інформаційні технології
(шифр і назва галузі знань)
**спеціальність** 125 Кібербезпека
(код і назва спеціальності)
**освітній ступень** бакалавр
**освітня програма** Кібербезпека
(назва освітньо-професійної програми)
**на тему:** «Комплексний метод захисту вебчату»

**Виконавець:** студент IV курсу, групи КБ-44 (мс)

_______________ **Руслан РИМАРЕНКО**
(підпис) (ім’я, прізвище)

| | Підпис | Ім’я, прізвище |
|---|---|---|
| **Керівник** | | Олександр ЛАПТЄВ |

| | | |
|---|---|---|
| **Нормоконтроль** | | Сергій ДАКОВ |

**Київ 2025**

**Міністерство освіти і науки України**
**Київський національний університет імені Тараса Шевченка**

**Факультет інформаційних технологій**
**Кафедра кібербезпеки та захисту інформації**

**ЗАТВЕРДЖЕНО:**
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
________ Іван ПАРХОМЕНКО
«29» листопада 2024 р.

# ЗАВДАННЯ

**на виконання кваліфікаційної роботи**

**спеціальності** 125 Кібербезпека
(код і назва спеціальності)

**освітньої програми** Кібербезпека
(назва освітньо-професійної програми)

**Студенту** **КБ-44 (мс)** **Римаренку Руслану Миколайовичу**
(група) (прізвище ім'я по батькові)

**Тема кваліфікаційної роботи** Комплексний метод захисту вебчату

**1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ**

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №6 від 28.11.2024 р.

**2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ**

Архітектура вебдодатків, захист API, React, Django, PostgreSQL

**3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ**

Необхідно дослідити архітектуру вебдодатків, принципи роботи вебчатів, типові вразливості при передачі даних, методи захисту API, засоби валідації та фільтрації, механізми запобігання ін'єкціям, розробити рекомендації при роботі з вебчатами.

**4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ**

**Практична цінність** Розроблений вебдодаток вебчату та рекомендації,

щодо його захисту.

**5. ДАТА ВИДАЧІ ЗАВДАННЯ**

Дата видачі завдання: 29 листопада 2024 року

Завдання видав ______________ Олександр ЛАПТЄВ
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання ______________ Руслан РИМАРЕНКО
(підпис) (ім'я, прізвище)

**КАЛЕНДАРНИЙ ПЛАН**

| **№ п/п** | **Найменування етапів робіт** | **Строки виконання робіт (початок-кінець)** | **Відмітка про виконання** |
|---|---|---|---|
| 1 | Уточнення постановки задачі | 29.11.2024 – 22.01.2025 | *виконано* |
| 2 | Аналіз літератури | 29.01.2025 – 11.02.2025 | *виконано* |
| 3 | Обґрунтування вибору рішення | 12.02.2025 – 15.02.2025 | *виконано* |
| 4 | Дослідження загроз безпеці вебчату | 16.02.2025 – 04.03.2025 | *виконано* |
| 5 | Аналіз механізмів та методів захисту вебчату | 05.03.2025 – 21.03.2025 | *виконано* |
| 6 | Реалізація комплексного методу захисту вебчату | 22.03.2025 – 08.04.2025 | *виконано* |
| 7 | Розробка рекомендацій, що до захисту вебчату | 09.04.2025 – 10.05.2025 | *виконано* |
| 8 | Оформлення пояснювальної записки | 11.05.2025 – 27.05.2025 | *виконано* |
| 9 | Підготовка до захисту кваліфікаційної роботи | 28.05.2025 – 13.06.2025 | *виконано* |

Завдання видав ______________ Олександр ЛАПТЄВ
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання ______________ Руслан РИМАРЕНКО
(підпис) (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 13 червня 2025 року

# РЕФЕРАТ


Пояснювальна записка кваліфікаційної роботи складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Основний текст займає 67 сторінок, включає в себе зміст, вступ, три розділи кваліфікаційної роботи, висновки та список джерел. Крім того, робота містить 2 додатки із загальною кількістю сторінок 6. У пояснювальній записці кваліфікаційної роботи міститься 10 рисунків та 1 таблиця.

*Мета дослідження*: підвищення захищеності вебчатів за рахунок розробки веб додатку та видача рекомендацій з їх захисту.

Для досягнення зазначеної мети поставлено наступні завдання:

- аналіз методів та методик захисту вебчатів;
- дослідження процесів захисту вебчатів;
- розробка рекомендацій щодо підвищення захисту вебчатів.

*Об'єктом дослідження* є процес захисту вебчатів.

*Предметом дослідження* методи захисту вебчатів.

*Практичною цінністю отриманих результатів* є програмна реалізація засобів та механізмів захисту вебдодатків.



*Ключові слова:* ВЕБЧАТ, ЗАХИСТ ІНФОРМАЦІЇ, АУТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ, SQL-ІН'ЄКЦІЇ, XSS, CSRF, BOT-ЗАХИСТ, TLS, МОНІТОРИНГ БЕЗПЕКИ.

# ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 7
ВСТУП 8
РОЗДІЛ 1. ДОСЛІДЖЕННЯ ЗАГРОЗ БЕЗПЕЦІ ОБ'ЄКТІВ ВЕБЧАТУ 9
1.1 Визначення та основні характеристики вебчату 9
1.2 Функціональні модулі та інтерфейси 11
1.3 Інформаційні активи 14
1.4 Технологічне середовище та інфраструктура 15
1.5 Користувачі та ролі доступу 17
1.6 Потенційні загрози безпеці та вразливості 19
Висновки за розділом 1: 21
РОЗДІЛ 2. АНАЛІЗ МЕХАНІЗМІВ ТА МЕТОДІВ ЗАХИСТУ ВЕБЧАТУ 23
2.1. Основні принципи побудови системи захисту інформації для вебчату 23
2.2 Аутентифікація, авторизація та управління сесіями 24
2.3 Захист каналів зв'язку та передавання даних 26
2.4 Захист від SQL-ін'єкцій та XSS-атак, валідація та фільтрація 27
2.5 Обмеження доступу до API 29
2.6 Захист від ботів, спаму та автоматизованих загроз 32
2.7 Моніторинг, аудит та виявлення інцидентів інформаційної безпеки 33
Висновки за розділом 2: 35
РОЗДІЛ 3. РЕАЛІЗАЦІЯ КОМПЛЕКСНОГО МЕТОДУ ЗАХИСТУ ВЕБЧАТУ 37
3.1 Вибір стеку інструментів для розробки вебдодатку 37
3.2 Реалізація функціоналу вебчату як окремого модуля 40
3.3 Впровадження методів захисту у процеси обміну даними 44
3.4 Застосування валідації, фільтрації та захисту від ін'єкцій 49
3.5 Реалізація захисту від автоматизованих загроз 52
3.6 Тестування застосованих засобів безпеки 55
3.7 Порівняння з існуючими аналогами 57

Висновки за розділом 3: 65

ВИСНОВКИ 67

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 69

Додаток А 73

# ПЕРЕЛІК СКОРОЧЕНЬ

| | | |
|---|---|---|
| CRM | – | Customer Relationship Management, система управління взаємовідносинами з клієнтами |
| XSS | – | Cross-Site Scripting, міжсайтове скриптування |
| HTTPS | – | HyperText Transfer Protocol Secure – захищений протокол передачі гіпертексту |
| UI | – | User Interface, користувацький інтерфейс |
| CSRF | – | Cross-Site Request Forgery, міжсайтова підробка запиту |
| SQL | – | Structured Query Language, структурована мова запитів |
| TLS | – | Transport Layer Security, протокол безпеки транспортного рівня |
| MITM | – | Man-In-The-Middle, атака «людина посередині» |
| WSS | – | WebSocket Secure, захищений протокол WebSocket |
| API | – | Application Programming Interface, інтерфейс програмування застосунків |
| WSGI | – | Web Server Gateway Interface, інтерфейс взаємодії вебсервера і Python-додатку |
| REST | – | Representational State Transfer, архітектурний стиль передачі стану представлень |
| RBAC | – | Role-Based Access Control, керування доступом на основі ролей |
| JWT | – | JSON Web Token, вебтокен у форматі JSON |
| ORM | – | Object-Relational Mapping, об’єктно-реляційне відображення |
| WAF | – | Web Application Firewall, фаєрвол вебдодатків |

# ВСТУП

У сучасних умовах стрімкого розвитку онлайн-комунікацій вебчати стали одним із найпоширеніших засобів оперативного обміну інформацією між користувачами та сервісами. Їх застосовують у сферах підтримки клієнтів, внутрішнього спілкування, електронної комерції та багатьох інших бізнес-процесах. Водночас саме вебчати є слабкою ланкою системи для реалізації кіберзагроз: від SQL-ін'єкцій і XSS-атак, націлених на компрометацію системи, до DDoS-атак, які зупинять її функціонування. Ці загрози можуть призводити не лише до витоку персональної інформації, а й до втрати довіри користувачів, порушення цілісності обміну повідомленнями та зниження доступності сервісу.

**Мета дослідження** полягає у підвищенні захищеності вебчатів шляхом розробки спеціалізованого вебдодатку та формулювання практичних рекомендацій щодо їхнього захисту від актуальних кіберзагроз.

**Завдання роботи:** аналіз існуючих методів та методик захисту вебчатів, дослідження процесів їхньої безпеки, а також розробка рекомендацій для підвищення стійкості системи до атак. У ході дослідження буде проведено аналіз архітектурних підходів до реалізації захищених чат-модулів, виявлено слабкі місця у механізмах обміну повідомленнями та запропоновано комплекс рекомендацій, щодо підвищення захищеності вебчатів.

**Об'єктом дослідження** є процеси захисту вебчатів.

**Предметом дослідження** виступають методи захисту вебчатів.

**Актуальність** роботи полягає в тому, що відпрацьовані методи можуть бути інтегровані в існуючі платформи клієнт-серверної взаємодії, забезпечивши підвищену стійкість до SQL-ін'єкцій, XSS-атак, підробки запитів та атак здатних викликати відмову в обслуговуванні.

# РОЗДІЛ 1. ДОСЛІДЖЕННЯ ЗАГРОЗ БЕЗПЕЦІ ОБ’ЄКТІВ ВЕБЧАТУ

## 1.1 Визначення та основні характеристики вебчату

Під вебчатом у рамках даної роботи розуміється спеціалізований модуль, що інтегрується в існуючий вебсайт і надає можливість оперативної взаємодії користувачів з технічною підтримкою або іншими сервісними підрозділами організації. Така інтеграція не потребує суттєвих змін у структурі основного ресурсу: чат вбудовується як окремий компонент інтерфейсу, що ініціює двосторонній канал зв’язку між браузером користувача та серверною частиною системи. З технічної точки зору цей канал зазвичай базується на протоколі WebSocket, який після початкового встановлення з’єднання через HTTPS підтримує постійний, довготривалий зв’язок із мінімальними затримками. Завдяки цьому оператори служби підтримки фактично знаходяться «поруч» із клієнтом, що дозволяє надавати консультації в режимі реального часу без потреби повторних завантажень сторінок або чекання відповіді від сервера.

Інтерфейс вебчату, як правило, реалізований у вигляді компактного вікна або панелі, яка може бути вбудована в будь-який розділ сайту: від сторінки з документацією до інтерфейсу оформлення замовлень. Користувачі, які вводять свої запитання або повідомлення, відразу бачать індикатор «з’єднання встановлено» та статус оператора («онлайн» або «офлайн»), а також підтвердження доставки і прочитання повідомлень. У розмові з оператором можливе не лише обмін текстовими повідомленнями, а й пересилання файлів - наприклад, скріншотів помилок чи логів - що значно пришвидшує діагностику та вирішення технічних проблем. Усе це створює відчуття безперервного, живого діалогу, навіть якщо обидві сторони знаходяться в різних часових зонах або використовують різні пристрої.

З погляду архітектури сервісу, вебчат як модуль передбачає наявність кількох складових: компонент клієнтської частини, що відповідає за відображення інтерфейсу та обробку подій введення; серверна логіка, що керує чергами запитів, маршрутизує повідомлення до вільного оператора та забезпечує механізми збереження історії чат-сесій; а також проміжні сервіси, які займаються обробкою великих обсягів одночасних з’єднань, балансуванням навантаження та кешуванням статичних ресурсів. Така модульність дозволяє масштабувати чат, не впливаючи на основну платформу сайту, і розгортати його в окремих контейнерах для підвищення відмовостійкості.

Ключовою перевагою вебчату є можливість зберігати історію взаємодії: від перших повідомлень до остаточного закриття сесії. Ці дані використовуються не лише для аналізу і аудиту якості обслуговування, а й для автоматизованого навчання чат-ботів або побудови звітів про типові запити. При цьому серверна частина забезпечує розмежування доступу: лише уповноважені оператори мають змогу переглядати та відповідати на приватні повідомлення, тоді як адміністрація сайту може аналізувати узагальнені метрики без доступу до персональних деталей окремих розмов. Такий підхід дозволяє зберігати баланс між прозорістю процесу обслуговування та конфіденційністю інформації користувачів. На рисунку 1.1 зображено приклад реалізації вебчату.

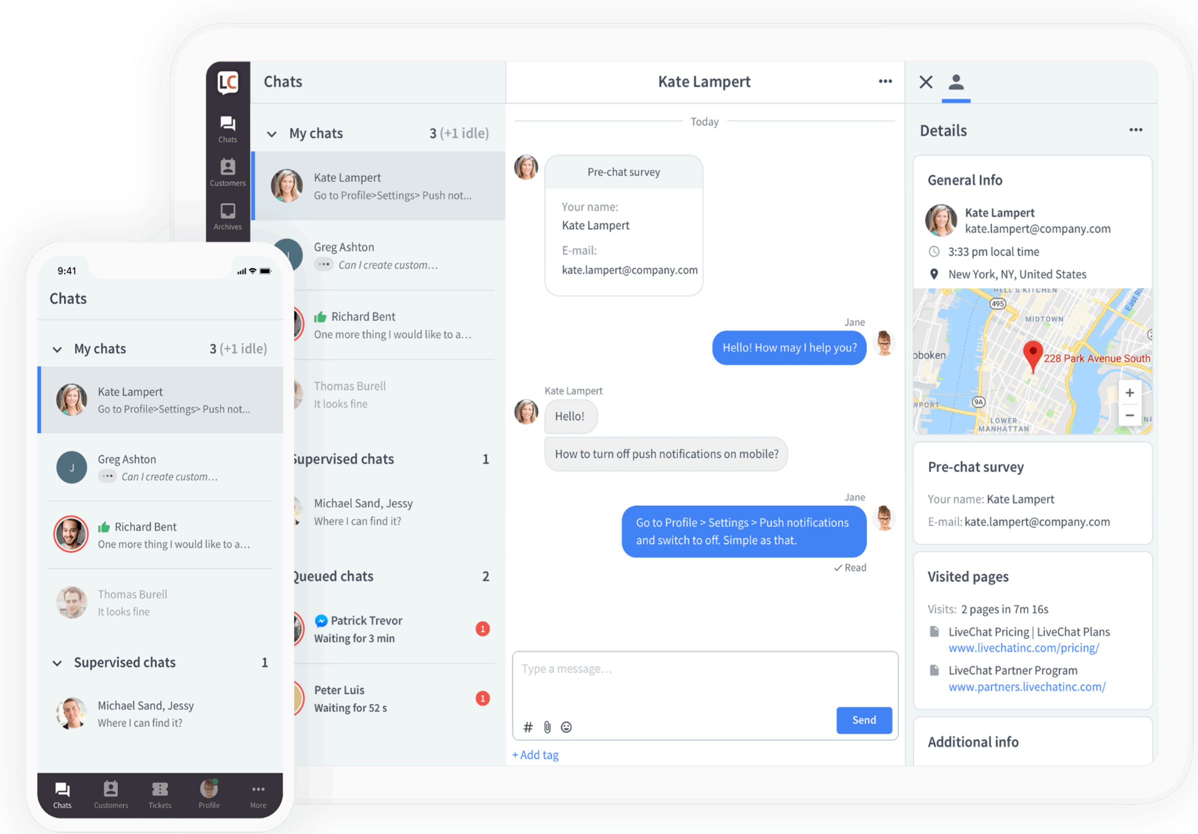


Рисунок 1.1. – Приклад реалізації вебчату

## 1.2 Функціональні модулі та інтерфейси

Функціональні модулі та інтерфейси системи Під функціональними модулями вебчату розуміють набір окремих компонентів, кожен із яких відповідає за певний етап обробки користувацьких повідомлень та взаємодії з системою. Першим зазвичай ініціюється модуль клієнтського інтерфейсу, вбудований у сторінку сайту у вигляді компактного вікна або плаваючої панелі. Цей компонент відстежує дії користувача: відкриття чат-вікна, введення тексту, вибір вкладених файлів і медіа, а також ініціює встановлення двостороннього з’єднання із сервером за допомогою WebSocket. Він відповідає за відображення вхідних повідомлень у тому порядку, в якому їх надсилають оператори, і

синхронізує локальний буфер із серверною чергою, щоб навіть після оновлення сторінки або розриву з'єднання користувач бачив історію переписки.

На серверному боці ключовим модулем є ядро обробки повідомлень, що забезпечує маршрутизацію тексту й файлів між учасниками сеансу. Воно приймає повідомлення від одного клієнта, формує відповідний запис у базі даних та передає його іншим підключеним користувачам або операторам черги. У цій частині також реалізовано логіку чергування: якщо всі оператори зайняті, модуль ставить звернення в очікування й інформує користувача про орієнтовний час відповіді. Крім того, саме тут відбувається інтеграція з CRM або системами тикетування, коли чат-запит конвертується у внутрішню заявку для подальшої обробки поза межами онлайн-сеансу.

Модуль збереження історії відповідає за довготривале зберігання всіх повідомлень, метаданих сесій та файлів, що пересилаються. При кожному новому зверненні сервер створює запис у реляційній або документній базі даних з часовими мітками, ідентифікаторами учасників та іншими атрибутами. Це дає змогу не лише відновити діалог після розриву з'єднання, а й проводити статистичний аналіз тематики звернень, оцінювати навантаження на операторів та будувати звіти про якість обслуговування. Усі дані шифруються перед збереженням, щоб виключити ризик компрометації у випадку несанкціонованого доступу до сховища.

Не менш важливим є модуль сповіщень, який формує push-повідомлення для клієнта та внутрішні оповіщення для операторів. Він стежить за зміною стану сесії - наприклад, коли оператор відкриває діалог або надсилає відповідь - і негайно передає ці події користувачу через WebSocket або через браузерні push-сервіси. Завдяки цьому користувачі отримують миттєве повідомлення про новий вхідний текст, навіть якщо чат-вікно було тимчасово сховано або користувач перейшов на іншу вкладку.

Інтерфейси для інтеграції формують зв'язок внутрішніх модулів із зовнішніми системами. REST- та WebSocket-API надають методи для створення нових сесій, відправки та отримання повідомлень, керування статусом

операторів та отримання аналітики в режимі реального часу. Крім того, вебчат експонує webhook-точки для сторонніх сервісів, що дозволяє автоматично передавати нові звернення у CRM або запускати бізнес-логіку у залежності від контенту повідомлення, наприклад, відправку SMS-сповіщень чи запуск сценарію ескалації. На рисунку 1.2 зображена загальна архітектурна схема системи вебчату.

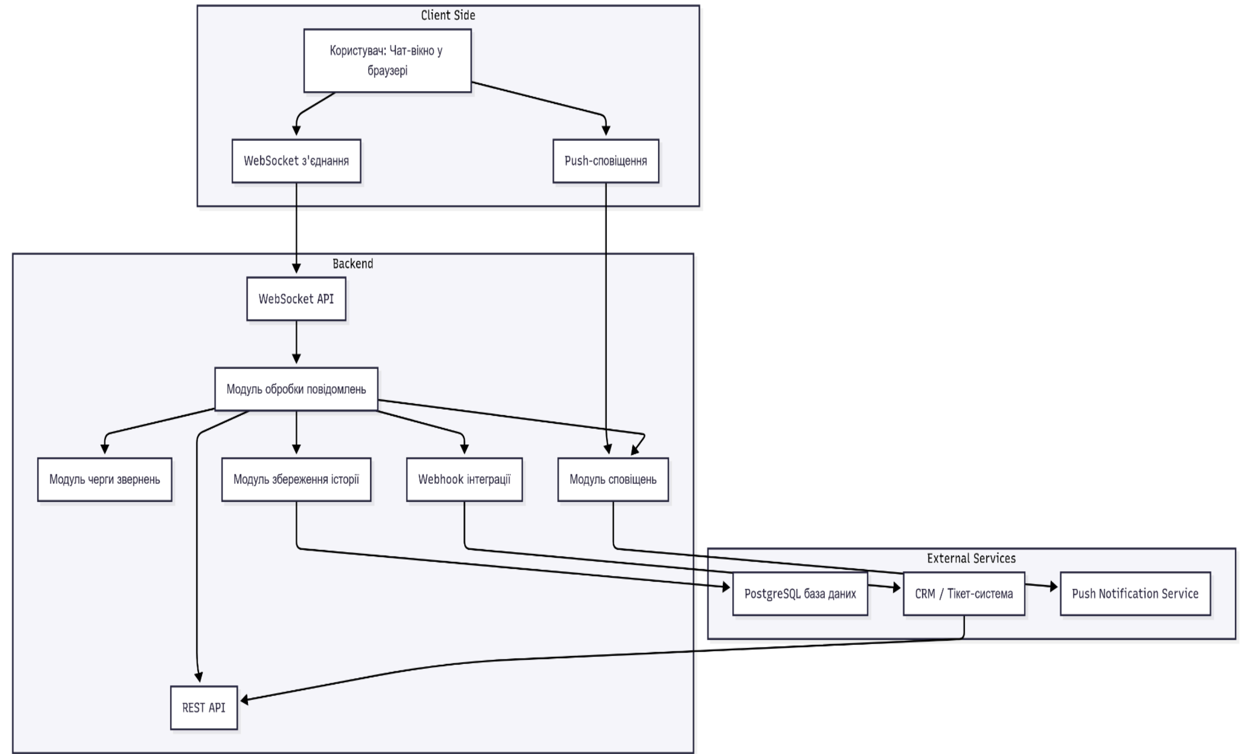


Рисунок 1.2. – Структурна діаграма системи вебчату

Загалом, кожен із цих модулів взаємодіє через чіткі та документовані інтерфейси, що забезпечують масштабованість і майбутнє розширення системи. Ключовим у їхній взаємодії є розділення обов’язків: клієнтський код відповідає лише за презентацію та ініціацію дій, серверна логіка - за маршрутизацію та збереження, а проміжні сервіси - за балансування й доступність. Така архітектурна схема гарантує, що розгортання нового функціоналу або оновлення компонентів не порушить цілісності загальної системи й знизить ризик появи критичних уразливостей.

### 1.3 Інформаційні активи

Інформаційні активи становлять ядро будь-якої інформаційної системи, зокрема вебзастосунків. Ці активи не обмежуються лише особистими даними користувачів, а включають також структуровану та не структуровану інформацію, що передається, обробляється або зберігається в межах цифрової платформи.

Особливе значення в системах вебчатів мають дані користувачів, що надходять до системи через інтерфейси введення. Це можуть бути імена, електронні адреси, номери телефонів, текстові повідомлення, файли вкладень, IP-адреси, а також технічна інформація про пристрої та сесії користувачів. Порушення конфіденційності цих активів може спричинити наслідки, пов’язані з витоком персональної інформації та репутаційними втратами для організації.

Окрім персональних даних, до інформаційних активів належать програмні конфігурації, налаштування доступу, службові ключі, токени авторизації, сесійні ідентифікатори, лог-файли, бази даних, облікові записи користувачів та адміністраторів, а також сам код вебзастосунку. Код і його компоненти, зокрема скрипти, шаблони, бібліотеки, API-інтерфейси, а також логіка бізнес-процесів, що реалізується на серверній і клієнтській стороні, також є критичними активами. Їх компрометація або модифікація в результаті несанкціонованого втручання може призвести до збоїв у роботі системи, втрати цілісності, зміни поведінки вебдодатку або відкриття нових векторів атак.

Також важливим активом є метаінформація про функціонування системи, включаючи журнали активності, телеметрію, моніторингові дані, технічні характеристики інфраструктури, конфігурації безпеки тощо. Вона може містити важливі відомості, які в разі витоку або знищення ускладнюють аналіз інцидентів безпеки, унеможливлюють аудит або спричиняють дезорієнтацію під час реагування на інциденти інформаційної безпеки.

Критичність інформаційних активів визначається не лише їхнім змістом або технічною реалізацією, а й контекстом використання, залежністю

бізнес-процесів від наявності та доступності даних, а також потенційними наслідками у разі втрати, компрометації або порушення доступу до них. Відповідно, класифікація активів за ступенем критичності є необхідною передумовою для побудови ефективної системи захисту, що дозволяє розподіляти ресурси на безпеку пропорційно до значущості кожного елементу інформаційної екосистеми.

## 1.4 Технологічне середовище та інфраструктура

Технологічне середовище системи базується на взаємодії кількох ключових мережевих протоколів, що забезпечують як захищеність, так і високу продуктивність передачі даних. Усі запити від клієнтської частини до серверного оточення здійснюються через HTTPS‑канал із застосуванням TLS‑шифрування, що гарантує конфіденційність обміну інформацією та захист від перехоплення або модифікації трафіку. Для забезпечення режиму реального часу використовується механізм оновлення з'єднання до WebSocket, який після початкового рукостискання по TLS дозволяє підтримувати довготривале двостороннє з'єднання без надлишкових заголовків HTTP. При взаємодії модулів системи та зовнішніх сервісів дотримується REST‑парадигма, що спирається на структуровані HTTP‑методи та уніфіковані формати запитів і відповідей, що значно спрощує масштабування й документування API.

Серверна частина реалізована на стеку Python із застосуванням Django, що надає строгий розподіл зон відповідальності між моделями, контролерами й шаблонами, а також містить усталені механізми захисту від поширених веб‑атак. Обробка HTTP‑запитів ведеться через високопродуктивний WSGI‑сервер, який керує пулом робочих процесів, автоматично налаштовуючи їх кількість відповідно до навантаження та забезпечуючи відновлення сервісу у разі збоїв окремих воркерів. Передній край інфраструктури представлений Nginx, що виконує термінацію SSL, кешування статичних ресурсів і розподіл

вхідного трафіку між екземплярами бекенд-додатку для підтримки необхідного рівня доступності та відмовостійкості.

Контейнеризація застосовується як основний підхід до розгортання компонентів, коли кожен сервіс поміщається в ізольований контейнер із чітко зконфігурованим оточенням та залежностями. Оркестрація таких контейнерів дозволяє автоматично контролювати життєвий цикл мікросервісів, здійснювати самовідновлення при виникненні збоїв та горизонтально масштабувати компоненти відповідно до фактичного навантаження. Опис інфраструктури в декларативних маніфестах гарантує ідентичність середовищ розробки, тестування і продакшену, усуваючи проблему розбіжності конфігурацій.

Для баз даних та інших станозалежних сервісів використовуються віртуальні машини, що розгортаються у публічній хмарі. Обраний підхід дозволяє організувати сегментовані мережі з чітким розділенням інтерфейсної зони та внутрішніх зон із обмеженим доступом, а також налаштовувати політики резервного копіювання й відновлення даних. Хмарна платформа забезпечує можливості автоматичного масштабування за ключовими метриками (процесорні ресурси, пам'ять, кількість з'єднань) та модель оплати за фактичне споживання, що дозволяє оптимізувати витрати без зниження показників надійності.

Усі компоненти оточення інтегровані з централізованою системою моніторингу й логування, яка акумулює метрики продуктивності, журнали доступу й системні сповіщення в єдиному репозиторії. Це дає змогу здійснювати активне виявлення аномалій, надсилати тривожні сигнали відповідальним фахівцям і проводити ретельний аудит інцидентів. Завдяки такому комплексному підходу до побудови технологічного середовища та інфраструктури забезпечується стійкість, масштабованість і безперервність роботи.

### 1.5 Користувачі та ролі доступу

Кожен користувач проходить процедуру автентифікації, у ході якої визначається його роль у системі. Анонімний відвідувач має доступ до мінімального набору функцій, як початок нового чат-сеансу, при цьому система збирає лише базову інформацію про сесію (IP-адресу, час початку діалогу) для подальшого моніторингу. Після реєстрації користувач отримує розширені можливості: йому доступні історія попередніх звернень та збереження незавершених чат-повідомлень.

Роль оператора передбачає вміння працювати з великою кількістю одночасних звернень. Після входу до системи оператор бачить чергу нових запитів та може призначати пріоритет їх обробки. При цьому йому відкривається доступ до функцій редагування статусу звернення та відправлення шаблонних відповідей, проте обмежено право перегляду особистих даних користувача до тих, що необхідні для поточного діалогу. У цьому разі застосовується принцип мінімальних привілеїв: оператор не може змінювати налаштування системи, переглядати зашифровані журнали або отримувати доступ до ключів шифрування повідомлень.

Адміністратор системи володіє повним набором прав для конфігурації компонентів платформи: він може створювати і видаляти облікові записи, призначати ролі, керувати політиками безпеки та здійснювати аудит дій інших користувачів. Через спеціальний панельний інтерфейс адміністратор коригує налаштування TLS-сертифікатів, переглядає метрики продуктивності, встановлює правила обробки чат-сесій, а також контролює інтеграцію з CRM-системою та зовнішніми API. Ця роль передбачає доступ не лише до користувацького контенту, але й до всіх службових журналів і конфігураційних файлів, тому до адміністраторських облікових записів висуваються підвищені вимоги щодо складності паролів і використання двофакторної автентифікації.

Розмежування функціональних можливостей між ролями забезпечується через систему контролю доступу на основі ролей (RBAC), де кожна роль має

чітко визначені дозволи на виконання операцій із різними типами ресурсів.[1] При кожній спробі виконання критичних дій система перевіряє, чи належить користувач до відповідної групи, і, в разі невідповідності, блокує запит із повідомленням про відсутність прав. Такий підхід гарантує, що навіть у разі компрометації облікового запису ризик інцидентів залишиться мінімальним, оскільки кожен користувач володіє лише тим обсягом повноважень, який необхідний для виконання його безпосередніх завдань. Механізм автентифікації зображено на рисунку 1.3.

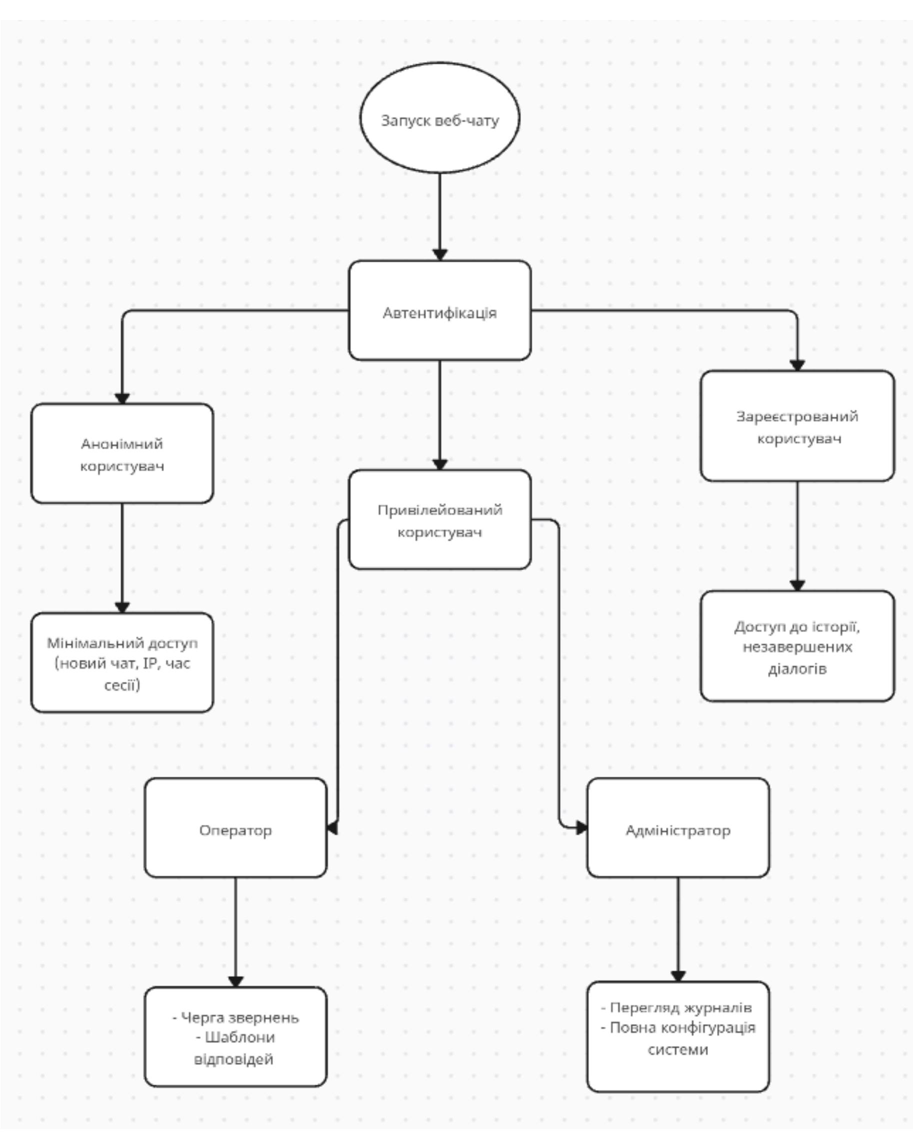


Рисунок 1.3. – Загальна блок-схема автентифікації

### 1.6 Потенційні загрози безпеці та вразливості

Потенційні загрози безпеці та вразливості є фундаментальним аспектом, що визначає ризиковий профіль вебзастосунків [2]. Такі системи, за своєю природою, мають відкриті точки взаємодії з користувачем, що створює численні можливості для експлуатації слабких місць як у клієнтській, так і в серверній частині. Розуміння повного спектру можливих загроз та виявлення вразливих місць є обов'язковою передумовою для побудови надійної системи захисту інформації, особливо в контексті обробки персональних та конфіденційних даних.

Основною загрозою для таких систем є несанкціонований доступ до інформації, який може бути досягнутий через технічні або логічні вразливості у коді, неправильні конфігурації системи чи недостатній контроль за автентифікацією. Зокрема, уразливості в механізмах авторизації можуть дозволити зловмисникам здійснити вертикальне або горизонтальне підвищення привілеїв, отримавши доступ до інформації інших користувачів або адміністративних функцій системи. В умовах сучасного середовища загрози можуть реалізовуватись як із боку автоматизованих ботів, так і в рамках цільових атак, спрямованих на конкретну організацію чи сервіс.

Значну небезпеку становить недостатній рівень валідації введених користувачем даних, що може призвести до ін'єкційного впливу на систему. Це включає SQL-ін'єкції, за допомогою яких зловмисники можуть отримати доступ до бази даних, а також XSS-атаки, що дозволяють впроваджувати та виконувати шкідливий скрипт у браузерах інших користувачів. Особливо вразливою в цьому сенсі є система чату, де повідомлення користувачів зберігаються і відображаються у реальному часі, часто без додаткової обробки. Уразливість на цьому рівні відкриває шлях до викрадення сесій, підміни контенту, фішингових атак або створення умов для соціальної інженерії. [3]

Канал передачі даних між клієнтом і сервером також є джерелом загроз, особливо якщо не використовується захищене з'єднання або налаштування

HTTPS є неповними чи некоректними. У таких випадках можливим є перехоплення даних у процесі їх передачі, що відкриває можливості для реалізації атак типу «людина посередині». Особливу небезпеку становлять випадки, коли передаються незашифровані дані автентифікації, включаючи логіни, паролі чи токени доступу, оскільки це дає змогу миттєво отримати контроль над обліковим записом.

Іншою критичною вразливістю є неправильне управління сесіями. У разі відсутності механізмів автоматичного завершення неактивних сесій або недоліків у генерації сесійних ідентифікаторів відкривається можливість їх викрадення і подальшого використання для входу в систему без потреби автентифікації. Окрім того, якщо токени доступу або cookie не мають захисних атрибутів, таких як HttpOnly або Secure, вони можуть бути легко вилучені через XSS-атаки.

Важливо також враховувати загрози, пов’язані з недоліками конфігурації сервера або середовища виконання. Наприклад, відсутність обмежень на доступ до службових інтерфейсів, відкриті адміністративні панелі або доступні для зовнішнього світу діагностичні маршрути створюють додаткові вектори атаки. Також до загроз слід віднести використання застарілих компонентів з відомими вразливостями, оскільки системи, що побудовані на таких модулях, автоматично стають мішенню для сканерів і ботнетів.

Особливу увагу слід приділити загрозам, пов’язаним із відмовою у наданні послуг. Атаки типу DoS або DDoS, спрямовані на знецінення обчислювальних ресурсів або переповнення каналів зв’язку, можуть зробити систему тимчасово недоступною для легітимних користувачів.

Не менш актуальними є загрози, пов’язані з витоком інформації через логи, помилки або внутрішню діагностику. Якщо система зберігає в логах критичні дані без відповідного шифрування або обмеження доступу, це відкриває шлях до пасивного збору інформації, яка в подальшому може бути використана для побудови більш складних сценаріїв атак. Також до вразливостей слід віднести надмірну деталізацію повідомлень про помилки, які

можуть містити службову інформацію, включно з трасуванням, конфігурацією системи або фрагментами коду. На блок-схемі (рисунок 1.4.), можна побачити, скільки офенсів кожного виду потенційно нам загрожує.

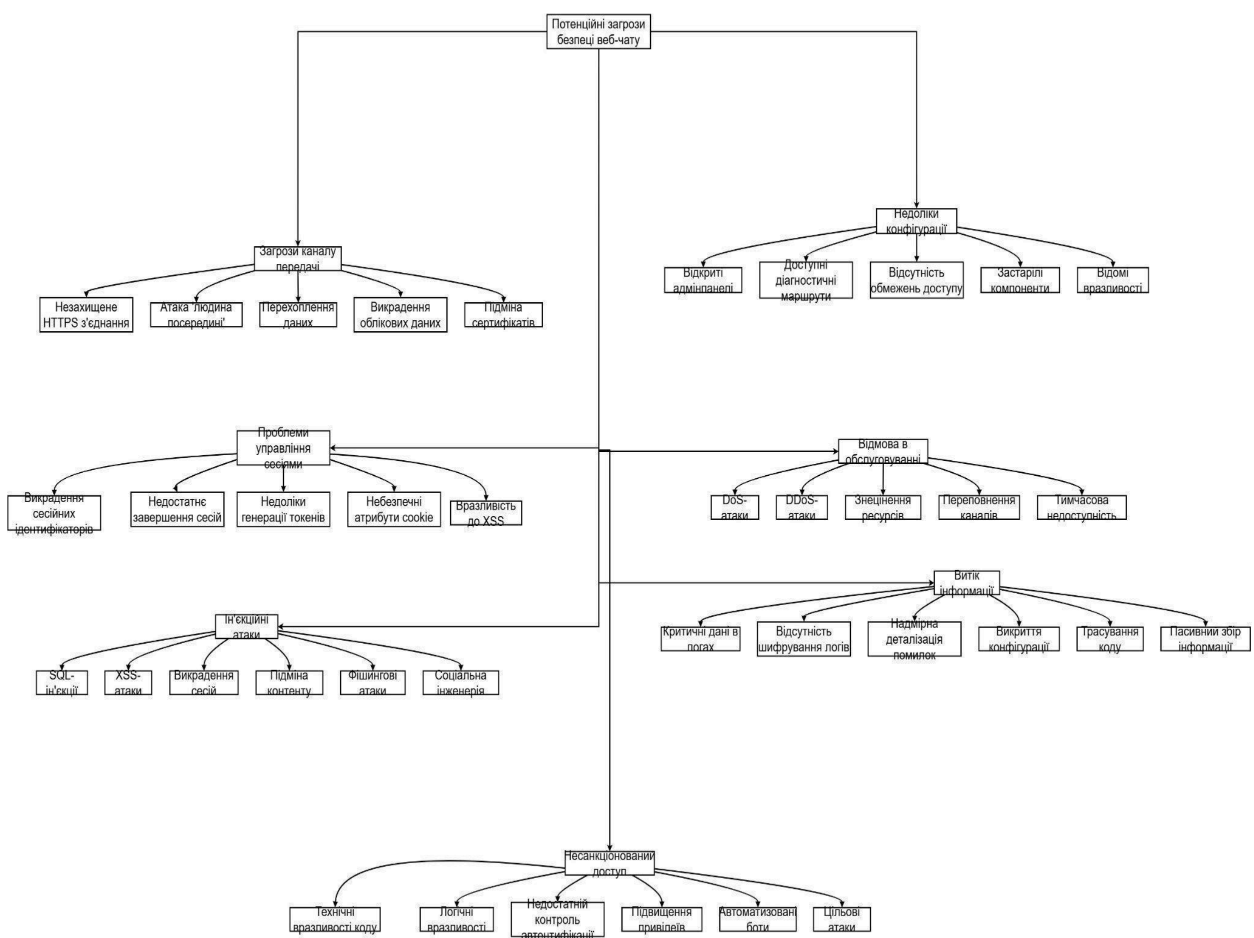

Рисунок 1.4. – Загальна блок-схема потенційних загроз інформаційної безпеки

**Висновки за розділом 1:**

У результаті проведеного аналізу об’єкта захисту - веб‑чату було встановлено низку ключових особливостей, що визначають специфіку їхньої безпеки. По‑перше, інтерактивність чатів означає, що система постійно обробляє вхідні дані від користувача, тому кожен елемент інтерфейсу повинен забезпечувати коректну та безпечну передачу повідомлень. Чат у реальному часі вимагає мінімальних затримок і одночасної підтримки великої кількості сесій.

Виділені інформаційні активи - від повідомлень і персональних даних користувачів до конфігурацій та внутрішніх логів - створюють розгалужену мережу цінних ресурсів, контроль над якими є запорукою збереження цілісності та доступності сервісу. Описані технологічне середовище та інфраструктура свідчать про важливість застосування сучасних протоколів захисту, контейнеризації та розподілених архітектур для забезпечення надійності й масштабованості рішення. Наявність чіткої ролі користувача і моделі управління доступом додає ще один рівень контролю, необхідний для реалізації принципу найменшого привілею та ізоляції сесій.

Було проаналізовано потенційні загрози, серед яких SQL- і XSS-ін’єкції, атаки на мережеві з’єднання та вразливостей суміжних сервісів. Встановлено, що найнебезпечніші вектори атак найчастіше виникають через поєднання технічних недоліків у реалізації та відсутності системного підходу до валідації вхідних даних.

# РОЗДІЛ 2. АНАЛІЗ МЕХАНІЗМІВ ТА МЕТОДІВ ЗАХИСТУ ВЕБЧАТУ

## 2.1. Основні принципи побудови системи захисту інформації для вебчату

Будова ефективної системи захисту інформації в межах веб‑додатка починається з глибокого розуміння взаємозв’язку бізнес‑процесів, інформаційних активів і потенційних загроз. Першочерговою задачею є інтеграція безпеки на всіх етапах життєвого циклу програмного забезпечення: від етапу планування і моделювання загроз до розробки, тестування, розгортання та підтримки в експлуатації. Такий підхід, відомий як «безпека за дизайном», передбачає, що кожне рішення - від вибору архітектури до написання конкретного рядка коду - оцінюється з точки зору впливу на рівень захищеності системи.

Наступним невід’ємним принципом є багаторівневість захисту. Замість того щоб покладатися на єдиний бар’єр, система містить кілька незалежних шарів контролю: мережевий екран на вході, проксі‑сервер, захищені канали зв’язку, системи перевірки даних на кордонах кожного модуля, внутрішні механізми автентифікації та авторизації, а також моніторинг і аудит. Не менш важливим є принцип мінімізації прав доступу, який передбачає надання кожному компоненту і кожному користувачеві тільки тих повноважень, які є критично необхідними для виконання їхніх безпосередніх завдань. Це обмежує потенційні втрати у разі компрометації облікового запису або сервісу, оскільки зловмисник не отримає розширеного доступу до інших частин системи. У комплексі з цим принципом часто застосовуються механізми сегментації мережі та розмежування середовищ, коли чутливі сервіси і дані фізично або логічно розпорошені по різних ізольованих зонах.

Ризик‑орієнтований підхід до вибору засобів захисту дозволяє спрямувати ресурси на найбільш критичні зони системи та адаптувати механізми безпеки до

специфіки оброблюваних даних і сценаріїв використання. Проведення регулярного аналізу ризиків та моделювання загроз забезпечує актуальність заходів і унеможливлює залишку старих політик, які вже не відповідають поточним викликам. У цьому контексті автоматизація процесів тестування безпеки та безперервне сканування вразливостей стають стандартною практикою, а результати таких перевірок напряму впливають на план вдосконалення системи.

## 2.2 Аутентифікація, авторизація та управління сесіями

Аутентифікація в межах вебчату виступає першим і найважливішим рубежем, що перевіряє особистість користувача та визначає, чи може він отримати будь-який доступ до захищених ресурсів. У сучасних реалізаціях аутентифікація часто ґрунтується на збереженні хешованих паролів у базі даних із використанням алгоритмів із додаванням «солі» та ітерацій, що значно ускладнює операції підбору паролів навіть за наявності доступу до самої бази. При вході користувач вводить свої облікові дані, які на сервері порівнюються з хешами, а у разі успішної верифікації відбувається створення сесії.

Авторизація постає наступним рівнем контролю, що визначає, які саме дії користувач із підтвердженою особою може виконувати всередині системи. Після успішної аутентифікації логіка розподілу прав спирається на роль користувача, при цьому кожна роль корелюється з конкретним набором дозволених операцій та видимістю різних елементів інтерфейсу. Ключовим аспектом є принцип найменших привілеїв, коли навіть авторизований користувач отримує лише мінімально необхідні права для виконання поточних завдань. Таким чином, оператор служби підтримки може вести й архівувати чат-сеанси, але не отримує прав на зміни в налаштуваннях платформи або доступ до закритих журналів; а адміністратор, у свою чергу, оперує найширшим набором функцій для управління системою.

Управління сесіями забезпечує зв’язок між запитом клієнта та контекстом аутентифікованого користувача в процесі його роботи з веб-додатком. Для цього зазвичай використовується механізм виписування сесійного ідентифікатора, який передається клієнту у вигляді cookie з встановленими атрибутами Secure, HttpOnly та SameSite. Ці атрибути унеможливлюють доступ до ідентифікатора з боку JavaScript та захищають від атак XSS, а також обмежують передавання cookie лише в межах одного домену. Термін життя сесії може визначатися як зафіксованою датою закінчення чинності, так і умовою неактивності: у разі тривалої відсутності дій сесійний маркер автоматично знищується, змушуючи користувача повторити процедуру входу й тим самим мінімізуючи ризик використання викрадених токенів.

Управління сесіями є наступним елементом захисту, яке підтримує зв’язок між клієнтом і сервером протягом усього часу роботи в чаті. Після підтвердження особи користувачеві видається унікальний токен сесії, який зберігається в cookie з атрибутами Secure та HttpOnly. Ці налаштування забороняють доступ до токена через JavaScript і гарантують його передавання лише через зашифроване HTTPS-з’єднання. Крім того, встановлюється таймаут неактивності — у разі відсутності взаємодії з інтерфейсом чат-сесія автоматично завершується, а токен стає недійсним. Якщо користувач виконує повторну авторизацію, старий токен відкликається, і видається новий, що запобігає повторному використанню викрадених маркерів.

Управління життєвим циклом сесії також передбачає активний моніторинг аномальної активності. Коли система фіксує незвичні патерни, такі як спроби авторизації з різних географічних регіонів протягом короткого часу або надмірну кількість невдалих входів, вона може застосовувати додаткові перевірки або примусово завершувати сесії. До такого моніторингу залучаються журнали подій, де фіксується IP-адреса, часові мітки та інші артефакти, що допомагають у розслідуванні можливих інцидентів.

Таким чином, тісна інтеграція аутентифікації, авторизації та управління сесіями створює єдину захисну парадигму. Цей підхід дозволяє забезпечити як

надійність верифікації особистості, так і чіткий контроль над тим, які саме ресурси можуть бути використані, а також гарантує, що в разі спроби зловмисних дій система відреагує своєчасно, ізоляцією недобросовісних сесій і застосуванням заходів безпеки.

### 2.3 Захист каналів зв’язку та передавання даних

Захист каналів зв’язку починається з впровадження сквозного шифрування на транспортному рівні, що гарантує конфіденційність і цілісність переданого трафіку. Для реалізації цього використовуються сучасні версії протоколу TLS, які забезпечують стійкий набір криптографічних алгоритмів, стійких до відомих атак. Процес встановлення з’єднання передбачає складний обмін повідомленнями між клієнтом і сервером, у ході якого відбувається автентифікація сервера за допомогою цифрового сертифікату, перевіреного через довірчий ланцюг центрів сертифікації, вибір спільних алгоритмів шифрування й генерація сеансових ключів. Після завершення рукостискання весь подальший обмін даними захищений симетричним шифруванням, що забезпечує мінімальні затримки й високу продуктивність обробки повідомлень.

Контроль налаштувань HTTPS включає вимогу обов’язкового використання HSTS-заголовка з максимальним терміном дії, що змушує клієнтські браузери автоматично переводити всі запити на захищений протокол, навіть якщо користувач вводить адресу без префіксу «https». Додатково сертифікати можуть фіксуватися на клієнтській стороні через механізми certificate pinning, що виключає можливість успішного проведення атаки «людина посередині» з використанням підробленого чи скомпрометованого центра сертифікації. Своєчасне оновлення та ротація сертифікатів, а також регулярна перевірка списків відкликаних сертифікатів дозволяють оперативно реагувати на загрози появи недійсних або викрадених ключів.

Для інтерактивного обміну повідомленнями в чаті застосовується протокол WebSocket, який ініціює встановлення довготермінового

TCP-з’єднання через початковий запит по HTTPS з подальшим переходом у режим WebSocket Secure. Весь потік двосторонніх повідомлень залишається зашифрованим тим самим TLS-каналом, а зведення накладних витрат лише до криптофреймів мінімізує затримку при надсиланні коротких пакетів. Механізми перевірки цілісності на прикінцевих вузлах гарантують, що жодна частина повідомлення не була змінена під час передачі, а додаткові таймаути та перевірки стану з’єднання дозволяють виявляти розриви чи спроби втручання в канал.

На рівні мережевої інфраструктури застосовуються міжмережеві екрани й системи виявлення вторгнень, що аналізують пакети даних у режимі реального часу, виявляючи аномальні патерни трафіку, несанкціоновані сканування портів або спроби переповнення з’єднань. Адаптивні захисні механізми здатні автоматично блокувати IP-адреси, що генерують підозрілий трафік, або перенаправляти їх на спеціалізовані скринінгові сервери для додаткової перевірки. Це дозволяє не тільки захистити систему від DDoS-атак, але й запобігти розповсюдженню хробаків і ботнет-активності.

Комплексний захист передбачає постійний моніторинг стану каналів зв’язку та автоматизоване оповіщення про будь-які відхилення від нормальних параметрів: сповільнення відповіді сервера під час TLS-рукостискання, збільшення кількості розривів з’єднання або невдалих оновлень WebSocket-сесій. Агреговані метрики і лог-файли аналізуються централізовано, що дозволяє вчасно виявити ознаки інцидентів інформаційної безпеки і провести глибокий розбір причин для оперативної ліквідації виявлених вразливостей. Такий багатоаспектний підхід до захисту каналів зв’язку та передавання даних створює стійку основу для безпечної роботи вебчату.

### 2.4 Захист від SQL-ін'єкцій та XSS-атак, валідація та фільтрація

У межах забезпечення стійкості вебчату до SQL-ін’єкцій та XSS-атак ключову роль відіграє налаштування процесу обробки вхідних даних таким

чином, щоб категорично виключити можливість ін’єкції шкідливого коду в запити до бази даних або вміст сторінки. Першим кроком стає відмова від безпосередньої конкатенації рядків у SQL-запитах та застосування підготовлених виразів або ORM-інтерфейсів. Ці технології автоматично відокремлюють логіку запиту від даних, які вносяться користувачем, й під час виконання запиту обробляють кожен параметр як літеральний фрагмент. Таким чином навіть якщо користувач спробує вставити фрагмент типу:

*'; DROP TABLE users;--,*

система розцінить його як текстовий вміст, а не як частину мови запитів. Крім того, для кожного запиту встановлюються чіткі обмеження на типи й довжину даних, що додатково знижує ризик випадкового або навмисного перевищення обсягів, здатних порушити структуру бази.

Паралельно з налаштуванням безпечного доступу до бази даних здійснюється ретельна валідація та фільтрація на серверному боці. Сервер контролює коректність формату електронних адрес, телефонних номерів та інших спеціалізованих полів, використовуючи надійні регулярні вирази й бібліотеки, що виключають широкий спектр «нетипових» вводів. При виявленні невідповідності запросивши дані не потрапляють до жодного виконуваного коду, а користувач отримує повідомлення про помилковий формат. Цей підхід не лише перешкоджає спробам ін’єкції, а й гарантує цілісність структури даних у системі.

Щодо захисту від XSS-атак, ключове завдання полягає в тому, щоб на рівні виводу екранувати всі символи, які могли б бути інтерпретовані браузером як частина HTML або JavaScript. Під час генерації відповіді сервер застосовує контекстне ескейпування: символи “ <, >, & “ та інші потенційно небезпечні символи перетворюються на їхні HTML-еквіваленти. Якщо вебфреймворк надає механізми шаблонізатора з автоматичним екрануванням, вони активуються за замовчуванням, а у виняткових випадках, коли потрібно передати унікод чи спеціальні символи, використовується контрольований API. Крім екранування, реалізовано політику Content Security Policy у заголовках відповідей, що

обмежує завантаження скриптів і стилів лише з довірених джерел. Це виключає можливість запуску сторонніх скриптів навіть у разі успішної ін’єкції, адже браузер відкине всі ресурси, не зазначені у списку авторизованих.

Додатковим шаром захисту стає фільтрація HTML-контенту за допомогою спеціалізованих бібліотек, що аналізують структуру тегів, атрибутів і вміст елементів. Навіть якщо користувач передає розмітку, система видаляє всі потенційно небезпечні елементи, такі як <script>, onerror чи javascript:-посилання, залишаючи лише безпечні теги для форматування тексту. Така фільтрація поєднується з механізмами валідації, щоб заблокувати як видимі, так і приховані спроби вставити шкідливий код.

Поєднання підготовлених SQL-виразів, серверної валідації, екранування вихідних даних, політики CSP [4] та фільтрації HTML створює багаторівневий бар’єр, який значно знижує ризик експлуатації класичних ін’єкційних вразливостей. Навіть за наявності нових чи нестандартних методик ін’єкцій система продовжує відмовлятися виконувати підозрілий вміст, гарантуючи безпечну роботу вебчату та збереження цілісності даних.

### 2.5 Обмеження доступу до API

Обмеження доступу до API [5] є критично важливою складовою безпеки сучасних веб сервісів, оскільки API виступають точкою входу до внутрішніх функцій системи, доступ до яких має бути контрольованим. Неналежно захищений API може стати причиною витоку даних, несанкціонованого доступу до ресурсів або навіть повного компрометування інформаційної системи. З огляду на це, обмеження доступу до API включає використання різних механізмів контролю, серед яких ключову роль відіграють сесії, токени аутентифікації та контроль частоти запитів, відомий як throttling.

Механізм сесій передбачає встановлення контрольованого стану між клієнтом і сервером. Після проходження аутентифікації користувачеві присвоюється унікальний ідентифікатор сесії, який може зберігатися у cookie

або передаватися як заголовок запиту. На сервері підтримується інформація про активні сесії, і кожен наступний запит перевіряється на наявність дійсного ідентифікатора сесії. Цей підхід дозволяє забезпечити стабільну взаємодію між клієнтом і системою протягом певного часу, з урахуванням таймаутів неактивності. Проте сесії мають обмежену гнучкість у випадках, коли система потребує масштабування або розподілення навантаження між кількома вузлами, оскільки це вимагає синхронізації сесійної інформації.

Іншим поширеним і більш масштабованим підходом є використання токенів доступу. Один з найпопулярніших форматів токенів - це JSON Web Token, який містить закодовану інформацію про користувача та його права доступу. Після проходження автентифікації користувач отримує токен, який прикріплюється до кожного запиту до API. Сервер, отримуючи запит із токеном, перевіряє його підпис, термін дії та інші параметри, перш ніж надати доступ до ресурсу. Такий підхід дозволяє уникнути зберігання стану на сервері, що значно спрощує масштабування системи. Однак важливо забезпечити безпечне зберігання та передачу токенів, а також регулярно оновлювати ключі підпису.

Ще одним важливим компонентом захисту API є обмеження частоти запитів. Throttling дозволяє запобігти зловживанню API з боку окремих користувачів або автоматизованих систем. За допомогою цього механізму система визначає допустиму кількість запитів у певний проміжок часу, які можуть бути здійснені з однієї IP-адреси або за допомогою одного токена. Якщо встановлений поріг перевищено, подальші запити тимчасово блокуються або обробляються зі сповільненням. Така практика ефективно протидіє атакам типу DoS, а також допомагає балансувати навантаження на серверну інфраструктуру.

Комбіноване використання сесій, токенів і throttling дозволяє досягти високого рівня безпеки при взаємодії з API. Це забезпечує як контроль доступу до ресурсів, так і захист від зловмисних дій, гарантуючи надійну роботу сервісу навіть у випадках підвищеної активності або потенційних атак. Усі ці механізми повинні бути налаштовані з урахуванням специфіки системи, типів користувачів

та чутливості оброблюваних даних. На рисунку 2.1. зображена загальна блок схема, яка демонструє обмеження до API у вебдодатку.

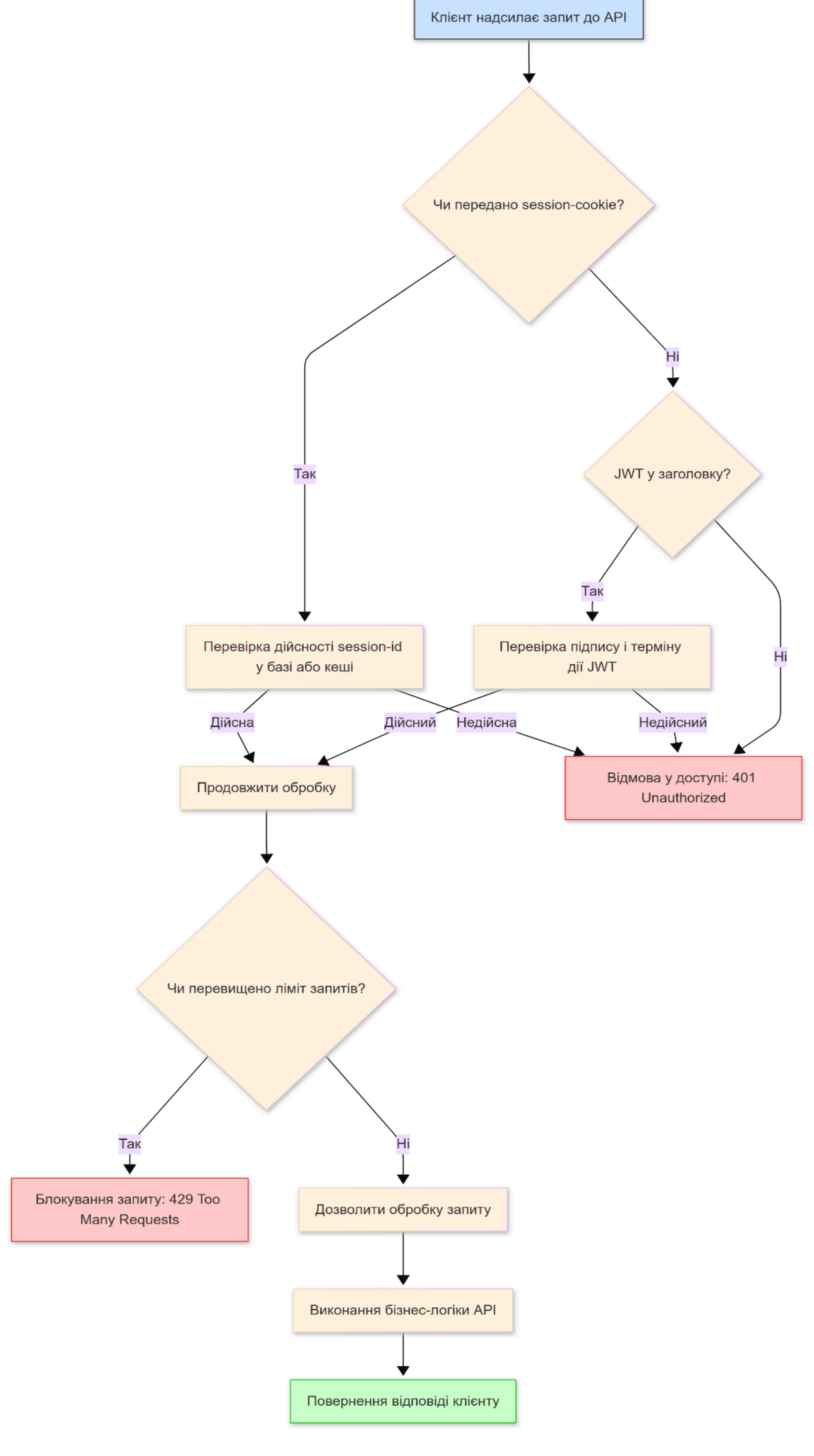

Рисунок 2.1. – Загальна блок-схема обмеження доступу до API

## 2.6 Захист від ботів, спаму та автоматизованих загроз

Захист від ботів, спаму та автоматизованих загроз є ключовим аспектом забезпечення безпеки веб застосунків, особливо коли йдеться про сервіси, що передбачають активну взаємодію з користувачами через вебчати або інші інтерфейси введення даних. Наявність незахищених точок взаємодії створює можливість для автоматизованих програм, які здатні генерувати велику кількість небажаних запитів, надсилати спам або навіть використовувати ресурс системи для реалізації інших злочинних сценаріїв, таких як збирання особистих даних, шахрайство або спроби компрометації облікових записів шляхом перебору паролів.

Одним з базових підходів до протидії таким загрозам є впровадження механізмів виявлення та ідентифікації ботів на ранньому етапі взаємодії. Це може бути реалізовано за допомогою аналізу поведінкових характеристик користувача під час заповнення форм або взаємодії з інтерфейсом. Людина має властивий спосіб переміщення курсору, швидкість введення тексту та інші ознаки, які важко точно відтворити автоматизованим засобам. Система може відслідковувати ці параметри та виявляти підозрілу активність.

Важливим інструментом у боротьбі з ботами є використання перевірок типу CAPTCHA, які покликані відрізнити людину від програми. Сучасні варіації таких механізмів, зокрема невидимі або ті, що ґрунтуються на оцінюванні ризику, дозволяють знизити навантаження на користувача при збереженні ефективності захисту. Проте цей підхід не завжди є достатнім, особливо з урахуванням постійного розвитку технологій машинного навчання, що дозволяють автоматизованим системам обходити прості форми перевірки.

Серверна логіка захисту має включати можливості моніторингу частоти та типу запитів з певних джерел. Якщо система виявляє незвично високу

активність з одного IP-адресу, повторювані однакові дії або шаблонну поведінку, ці запити можуть бути автоматично заблоковані або перенаправлені на додаткові перевірки. Крім того, застосовуються техніки аналізу заголовків HTTP, структури запитів і нетипових шаблонів, які притаманні ботам або скриптам.

Також ефективним методом є впровадження honeypot-полів - прихованих від користувача полів форм, які залишаються порожніми при легітимному введенні, але часто заповнюються ботами. Надсилання таких полів із даними слугує сигналом для блокування запиту. Іншою поширеною практикою є додавання обмежень на час, за який дозволено відправити форму, оскільки боти зазвичай виконують дії майже миттєво після завантаження сторінки.

З метою протидії автоматизованим загрозам також доцільно реалізовувати фільтрацію на рівні мережевої інфраструктури. Проксі-сервери, брандмауери веб застосунків та спеціалізовані сервіси захисту на кшталт Web Application Firewall [6] дозволяють ефективно фільтрувати шкідливий трафік ще до того, як він досягне прикладного рівня системи. Використання загальнодоступних баз даних підозрілих IP-адрес і поведінкових шаблонів значно підвищує якість такого захисту.

Таким чином, боротьба з ботами, спамом та автоматизованими загрозами потребує комплексного підходу, який включає як фронтенд-методи, так і серверні рішення, засоби аналізу поведінки, а також мережеві інструменти. Ретельне поєднання цих технологій дозволяє створити багаторівневу систему захисту, здатну ефективно протистояти як простим ботам, так і складним сценаріям атак, що використовують розподілені або обфусковані схеми дій.

## 2.7 Моніторинг, аудит та виявлення інцидентів інформаційної безпеки

Моніторинг, аудит та виявлення інцидентів безпеки утворюють єдину систему оперативного спостереження за станом інформаційної інфраструктури й забезпечення своєчасного реагування на потенційні загрози. На рівні додатка

відстежуються всі ключові події: успішні та невдалі спроби входу, зміни прав доступу, незвичайні моделі навантаження, спроби передачі аномального обсягу даних або масові звернення до API. Для цього застосовується централізована платформа збору логів та метрик, до якої надходять дані з вебсервера, WSGI-серверів, баз даних, проксі та міжмережевих екранів. Кожен лог-рядок форматується за єдиним шаблоном із включенням часових міток, ідентифікаторів облікового запису, джерела події та результату дії, що забезпечує можливість автоматизованого аналізу як на рівні простих умов («невдала спроба входу більше ніж п’ять разів за хвилину»), так і складніших кореляцій між подіями («підозрілі запити до API одночасно з незвичними змінами в конфігурації»).

Усі отримані дані збираються та аналізуються. Наприклад, різке збільшення затримок у відповіді бази даних або раптовий підйом кількості одночасних WebSocket-з’єднань може вказувати на розгортання автоматизованої бот-мережі чи початок DDoS-атаки. У разі підтвердження підозри система формує тривожні сповіщення.

Аудит інформаційної безпеки забезпечує глибокий після інцидентний аналіз і стосується вже зафіксованих подій. Для цього зберігаються необроблені логи, дампи мережевого трафіку, конфігурації серверів та знімки стану віртуальних машин. За допомогою спеціальних інструментів фахівці з безпеки відтворюють ланцюжок атак, виявляють вразливі компоненти та оцінюють ефективність скоординованих заходів реагування. Результатом такого аудиту стає перелік рекомендацій із покращення налаштувань, виправлення конфігурацій та оновлення програмних компонентів, а також вдосконалення процесів реагування на інциденти.

Система реагування на інциденти реалізує автоматичний та ручний механізми усунення загроз. Після отримання тривожного повідомлення перший рівень захисту - автоматизовані правила блокування або ізоляції виявлених джерел атаки - відсікає найбільш очевидні загрози. Паралельно команда безпеки отримує сповіщення з детальним контекстом подій та інструкціями щодо

подальших дій: від розгортання оновлень безпеки до зміни конфігураційних параметрів. Після ліквідації наслідків інциденту відбувається постінцидентний аналіз, який не лише документується, а й лягає в основу плану вдосконалення політик моніторингу та схеми резервного копіювання.

Таким чином, поєднання моніторингу, централізованого збирання логів, інтелектуального аналізу подій та аудиту створює стійкий каркас інформаційної безпеки. Ця багаторівнева архітектура дозволяє не лише виявляти й локалізувати кібератаки на ранніх стадіях, але й безперервно підвищувати рівень захищеності вебчату на основі отриманого досвіду.

**Висновки за розділом 2:**

У результаті дослідження механізмів і методів захисту вебчату було доведено, що застосування принципів комплексного захисту є необхідною умовою для побудови стійкої системи безпеки. Послідовне впровадження незалежних шарів контролю — від мережевого рівня до бізнес-логіки — створює множинний бар’єр проти зловмисних спроб проникнення та знижує ймовірність успішної атаки навіть у разі обходу одного із методів захисту. Такий підхід закладає основу для масштабованої архітектури, де нові компоненти інтегруються без ослаблення загальної безпеки.

Аутентифікація та авторизація, у поєднанні з міцним управлінням сесіями, забезпечують першочерговий рубіж захисту, фільтруючи небажані запити ще до того, як вони досягнуть бізнес-логіки. Використання одно- та двофакторної перевірки, рольових моделей доступу й безпечних токенів із коротким терміном дії дозволяє гарантувати, що до чату потрапляють лише уповноважені користувачі. Зашифровані канали зв’язку на базі TLS/HTTPS і WSS із політиками сертифікації додатково оберігають дані від перехоплення та підміни в процесі передачі.

Захист від SQL-ін’єкцій і XSS-атак досягається завдяки єдиному механізму валідації та фільтрації вхідних даних на всіх рівнях: від серверного

коду до браузерних шаблонів і політик Content Security Policy. Використання підготовлених виразів для роботи з базою даних унеможливлює підміни структурованих запитів, а екранування та контекстне ескейпування виводу захищають від виконання сторонніх скриптів у клієнтському середовищі. Обмеження доступу до API за допомогою токенів і контролю частоти запитів робить ці інтерфейси менш привабливими для автоматизованих атак, тоді як спеціалізовані алгоритми виявлення бот-активності і спаму додатково зменшують обсяг небажаного трафіку.

Централізована система моніторингу, аудит логів і аналіз логів забезпечують своєчасне виявлення аномалій і формування тривог за перших ознак інцидентів. Поєднання автоматичного реагування з ручним розслідуванням дає змогу не лише обмежити збитки, а й постійно вдосконалювати захисні механізми на основі реальних даних. У комплексі всі вищезазначені методи становлять єдину систему, у рамках якої кожний елемент взаємодіє для максимального захисту вебчату від загроз.

# РОЗДІЛ 3. РЕАЛІЗАЦІЯ КОМПЛЕКСНОГО МЕТОДУ ЗАХИСТУ ВЕБЧАТУ

## 3.1 Вибір стеку інструментів для розробки вебдодатку

Django відіграє центральну роль у серверній частині реалізації вебчату, оскільки поєднує високу продуктивність із вбудованими механізмами безпеки та чіткою структурою коду. Завдяки своєму модульному підходу Django дозволяє розділити всю логіку чат-додатку на окремі додатки (apps), кожен із яких відповідає за конкретний аспект функціоналу: моделі даних, обробка запитів або інтеграція з каналами комунікації. Перевагою є те, що ORM Django автоматично перетворює Python-об'єкти на SQL-запити, при цьому ізолюючи код бізнес-логіки від даних, що вводяться користувачем. Це суттєво знижує ризик виникнення SQL-ін'єкцій і одночасно спрощує підтримку й розвиток проєкту, адже кожна зміна в моделі даних відображається у міграціях управління базою.

Безпека у Django реалізована не лише на рівні ORM, але й через механізми проміжного програмного забезпечення (middleware), яке перехоплює запити до серверу та відповідає за захист від міжсайтової підробки запитів і міжсайтового скриптування. Кожна форма, створена з використанням вбудованого класу Form або ModelForm, автоматично містить прихований CSRF-токен, що захищає від неправомірних POST-запитів. Аналогічно, шаблонізатор Django екранує всі змінні за замовчуванням, перетворюючи потенційно небезпечні символи на безпечні HTML-еквіваленти. Це дозволяє розробникам бути впевненими, що основні загрози на рівні введення та виводу вже опрацьовані фреймворком, і зосередитися на реалізації специфічної для чату логіки.

Для підтримки обробки WebSocket-підключень Django Channels розширює стандартний HTTP-рантайм до асинхронного ASGI-інтерфейсу. Це

дозволяє обробляти одночасні з’єднання у неблокуючому режимі, створюючи справжній чат у реальному часі. Канали об’єднують повідомлення в групи, маршрутизують їх між користувачами та обслуговують сесії на рівні окремих чат-кімнат. Все це працює в межах Django-проєкту, дістаючи вигоду з його системи налаштувань, логування та контролю доступу. Оскільки Channels інтегрується з тими ж механізмами аутентифікації та авторизації, що й решта Django-додатку, адміністратори можуть використовувати знайомі інструменти для керування правами користувачів і розробки тестів.

Для фронтенду React забезпечує гнучкість і масштабованість у створенні інтерфейсу вебчату. Архітектура програми базується на компонентах, кожен із яких відповідає за своє відображення й стан. Розробка в React починається з розподілу інтерфейсу на дрібніший функціонал — обгортки для списку повідомлень, поля введення тексту, кнопки відправки та індикатори статусу. Використання хуків дозволяє обробляти життєвий цикл компонентів, підписуватися на події WebSocket-потоку та оновлювати стан без складного управління класовими компонентами. Таким чином, забезпечується миттєве відображення нових повідомлень та станів (наприклад, коли співрозмовник набирає текст), тоді як React efficiently проводить порівняння (reconciliation) і оновлює тільки ті частини DOM, що дійсно змінилися.

Механізм Context API у React забезпечує глобальну передачу даних, таких як токен аутентифікації, налаштування теми або об’єкт WebSocket-підключення, без необхідності передавати їх через props кожному компоненту. Це особливо корисно у чаті, де різні компоненти мають звертатися до спільного потоку повідомлень і метаданих. React Router використовується для маршрутизації в межах односторінкового застосунку, що дозволяє легко змінювати контекст між різними чат-кімнатами чи переглядати історію без перезавантаження сторінки. Підхід до стилізації, заснований на CSS-in-JS (наприклад, styled-components або Emotion), дає змогу писати стилі безпосередньо в коді компонентів, автоматично ізолюючи їх від глобальних CSS-конфліктів і підтримуючи динамічні теми.

Nginx виступає в ролі зворотного проксі та балансувальника навантаження, приймаючи всі запити від клієнтів і передаючи їх на відповідні бекенд-сервера Django або ASGI для WebSocket. Він забезпечує високопродуктивне SSL-термінування, розвантажуючи прикладний сервер від обробки шифрування і дешифрування, та прискорює доставку статичних ресурсів, використовуючи власний кеш. Конфігурація Nginx дозволяє встановити з'єднання WebSocket з довготривалими таймаутами, автоматично повторно підключатися у випадках розриву та проксувати HTTP-запити до Django через сокет або порт, залежно від реалізації. Завдяки Nginx можна також реалізувати обмеження частоти підключень, захист від DDoS і підключати WAF-модулі для фільтрації підозрілого трафіку ще на кордоні мережі, що суттєво підвищує загальну стійкість чат-системи.

Для спрощення процесу локального розгортання та забезпечення послідовності середовища розробки, а також з метою наближення його до продакшн-конфігурації, застосовується контейнеризація за допомогою Docker. Відповідно до цієї методології кожен із ключових компонентів чат-додатку — вебсервер Django, ASGI-воркер для обробки WebSocket-з'єднань, база даних і проксі Nginx — пакується в окремий контейнер з власним Dockerfile. У такому підході налаштування залежностей фреймворку, версії мови Python, системних бібліотек і змінних середовища описуються декларативно, що гарантує, що на будь-якій машині, де запущено Docker.

Оркестрація контейнерів здійснюється за допомогою Docker Compose, який координує одночасний запуск усіх сервісів і їх взаємодію. Конфігураційний файл визначає зв'язки контейнерів: Nginx передає запити як до HTTP-ендоіндентів Django, так і до WebSocket-портів ASGI-воркера. Крім того, він передбачає монтування локальних директорій з кодом і файлами налаштувань до контейнера розробки, що дає змогу миттєво відображати зміни без необхідності нового створення образів, а також використовувати "named volumes" для збереження даних бази між перезапусками контейнера.

## 3.2 Реалізація функціоналу вебчату як окремого модуля

Реалізація функціоналу вебчату як окремого модуля розпочалася з проєктування базових компонентів моделі даних у середовищі PostgreSQL[7]. За основу було взято структуру, що передбачає наявність таблиць для збереження інформації про користувачів, чат-сесій, повідомлень та вкладень. Кожен запис у таблиці повідомлень містить унікальний ідентифікатор, зв'язок із відповідною сесією та автором, часову мітку з високою точністю та можливість зберігання текстового вмісту або посилання на медіафайли. Використання PostgreSQL обумовлено його здатністю обробляти складні транзакції, підтримувати індексацію тексту для швидкого пошуку та гарантовано виконувати атомарні операції, що критично важливо у випадку одночасної обробки сотень чи тисяч повідомлень у режимі реального часу. Індексація полів для пошуку за автором чи контекстом повідомлення в поєднанні з ретельним налаштуванням параметрів connection pooling дозволяє уникнути перевантаження під час пікової активності, адже кожний одноразовий запит до бази, сформований через ORM Django, автоматично транслюється у параметризований SQL-запит із підключеними змінними, що знижує ризик SQL-ін’єкцій і підвищує продуктивність.

Переходячи до серверної частини, ядром бекенд-логіки став Django [8] із підтримкою асинхронних можливостей через Django Channels та систему фонового виконання завдань на базі Celery з Redis як брокером повідомлень. Django виконувало роль основного WSGI-фреймворку, що опікувався маршрутизацією HTTP-запитів та базовою бізнес-логікою авторизації, тоді як Django Channels відповідав за створення та підтримку довготривалих WebSocket-з’єднань. Після аутентифікації користувача Django підхоплював контекст його сесії та передавав інформацію у Channels-роутер, де кожне WebSocket-з’єднання інкапсулювалося в окремий consumer-об’єкт. Ці об’єкти

ділилися на дві великі категорії: ті, що обслуговували текстовий обмін повідомленнями, та ті, що відповідали за передачу повідомлень системних сповіщень, наприклад про досягнення певного порогу невдалих спроб входу або підозрілої активності. Взаємодія між стандартними HTTP-контролерами та Channels-консьюмерами здійснювалася через спільне сховище у Redis: коли сервер отримував нове повідомлення через REST-ендоіндпункт, він записував відповідну подію у Redis канал, а потім асинхронний consumer одразу отримував це повідомлення і транслював його через WebSocket клієнту.

Незалежно від того, чи користувач вводив новий текст, чи додавав вкладення, весь процес опрацьовувався двома етапами. По-перше, HTTP-запит, що надходив до Django, проходив валідацію та фільтрацію, реалізовану через форми й серіалізатори Django REST Framework: перевірявся формат тексту, допустимі символи, наявність CSRF-токену, а у випадку медіавкладень відбувався контроль типу та розміру файлу. Після успішного проходження валідації дані зберігалися в базі, генерувався запис повідомлення з відповідними метаданими, а результуючий об’єкт передавався в чергу Celery для обробки мультимедійних додатків: наприклад, створення зменшених ескізів зображень або конвертацію відеофрагментів у потрібний формат. Ці завдання виконувалися у фоні, що дозволяло не затримувати потокову передачу текстових повідомлень і не блокувати основний потік виконання сервера.

У ролі брокера повідомлень між різними процесами та сервісами було обрано Redis. Він одночасно виконував роль кешу для зберігання поточних станів WebSocket-з’єднань та черги завдань для Celery. Коли приходив новий HTTP-запит із повідомленням, сервер записував його у базу, а потім додавав завдання у чергу Celery із вказівкою обробити вкладення чи надіслати push-сповіщення іншим користувачам. Celery-воркери, запущені окремими процесами або контейнерами, підхоплювали завдання з Redis та виконували їх в асинхронному режимі. Наприклад, якщо оператор чат-служби надсилав шаблонну відповідь, то миттєво відправлявся запит до Celery, який перевіряв статус сесії, нотував час відправлення в базі та через WebSocket повідомляє

клієнта про нове вхідне повідомлення. Паралельно відбувався запис змін у логах подій, які збиралися централізовано для моніторингу та аудиту.

На рівні розгортання застосовано поєднання Gunicorn та Nginx. Gunicorn виступає в ролі WSGI-серверу, який запускає Django-додаток у вигляді декількох робочих процесів, кожен з яких слухає UNIX-сокет або порт. Така архітектура дозволяє масштабувати кількість воркерів відповідно до доступних ресурсів хостинга, уникаючи блокувань та збільшення часу відгуку. Nginx, розміщений перед Gunicorn, приймає всі зовнішні запити, обробляє SSL-шифрування за допомогою сертифікатів Let’s Encrypt, кешує статичні файли (CSS, JavaScript та зображення) і виконує роль зворотного проксі. У разі запиту до WebSocket-ендоіндпункту, Nginx передає з’єднання на спеціально налаштований upstream, який підтримує довготривалі WebSocket-сесії, враховуючи необхідні для протоколу заголовки «Upgrade» та «Connection: keep-alive». Завдяки такому поділу обов’язків Gunicorn-воркери можуть зосередитися виключно на обробці Python-коду, а Nginx оптимізує мережевий трафік та надає додаткові механізми захисту, зокрема обмеження максимальної кількості одночасних підключень, фільтрацію за IP-адресами та просторове балансування навантаження між декількома екземплярами Gunicorn.

Фронтенд частина вебчату, побудована на React [9], складається з сукупності компонентів, що реалізують весь життєвий цикл обміну повідомленнями: починаючи від введення тексту у компоненті input, перевірки сформованого рядка на порожність або заборонені символи, закінчуючи відтворенням успішно отриманого повідомлення у компоненті списку. React-додаток ініціалізується з передачею йому початкових налаштувань: URL для WebSocket-підключення, необхідні токени автентифікації та початковий набір повідомлень з локального кешу (якщо користувач перезапустив браузер). Компоненти React підтримують розділення відповідальності: ChatWindow відповідає за загальне розміщення елементів, MessageList динамічно рендерить кожне повідомлення за допомогою ключів і забезпечує плавну анімацію появи, а MessageInput обробляє всі події введення і надсилає готовий текст у бекенд. За

допомогою React Context API усі компоненти можуть звертатися до спільного об'єкта, який містить поточний стан підключення WebSocket, інформацію про користувача та методи для відправки нових повідомлень. За потреби компонент EmojiPicker додає у текстових полях можливість вставки спеціальних символів, а FileUploader обробляє drag-and-drop або вибір файлу через файловий діалог, одразу завантажуючи файл на сервер через REST-ендоіндпункт та очікуючи на повернене посилання для вставки у текст.

Інтеграція фронтенду та бекенду забезпечується через єдину точку входу: Nginx спочатку надає статичні React-бандли, а всі запити, що стосуються API-ендоіндпойнтів або WebSocket, перенаправляє до Gunicorn-воркерів або на асинхронний Channels-сервер. Таким чином, уникнуто конфліктів, оскільки фронтенд і бекенд працюють в одному домені й порту. На рисунку 3.1 зображено розроблений вебдодаток.

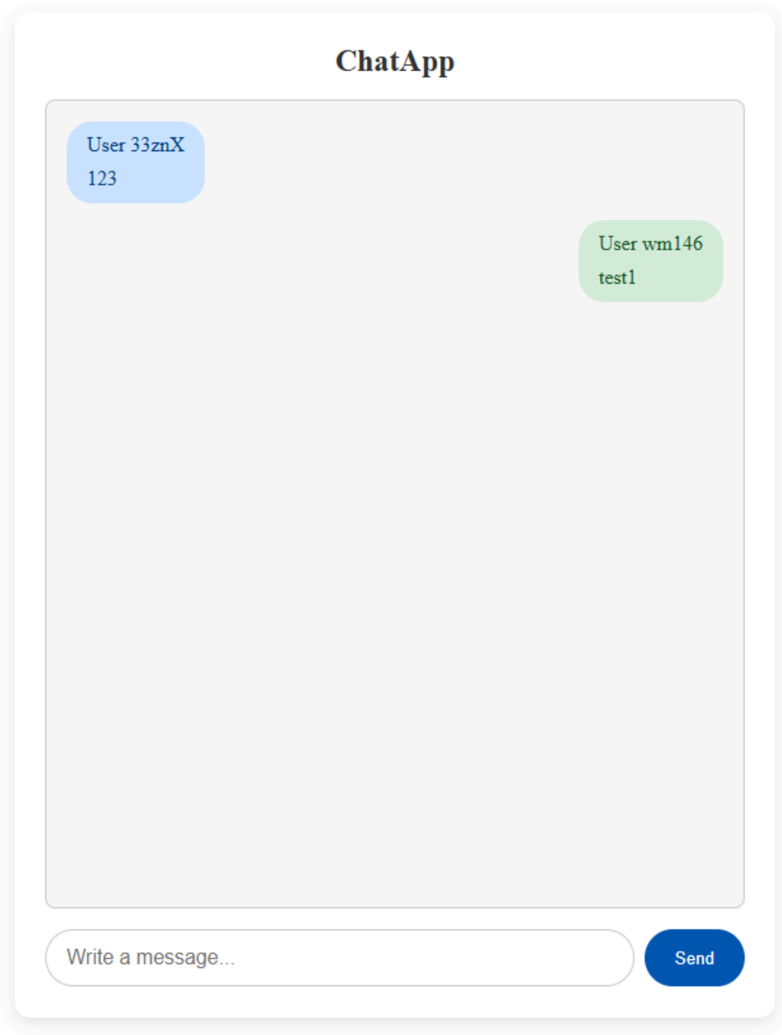


Рисунок 3.1. – Результат реалізації вебдодатку

## 3.3 Впровадження методів захисту у процеси обміну даними

У процесі розробки вебчату кожна операція обміну даними — від моменту введення тексту користувачем до його відображення в інтерфейсі одержувача — підлягає багаторівневому захисту для запобігання несанкціонованому доступу. Впровадження методів захисту в усі ключові точки обробки і передачі повідомлень починається з моменту, коли клієнт ініціює відправку нового тексту до серверу. Перед формуванням запиту браузер попередньо перевіряє коректність введення: поле введення повідомлення містить логіку клієнтської валідації, яка відсіює порожні рядки, надто великі повідомлення та символи, що не відповідають допустимому набору (наприклад,

деякі Unicode-символи чи заборонені HTML-теги). Паралельно з цим кожен запит супроводжується CSRF-токеном, що формується під час завантаження вебсторінки й асоціюється з активною сесією користувача. Сам CSRF-токен зберігається в cookie з атрибутами HttpOnly та Secure, що виключає можливість його прочитання за допомогою JavaScript і передавання лише через захищений канал HTTPS. У разі, якщо запит не містить дійсного токена або він не відповідає очікуваному значенню, сервер миттєво повертає клієнту помилку аутентифікації та не переймає спробу записати повідомлення до черги обробки.

Коли запит потрапляє до серверної частини, Django-процес аналізує заголовки та тіло повідомлення, спочатку перевіряючи цілісність TLS-з'єднання. Налаштування Nginx гарантує, що всі HTTP-запити автоматично перенаправляються на HTTPS із використанням редиректу та HSTS, а WebSocket-комунікація відбувається через WSS-протокол, що захищає трафік на транспортному рівні. Завдяки цьому маніпуляції із проміжним зв'язком — атаки "людина посередині" — унеможливлюються вже на рівні SSL/TLS. Сервер додатково перевіряє, чи сертифікат клієнта (у разі використання mTLS) належить довіреному центру сертифікації, що особливо важливо, якщо внутрішні сервіси мікросервісної архітектури також взаємодіють через захищені канали.

Після встановлення захищеного з'єднання система переходить до валідації отриманих даних. Серверні серіалізатори та форми на базі Django REST Framework виконують двоетапну процедуру перевірки. Спочатку перевіряються загальні характеристики: чи є обов'язкові поля присутніми, чи не перевищує рядок максимально допустимої довжини та чи відповідає він заздалегідь затвердженому шаблону для тексту повідомлення. Далі вбудовані механізми регулярних виразів перевіряють уміст на предмет наявності заборонених конструкцій: спроб вставити HTML-теґи, атрибути "onerror" чи "javascript:" у текст або фрагменти, що можуть призвести до SQL-ін'єкцій. Саме тут ORM-шар Django бере на себе роль фасилітатора: підготовлені запити (prepared statements) автоматично замінюють будь-які змінні, які користувач

може передати, у літеральні параметри, і тому вставлений зловмисний синтаксис не розпізнається як частина SQL-команди, а інтерпретується лише як текстовий фрагмент. Це дає змогу зводити до нуля ризик структурного вторгнення в запити.

Одночасно із захистом від SQL-ін’єкцій реалізується й контекстне екранування (escaping) на рівні шаблонів. Django-шаблонізатор за замовчуванням застосовує HTML-екранування до всіх змінних, тому навіть якщо зберегти вміст із потенційно небезпечними символами, при рендерингу сторінки вони перетворюються на HTML-еквіваленти “<”, “>” та “&”. Якщо спеціально передається конструкція, яка повинна бути відображена без екранування (наприклад, емодзі або форматований текст), то для цього передбачено окремий API, який пропускає дані через додаткові перевірки фільтрації HTML. Усі стилі та скрипти на сторінці контролюються політикою Content Security Policy (CSP), що блокує завантаження зовнішніх скриптів з недовірених доменів. Таким чином, навіть якщо хтось навмисно або випадково позначить текст як <script>alert(“XSS”)</script>, браузер не виконає цей код, оскільки у CSP не вказано джерел, з яких можна завантажувати скрипти.

У разі доставки повідомлення до користувача подальший потік даних також залишається під контролем. Компонент React на клієнтському боці отримує JSON-відповідь із сервера, що містить чистий текст або заздалегідь відфільтрований HTML. React відображає кожний елемент у MessageList, використовуючи ключі для оптимізації оновлення DOM і попереджаючи спроби рендерингу потенційно небезпечного JavaScript-коду. Жодного разу не допускається застосування innerHTML без попередньої фільтрації, і весь текст вставляється за допомогою звичайного JSX-виразу {message.text}, що гарантує екранування спільних символів.

У ситуаціях, коли користувач додає вкладення, спочатку на клієнтській стороні відбувається перевірка розміру та типу файлу. Якщо файл відповідає критеріям безпеки, React відправляє його через multipart/form-data до окремого REST-ендоіндпойнта, де Django обробляє його в окремому потоці Celery.

Фоновий воркер Celery бере файл, перевіряє контрольну суму (checksum), можливо, сканує на зловмисне програмне забезпечення (за допомогою інтегрованих антивірусних скриптів), шифрує його за допомогою симетричного алгоритму перед зберіганням у об’єктному сховищі (наприклад, AWS S3 або MinIO) та повертає підписане посилання (signed URL), яке дійсне протягом обмеженого часу. Після цього фронтенд автоматично отримує посилання та вбудовує його у вигляді <img> або <a href> у чат, при цьому React гарантує, що вкладення відображається без виконання стороннього коду, а браузер завантажує його лише з безпечного підписаного URL. На рисунку 3.2 зображено приклад передачі логів додатком.

| Status | Method | Domain | File |
|---|---|---|---|
| 200 | GET | local... | log.html?_=1609232413232 |
| 200 | GET | local... | log.html?_=1609232413233 |
| 200 | GET | local... | log.html?_=1609232413234 |
| 200 | GET | local... | log.html?_=1609232413235 |
| 200 | GET | local... | log.html?_=1609232413236 |
| 200 | GET | local... | log.html?_=1609232413237 |
| 200 | GET | local... | log.html?_=1609232413238 |
| 200 | GET | local... | log.html?_=1609232413239 |

Рисунок 3.2. – Передача логів вебдодатком

Кожен етап взаємодії — від моменту введення тексту до збереження вкладення — залишає по собі запис у централізованій системі логування. Django надсилає структуровані лог-рядки до ELK-платформи (Elasticsearch–Logstash–Kibana) або іншим SIEM-системам через UDP чи HTTP інтерфейс, де кожний лог містить часову мітку, ідентифікатор користувача, тип події (наприклад, “відправлено повідомлення” або “помилка валідації”) і результат дії. На рівні моніторингу задаються правила кореляції подій: якщо фіксується більш ніж п’ять невдалих спроб відправлення через неправильний формат за одну хвилину, генерується тривожне повідомлення в чат

Оперативного Центру Безпеки; якщо Celery-воркер повертає помилку шифрування файлу, адміністраторам надсилається SMS-сповіщення про можливі проблеми з ключами; якщо Nginx зафіксував серію відмов в обробці WebSocket-з’єднань, запускається автозамінник трафіку на резервний вузол. Таким чином, усі шари обробки даних — і клієнтський, і серверний, і обробка фонова — перебувають під жорстким контролем, що не дозволяє жодному шкідливому вмісту потрапити у систему або бути виконаним у браузері.

У проекті вебчату одним із критичних завдань стало забезпечення безпечного обміну даними між клієнтською частиною, що розгортається на окремому домені або порту, та бекендом на Django. Для цього було впроваджено механізм Cross‑Origin Resource Sharing [13], який визначає, які зовнішні джерела мають право звертатися до API, і у який саме спосіб. Відсутність правильно налаштованого CORS може призвести до того, що браузер блокуватиме запити із React‑додатку, якщо він розміщений на іншому походженні (домені, порту чи схемі).

У рамках Django‑сервера рішення було реалізовано через встановлення та налаштування спеціальної бібліотеки django-cors-headers. Вона підключається як Django‑middleware, розміщуючись безпосередньо у списку проміжного програмного забезпечення (middleware) вище, ніж компонент, що обробляє безпеку (SecurityMiddleware), але нижче за базові CORS‑middleware. Після імпорту пакету у файл налаштувань settings.py було додано конфігурацію, яка обмежує довіру лише до необхідних походжень: за замовчуванням дозволено тільки ті URL, де розгорнуються фронтенд-бандли. Задано точні рядки з іменами домену, включно із протоколом (наприклад, https://chat.example.com та https://admin.example.com), оскільки на етапі розробки React‑додаток запускався на http://localhost:3000, а у виробничій інфраструктурі — на окремому домені. Таким чином, у момент отримання HTTP‑запиту сервер перевіряє заголовок Origin, порівнює його з переліком дозволених походжень (CORS_ALLOWED_ORIGINS) та, у разі відповідності, додає у відповідь необхідні заголовки Access-Control-Allow-Origin, Access-Control-Allow-Methods

і Access-Control-Allow-Headers. Якщо ж походження не зазначене у списку, сервер не надсилає цих заголовків, й браузер одразу блокує запит ще на рівні клієнтської інфраструктури.

На стороні React‑інтерфейсу, щоб гарантувати правильне формування CORS‑запитів, усі виклики до бекенд‑ендпоінтів були обгорнуті у стандартний fetch, налаштовану з параметром withCredentials: true, коли необхідно передавати cookies із сесією. При цьому в конфігурації axios вказано базовий URL API, а також перелік заголовків, які можуть знадобитися для аутентифікації (наприклад, Authorization: Bearer <token>), що автоматично відправляються під час запиту. Якщо клієнт повинні додати специфічні заголовки (наприклад, CSRF‑token у заголовку X-CSRFToken), React‑код завжди попередньо отримує цей токен із cookie та передає разом із кожним POST/PUT/DELETE‑запитом, не забуваючи патчити його у заголовок CORS‑“whitelist”.

Особливостями налаштування стали ситуації, коли потрібно було обробити “preflight” запити, тобто запити типу OPTIONS, які браузер відсилає до сервера перед реальним викликом, щоб отримати підтвердження підтримки методів та заголовків. В django-cors-headers виявилося достатньо лише правильно вказати методи, що підтримуються (CORS_ALLOW_METHODS), і назви дозволених заголовків (CORS_ALLOW_HEADERS). Після цього Django‑сервер — автоматично, завдяки middleware — відповідав із усіма необхідними заголовками Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS та Access-Control-Allow-Headers: Content-Type, Authorization, X-CSRFToken, що вирішувало проблему з блокуванням preflight‑запитів.

На етапі тестування виявилося, що безпосереднє використання символу “*” у дозволених походженнях (тобто CORS_ALLOW_ALL_ORIGINS = True) є неприйнятним через міркування безпеки: такий підхід дозволяє будь-якому домену звертатися до API, що у поєднанні з CSRF може призвести до викрадення сесій користувачів. Замість цього було обрано стратегію “explicit

origins”: всі URL, які починаються з довірених адрес, було задано через масив рядків. Список походжень оновлюється динамічно, залежно від середовища, — у файлі конфігурації налаштовані відповідні змінні середовища CORS_ORIGIN_WHITELIST.

Оскільки чат працює через WebSocket, а не через суто HTTP‑протоколи, виникла потреба окремо забезпечувати захист на рівні WSS. На відміну від HTTP, WebSocket‑handshake відбувається через спеціальні заголовки Upgrade: websocket і Connection: Upgrade. Django Channels, опрацьовуючи такі запроси, також перевіряє заголовок Origin і порівнює його з дозволеним списком у налаштуваннях CORS_ALLOWED_ORIGINS. Завдяки цій перевірці WebSocket‑з’єднання приймаються лише від дозволених доменів. Також у Nginx, який проксуватиме WSS‑трафік до бекенду, було окремо налаштовано директиву map з перевіркою Origin і додаванням заголовка Access-Control-Allow-Origin у відповідь для WebSocket. Таким чином ще на рівні проксі запобігалося спробам встановити підроблене з’єднання з невідомих походжень.

### 3.4 Застосування валідації, фільтрації та захисту від ін’єкцій

У межах забезпечення надійного функціонування вебчату критично важливою є реалізація механізмів валідації, фільтрації та захисту від ін’єкцій. Ці процеси мають бути вбудовані як на рівні взаємодії з користувачем, так і всередині серверної логіки, забезпечуючи цілісність системи навіть у разі спроб шкідливого впливу. У сучасній архітектурі вебдодатків, реалізація цього захисту можлива завдяки поєднанню локальних і мережевих інструментів. Для цієї мети ефективно використовуються такі рішення, як Cloudflare, Django REST Framework (DRF), система фільтрації на основі Nginx, а також механізми ескейпування, інтегровані в шаблонізатори на боці бекенду.

Першим рівнем безпеки виступає периметровий захист, який реалізується через інтеграцію із сервісом Cloudflare. Завдяки вбудованому WAF (Web

Application Firewall), Cloudflare виконує глибоку перевірку вхідних HTTP-запитів, використовуючи актуальні сигнатури атак, а також аналіз шаблонів. Якщо запит містить підозрілі конструкції, наприклад SQL-вирази, JavaScript-вставки або фрагменти, що імітують ін'єкційні спроби, він не потрапляє до сервера взагалі — Cloudflare відсікає його ще до надходження до Nginx. Особливо важливою є здатність автоматично реагувати на масові спроби атак, включно з брутфорсом чи масовим надсиланням запитів на форму зворотного зв'язку, що в контексті чату особливо актуально.

Другим важливим етапом є рівень вебсервера, де застосовується Nginx як зворотний проксі. Окрім маршрутизації трафіку та обробки статичних файлів, Nginx дозволяє застосовувати базову фільтрацію запитів — наприклад, обмеження на максимальний розмір тіла запиту, блокування доступу до потенційно вразливих шляхів або обробка заголовків, які можуть бути використані в контексті ін'єкцій. Тут же можна застосувати вбудований модуль фільтрації, а також інтеграцію з ModSecurity для додаткового рівня фільтрації на основі OWASP Core Rule Set, який спеціалізується саме на виявленні ін'єкцій.

Коли запит досягає рівня застосунку, до обробки вхідних даних залучається серверна логіка на Django. Django, особливо у зв'язці з Django REST Framework, забезпечує чіткий контроль над валідацією вхідних параметрів. Через серіалізатори DRF перевіряється відповідність типів, довжини, допустимих символів і регулярних шаблонів. Наприклад, якщо користувач надсилає повідомлення в чат, перевіряється, що це повідомлення не містить шкідливих конструкцій, не перевищує допустиму довжину, а також не включає символів або тегів, які могли б бути інтерпретовані браузером як виконуваний код. Навіть якщо зловмисник вручну намагається змінити JSON-структуру запиту або додати в неї небезпечні конструкції, серверна валідація гарантує, що такі дані будуть відхилені до моменту виконання будь-якої логіки обробки.

Додатковим компонентом виступає етап виводу даних на фронтенд. Оскільки React рендерить HTML у браузері клієнта, критично важливо уникати

ситуацій, коли текст, введений користувачем, перетворюється на активний код. Тут на допомогу приходить ескейпування — автоматичне перетворення спеціальних символів у HTML-сутності. Символи <, >, &, ", ' замінюються на відповідні коди, що виключає можливість XSS-ін'єкцій через текстове поле. Якщо з якоїсь причини фрагмент розмітки потребує рендерингу як HTML (наприклад, відображення форматованого повідомлення), його потрібно пропускати виключно через строго контрольований API, який дозволяє лише безпечні теги. На рівні шаблонів Django застосовується контекстне ескейпування за замовчуванням, що також захищає внутрішній інтерфейс адміністратора або інші службові сторінки від відображення потенційно небезпечного вмісту.

У разі виконання асинхронних завдань за допомогою Celery та Redis, наприклад, при надсиланні системних повідомлень або сповіщень користувачам, теж відбувається валідація вхідних параметрів до передачі даних у чергу. Це забезпечує захист від сценаріїв, у яких зловмисник міг би обійти основну серверну логіку і передати некоректні дані безпосередньо в Celery, провокуючи помилки або ін'єкції на рівні фонової обробки.

Завершальним рівнем є формування політик безпеки у відповіді сервера. Через налаштування заголовків Content Security Policy (CSP) можна обмежити браузеру можливість виконувати будь-які скрипти, окрім явно дозволених. Навіть у разі, якщо ін'єкція HTML вдалася, виконання вставлених скриптів буде заблоковане політикою браузера. Встановлення заголовків X-Content-Type-Options, X-Frame-Options, Referrer-Policy та інших посилює цей захист і формує єдину політику взаємодії з браузером.

### 3.5 Реалізація захисту від автоматизованих загроз

Захист від розподілених атак відмови в обслуговуванні є одним із найважливіших компонентів забезпечення стійкості вебсервісів до навмисного перевантаження. У контексті функціонування вебчату, який є постійно

відкритим каналом зв’язку між користувачем і сервером, ризик атак типу DDoS є особливо критичним

Сучасні рішення, зокрема Cloudflare, реалізують захист від DDoS-атак [10] на рівні власної глобальної мережі доставки контенту, яка фізично розташована ближче до клієнта і бере на себе обробку запитів ще до того, як вони досягають основного сервера розробника. У разі атаки типу HTTP Flood, коли мільйони ботів надсилають стандартні HTTP-запити до вебресурсу з метою виснаження ресурсів, Cloudflare виконує розподілену обробку трафіку в реальному часі, виявляє аномальну активність і автоматично вмикає захисні правила. На практиці це означає, що переважна більшість шкідливих запитів навіть не дійде до внутрішнього додатка, а буде заблокована на рівні мережевої інфраструктури Cloudflare.

Окремо варто зазначити про атаки нижчого рівня, які націлені на порушення базових механізмів встановлення з’єднань. Такі атаки зазвичай не генерують повноцінні HTTP-запити, а натомість перевантажують мережевий стек, використовуючи великий обсяг незавершених TCP-з’єднань або неочікуваних UDP-пакетів. Cloudflare перехоплює та нейтралізує ці атаки за допомогою спеціалізованих фільтрів на рівні TCP/UDP, які забезпечують повну обробку трафіку без участі кінцевого сервера. Це дозволяє не лише зберігати продуктивність вебчату, але й уникати зайвих витрат обчислювальних ресурсів, які могли б бути витрачені на відхилення великої кількості запитів вручну.

Однією з ключових переваг такого захисту є миттєва реакція на сплески трафіку, що не потребує втручання адміністратора або розробника. Cloudflare постійно відстежує глобальні шаблони загроз і оновлює свої правила фільтрації на основі аналізу мільярдів запитів з усього світу. Таким чином, навіть нові або ще не класифіковані вразливості, які використовуються в рамках DDoS-кампаній, можуть бути ефективно заблоковані до того, як буде завдано реальної шкоди. На рисунку 3.3. зображено фрагмент панелі адміністратора для управління захистом від DDoS-атак.

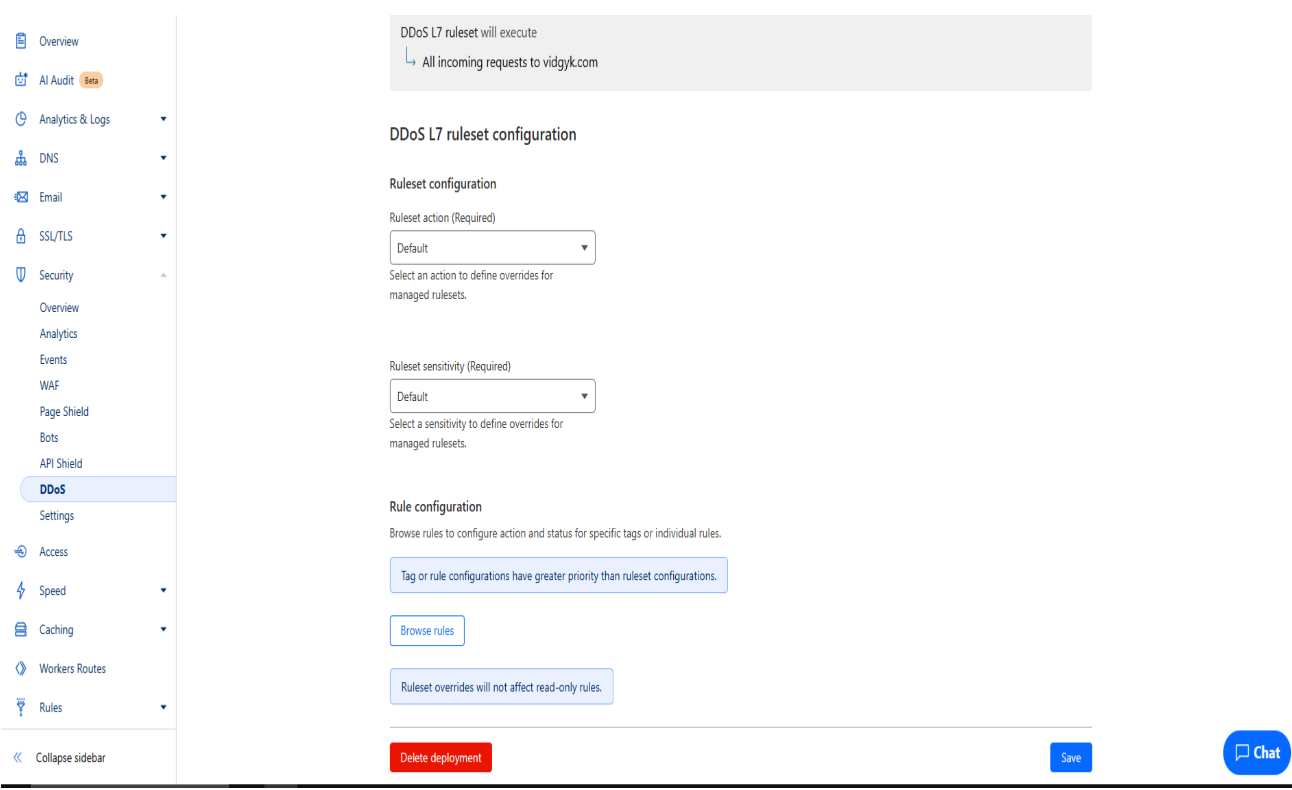


Рисунок 3.3. – Налаштування захисту від DDoS-атак за допомогою Cloudflare

Впровадження захисту від ботів із використанням Cloudflare [11] починається з активації можливостей Bot Management у панелі керування сервісом. Після внесення домену до облікового запису Cloudflare у розділі безпеки необхідно перейти до налаштувань Bot Fight Mode або Bot Management. У разі обрання Bot Fight Mode система автоматично починає відстежувати характерні ознаки автоматизованих запитів (відмінності у поведінці, заголовках, виконанні JavaScript) та відсіює їх ще на рівні глобальної мережі Cloudflare. Якщо сайт вимагає більш гнучкого підходу, доцільно перейти на повноцінний Bot Management, який дає змогу створювати власні правила та застосовувати скоринг бот-трафіку на основі машинного навчання. У результаті після цих початкових кроків усі вхідні запити оцінюються за внутрішніми алгоритмами Cloudflare: якщо запит розпізнається як бот, він буде перенаправлятися на сторінку перевірки, не досягаючи бекенду додатку.

Після активації базових налаштувань Bot Management необхідно налаштувати WAF-правила для тоншого фільтрування, яке враховує власну

специфіку вебчату. У цій частині конфігурації адміністратор створює правила, що блокують запити з бот-скором нижче певного порогу або з аномально високою частотою викликів API. Наприклад, якщо бот здійснює десятки запитів на секунду до одного і того ж ендпоінта чату, система автоматично обмежить або уповільнить такі запити відповідно до заданої політики. Cloudflare зберігає статистику бот-активності у розділі аналітики, що дозволяє аналізувати пікові навантаження та своєчасно налаштовувати умови реагування.

Водночас для забезпечення гнучкого керування ситуаціями із помилковою ідентифікацією легітимного користувача як бота слід налаштувати виключення для довірених IP-адрес або згрупованих URL-адрес, які відповідають внутрішнім сервісам (наприклад, CRM-інтеграції), які теж здійснюють велику кількість запитів за природними потребами. Для цього в Bot Management передбачена можливість створення списків делегованих IP (Allowlist), що виключаються з автоматичної блокувальної логіки. Крім того, адміністратор може задати власні критерії перевірки, наприклад перевіряти обов’язкову наявність певного заголовка або токена, який підтверджує легітимність запиту. Це важливо в контексті вебчату, де підключення WebSocket та REST-запити можуть спрацьовувати занадто часто в разі великої кількості активних користувачів. На рисунку 3.4. зображено фрагмент панелі адміністратора для управління захистом від ботів.

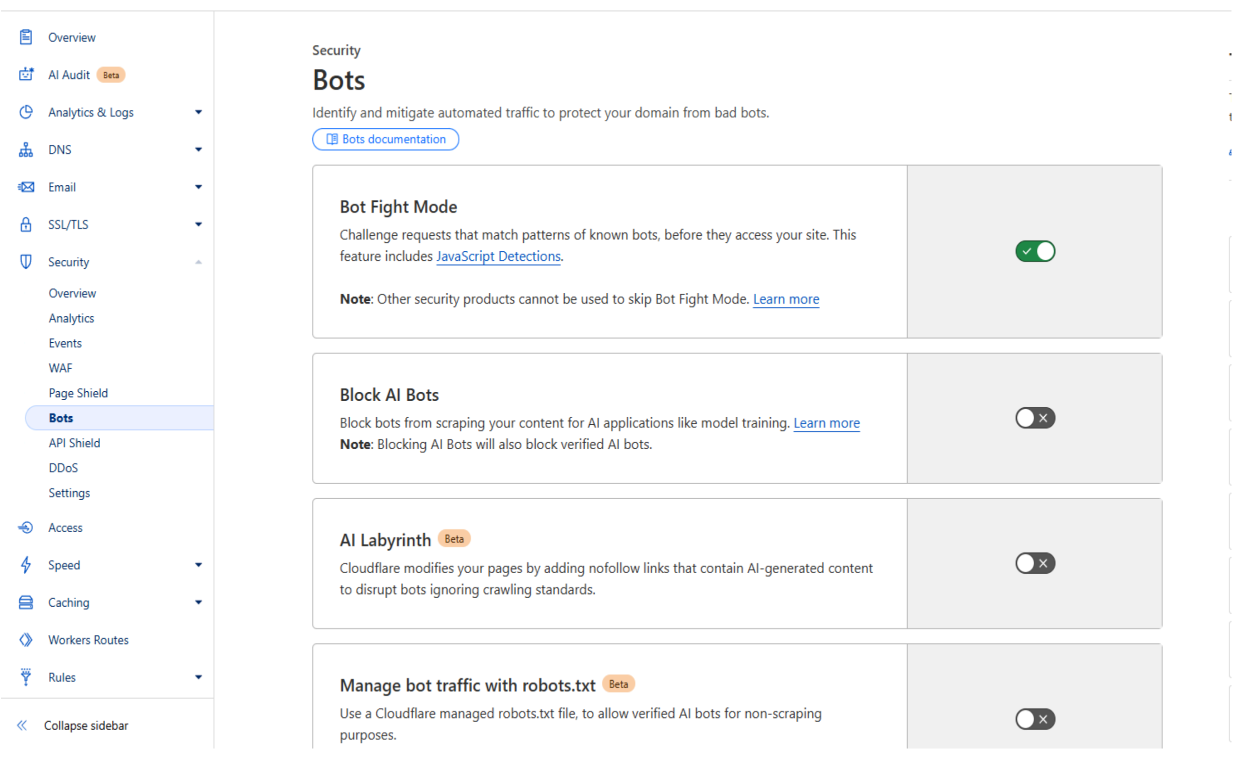


Рисунок 3.4. – Налаштування захисту від ботів за допомогою Cloudflare

### 3.6 Тестування застосованих засобів безпеки

Після впровадження комплексу заходів інформаційної безпеки у вебчаті, зокрема механізмів валідації, фільтрації, захисту від ін’єкцій, міжсайтових атак, автоматизованих ботів і обмеження доступу до API, виникає необхідність оцінити ефективність вжитих рішень на практиці. З цією метою було обрано автоматизований інструмент OWASP ZAP (Zed Attack Proxy) [12] — один із найпоширеніших і авторитетних засобів для динамічного тестування безпеки вебдодатків. Його основною перевагою є здатність симулювати дії зловмисника та виявляти найпоширеніші типи вразливостей відповідно до специфікацій OWASP Top 10.

Під час тестування вебчат було розгорнуто в тестовому середовищі, а сам процес відбувався у режимі активного сканування, де ZAP спочатку здійснює

автоматичну навігацію по всіх доступних маршрутах, викликає доступні форми, API-запити, ідентифікує точки введення, а потім намагається здійснити типові атаки: SQL-ін’єкції, XSS, CSRF, directory traversal, виявлення доступних директорій, обхід автентифікації, ін’єкції в параметри HTTP-заголовків та інші. Для кращої точності результатів попередньо було виконано ручне проксіювання трафіку між браузером та сервером, що дозволило зафіксувати автентичні сценарії взаємодії користувача з додатком.

Результати сканування продемонстрували відсутність критичних вразливостей у системі. Зокрема, ZAP не виявив можливості ін’єкції SQL-коду завдяки використанню підготовлених виразів (prepared statements) у всіх запитах до бази даних PostgreSQL. Тестування на XSS-атаки не принесло позитивного результату, оскільки всі вхідні дані проходять серверну валідацію, а на етапі виводу застосовується HTML-екранування. Виявлені потенційно небезпечні HTTP-заголовки, такі як відсутність X-Frame-Options чи Content-Security-Policy, були усунуті ще на етапі налаштування сервера Nginx і фреймворку Django.

Додатково, модуль API захищено аутентифікацією через JWT-токени, а всі запити до API проходять перевірку на допустимість методів і параметрів. Усі API маршрути були протестовані на наявність можливості обходу авторизації та ескалації привілеїв — подібних випадків виявлено не було. Інструмент також показав ефективність засобів, спрямованих на протидію автоматизованим запитам: Cloudflare Bot Management обмежував частоту звернень та блокував скриптову активність. Результат проведеного тестування зображено на рисунку 3.5.

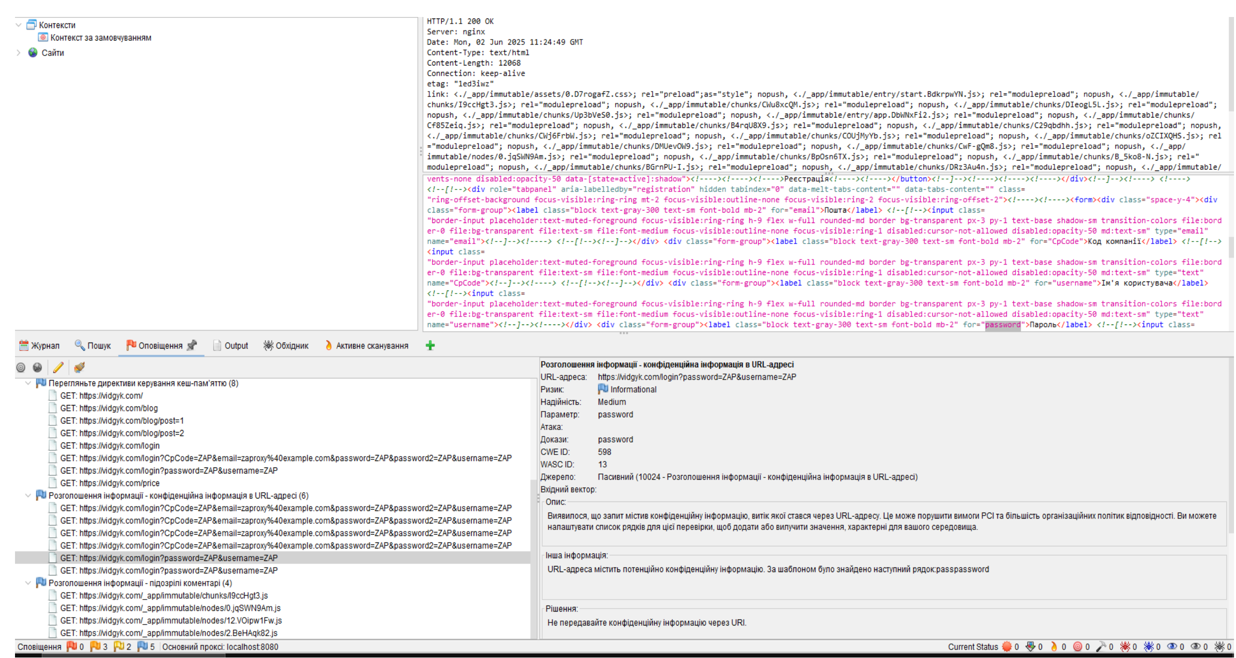


Рисунок 3.5. – Результат сканування вебдодатку за допомогою OWASP ZAP

### 3.7 Порівняння з існуючими аналогами

Порівнюючи запропоноване рішення з існуючими аналогами, слід відзначити, що більшість комерційних та open-source чат-систем орієнтовані насамперед на зручність установки і швидке отримання базового функціоналу, але водночас можуть мати обмежену можливість глибокого налаштування механізмів захисту та інтеграції з власною інфраструктурою організації. Наприклад, популярні хмарні сервіси LiveChat і Zendesk Chat надають зручний інтерфейс для користувачів і операторів, швидке розгортання та готові рішення з аналітикою й базовим захистом, але вони не дозволяють розміщувати дані на власних серверах замовника. Через це організаціям, які працюють із конфіденційною інформацією або обмежені політиками локального розгортання, доводиться шукати гібридні варіанти або розробляти власні доповнення. У той же час впроваджене рішення спирається на повністю кастомізовану архітектуру, де всі дані зберігаються в базі PostgreSQL, інфраструктура контролюється через Docker-контейнери, а Nginx із Cloudflare

гарантують захищений периметр. Тобто, навіть якщо порівнювати з хмарними аналогами, ми отримуємо повний контроль над середовищем, можливість глибокої кастомізації політик безпеки та індивідуальних правил обробки повідомлень, чого не можуть запропонувати «коробкові» рішення.

З боку open-source продуктів, наприклад, Rocket.Chat або Mattermost, можна розгорнути власний сервер із можливістю модифікації коду, але навіть вони найчастіше не включають готових механізмів багаторівневого захисту раніше налаштованих. Rocket.Chat забезпечує базову авторизацію, підтримку SSL та можливість встановлення WAF, однак для інтеграції з Cloudflare Bot Management, підготовленими SQL-виразами або детальною валідацією на рівні Django необхідно створювати власні плагіни або глибоко модифікувати ядро. У розробленому ж рішенні архітектура з самого початку побудована навколо Django зі вбудованими механізмами CSRF, XSS-фільтрації, підготовлених SQL-виразів та багаторівневого аудиту подій, що робить систему готовою до використання без додаткового переписування модулів безпеки. У підсумку ми отримали більш консистентний підхід, коли кожен компонент — від фронтенду у React до бекенду на Django Channels і Celery — уже спроєктований з урахуванням усіх ризиків.

Що стосується продуктивності, у хмарних SaaS-рішеннях вона часто обмежена архітектурою провайдера, який надає доступ до попередньо налаштованих серверів. Оскільки представлена система розгортається у власному контейнеризованому середовищі, ми можемо динамічно масштабувати кількість Gunicorn-воркерів, налаштовувати буферизацію Nginx, регулювати обсяг пам’яті та CPU для кожного сервісу. Крім того, використання Redis для чергування повідомлень разом із асинхронним Django Channels забезпечує швидку взаємодію у режимі реального часу навіть при одночасній обробці сотень WebSocket-з’єднань.

У частині інтеграції з іншими системами розроблене рішення має суттєві переваги. Хмарні платформи, як правило, підтримують обмежений набір CRM або маркетингових інструментів, тоді як у представленій реалізації webhook-и

забезпечують двосторонній зв’язок із будь-яким зовнішнім сервісом через REST. Це дозволяє передавати дані у внутрішню CRM, запускати бізнес-логіку або повідомляти сторонні сервіси про зміну статусу звернень в чаті. При цьому ролі й права доступу детально налаштовуються через RBAC у Django, а адміністратор може додавати нові інтеграційні точки без перезапуску системи, що неможливо або вкрай обмежено у готових рішеннях.

З позиції безпеки, хоча деякі open-source проекти уже впроваджують базові механізми, такі як LDAP-автентифікація або базову валідацію даних, саме комплексний підхід, який ми реалізували, дозволяє не лише блокувати підозрілий трафік на рівні Cloudflare та Nginx, а й проводити кореляцію інцидентів за допомогою SIEM. Зібрані логи передачі повідомлень, події входу, зміни облікових записів і аномальні навантаження аналізуються в режимі реального часу, що дає змогу не тільки виявити, а й автоматично нейтралізувати атаки. Інші платформи, навіть якщо мають інтерфейс логування, не завжди підтримують інтеграцію зі SIEM або системами моніторингу без великих доопрацювань.

Отже, порівняно з найближчими аналогами розроблене рішення вирізняється поєднанням високого рівня безпеки, масштабованості, повного контролю над інфраструктурою та глибокої кастомізації. Візуальне представлення порівняння з існуючими аналогами можна побачити на таблиці 3.1

*Таблиця 3.1*

Порівняльна таблиця існуючих аналогів і розробленого рішення

| **Критерій** | **LiveChat / Zendesk Chat (Cloud)** | **Rocket.Chat (Open Source)** | **Розроблений вебчат** |
|---|---|---|---|
| **Надання послуги та розгортання** | Хмарне рішення, повністю кероване провайдером. Користувач не володіє кодом і інфраструктурою. | Self-hosted або Cloud-варіант. Вимагає встановлення та адміністрування на власних серверах. | Self-hosted через Docker (PostgreSQL, Redis, Django, Nginx). Повний контроль над середовищем, можливість встановити на власній інфраструктурі. |
| **Вартість** | Фіксована щомісячна плата за користувача або за кількість операторів; безкоштовний пробний період. | Безкоштовно (AGPL-ліцензія); додаткові витрати на підтримку та хостинг. | Безкоштовно, витрати лише на власну інфраструктуру та обслуговування. |

*Продовження табл. 3.1*

| | | | |
|---|---|---|---|
| **Масштабованість** | Автоматичне масштабування в хмарі провайдера, необмежена кількість користувачів. | Масштабування залежить від ресурсів власного сервера. | Горизонтальне масштабування через Docker Compose / Kubernetes: додавання нових контейнерів Gunicorn, Celery, Channels. |
| **Функціональні можливості** | Повний набір готових інтеграцій (CRM, аналітика), готові шаблони, AI-боти, історія чатів. | Широкий функціонал, плагіни, підтримка групових чатів, відео-дзвінків (через додатки). | Мінімальний базовий функціонал, заточений під технічну підтримку: WebSocket, вкладення, push-сповіщення, інтеграція через webhook. |

*Продовження табл. 3.1*

| | | | |
|---|---|---|---|
| **Безпека каналів зв’язку** | TLS/HTTPS, WSS, захищені сертифікатами провайдера; додаткові WAF провайдера. | HTTPS забезпечується власноруч, можливість налаштування WSS; залежить від конфігурації. | TLS/HTTPS і WSS налаштовуються через Nginx; сертифікація автоматично через Let’s Encrypt або власні сертифікати. |
| **Захист від ін’єкцій (SQL, XSS)** | Реалізовано на стороні хмарного сервера, але детальна настройка недоступна клієнту. | Використовує власні вбудовані механізми валідації за замовчуванням, але тонкі налаштування – через плагіни. | Підготовлені SQL-вирази в Django ORM, контекстне екранування шаблонів, CSP. Усі запити проходять жорстку серверну валідацію через DRF. |
| **CSRF-захист** | Вбудований у хмарне рішення, але налаштування користувачем обмежені. | Можна активувати CSRF-middleware, але залежить від конфігурації вебсервера. | Від початку реалізовано CSRF-middleware Django; усі форми обов’язково містять CSRF-токен. |

*Продовження табл. 3.1*

| | | | |
|---|---|---|---|
| **Захист від ботів і спаму** | Bot Management у складі хмарного WAF, інтегровано без додаткової конфігурації. | Підтримка CAPTCHA-плагінів, але базових механізмів бот-фільтрації мало. | Інтегровано з Cloudflare Bot Management та Bot Fight Mode; реалізовані власні правила rate-limiting в Nginx; CAPTCHA можна додати в React. |
| **Захист від DDoS** | Повний захист на рівні інфраструктури провайдера; автоматичне визначення та блокування. | Базовий захист через налаштування серверного фаєрволу; необхідні додаткові інструменти. | Активація DDoS-захисту через Cloudflare, налаштування rate limits у Nginx, перевірка Cloudflare Challenge. |
| **Інтеграція з CRM / API** | Багато готових інтеграцій (Salesforce, Zendesk, HubSpot). | Наявні плагіни, але обмеженість лише за умови додаткового налаштування. | REST/WebSocket API, webhook-інтеграція під будь-яку CRM. |

*Продовження табл. 3.1*

| | | | |
|---|---|---|---|
| **Логування і моніторинг** | Вбудована аналітика, журнали доступні у хмарній панелі. | Базове логування, потребує інтеграції з зовнішніми SIEM/ELK. | Центральне логування через ELK/Graylog; інтеграція з SIEM; підтримка кореляції подій, real-time оповіщення. |
| **Контроль даних** | Дані зберігаються на серверах провайдера, можливий експорт, але без повного контролю. | Дані зберігаються локально або на виділеному сервері; користувач має доступ до сховища. | Повний контроль над даними в власній базі PostgreSQL; можливість шифрувати дані на рівні БД; збереження у власному сховищі. |
| **Гнучкість та кастомізація** | Обмежене налаштування: тема, чат-віджети, автоматичні відповіді. | Висока можливість кастомізації коду. | Повне налаштування архітектури: можна змінювати будь-який модуль. |

**Висновки за розділом 3:**

У результаті розробки даного розділу було сформовано цілісне уявлення про технічне оформлення та практичну реалізацію захисних механізмів у вебчаті. Починаючи з вибору стеку інструментів, ми обґрунтували переваги використання Django як основного серверного фреймворку разом із Celery/Redis для організації фонової обробки та PostgreSQL для надійного збереження повідомлень. На клієнтському боці React дозволив створити гнучкий і динамічний інтерфейс, а поєднання Gunicorn із Nginx забезпечило ефективне обслуговування запитів і розвантаження сервера за рахунок SSL-термінування та кешування статичних файлів. Така комбінація технологій створила міцну основу для побудови модулю вебчату, що легко інтегрується у вже наявну архітектуру сайту, і забезпечує достатню продуктивність навіть за високого навантаження.

Реалізація самого функціоналу вебчату показала, як за допомогою Django Channels і WebSocket можна організувати справді «життєвий» діалог із мінімальними затримками. Кожен етап обміну повідомленнями — від введення тексту й завантаження вкладень до їхньої доставки та відображення — отримав власну логіку валідації, збереження в базі та розсилки через асинхронні воркери Celery. Архітектура, заснована на Redis як посереднику для черг, дозволила чітко розділити потоки даних: синхронні HTTP-запити й асинхронну обробку мультимедіа. Це дало змогу не затримувати основний потік під час конвертації чи шифрування файлів, а також реалізувати механізми автоматичного відновлення після розриву з’єднання, зберігаючи чернетки повідомлень у локальному сховищі браузера.

Впровадження методів захисту у процеси обміну даними дозволило створити декілька незалежних рівнів фільтрації та контролю. Усі запити виконуються тільки через захищене з’єднання TLS/HTTPS, а WebSocket-сесії захищаються WSS-протоколом із перевіркою сертифікатів. Серверна валідація та ORM-запити із підготовленими виразами виключають найпоширеніші загрози SQL-ін’єкцій, а контекстне екранування й політики CSP забезпечують

надійний захист від XSS. Навіть якщо зловмисник спробує обійти один із фільтрів, наступний шар — чи то підготовлені SQL-вирази, чи то контекстне ескейпування на рівні шаблонів — блокує небажаний контент, завдяки чому система залишається стійкою до ін’єкцій.

Особливу увагу було приділено механізмам валідації, фільтрації та захисту від ін’єкцій. Завдяки поєднанню серверної перевірки вхідних даних у Django REST Framework із клієнтським екрануванням у React виключено можливість передачі до бази або відображення у браузері шкідливих фрагментів. Застосування регулярних виразів для перевірки форматів ключових полів (email, текстових коментарів, параметрів API) та відмова від конкатенації рядків у запитах гарантують, що навіть за наявності незвичайного вмісту жоден запит не буде інтерпретовано як частину SQL-команди. Крім того, у разі необхідності валідація даних виконується також на рівні Celery-воркерів перед обробкою фонового контенту, що підвищує загальну безпеку всієї системи.

Реалізація захисту від автоматизованих загроз завершила логіку побудови комплексного методу безпеки. Інструменти Cloudflare Bot Management та Bot Fight Mode, інтегровані в систему через проксі Nginx, забезпечили фільтрацію бот-активності ще до того, як запит потрапить до бекенду. Запити, які не проходять перевірку поведінкових шаблонів і JavaScript-челенджів, блокуються автоматично, що запобігає масовому скануванню API. Також, для внутрішніх систем або довірених клієнтів, налаштовано списки виключень, щоб уникнути помилкового блокування легітимних запитів. Завдяки моніторингу статистики бот-активності в реальному часі адміністратори можуть коригувати правила фільтрації, що дозволяє системі адаптуватися до нових загроз і підтримувати високу доступність чату для реальних користувачів.

# ВИСНОВКИ

Серед ключових переваг запропонованого комплексного методу захисту вебчатів слід виділити:

- підвищення стійкості до кіберзагроз завдяки багаторівневому захисту (від транспортного шифрування до валідації вхідних даних)
- зменшення ризиків витоку конфіденційних даних через автоматичне блокування ін'єкцій (XSS/SQL) та механізм авторизації
- суттєво уповільнюється та блокується автоматизована ворожа активність (боти, спам або DDoS-атаки) завдяки інтеграції з Cloudflare та налаштованим правилам захисту.

У першому розділі визначено об'єкти захисту вебчатів, проаналізовано архітектурні вразливості та класифіковано інформаційні активи. На основі моделювання загроз встановлено пріоритетні вектори атак, що заклало фундамент для розробки захисних механізмів.

У другому розділі систематизовано механізми захисту: від аутентифікації на основі JWT-токенів до моніторингу логів. Запропоновано комбінацію методів TLS/WSS, CSP, підготовлені SQL-вирази, обмеження API, що утворює комплексний захист, здатний нейтралізувати навіть загрози нульового дня.

У третьому розділі реалізовано практичне рішення на стеку Django/React/PostgreSQL. Впроваджено методи захисту:

- Впроваджено багатоетапну валідацію, що значно знизила ризик ін'єкцій до нуля.
- Інтегровано захист від ботів через Cloudflare Bot Management, який автоматично блокує більшість спам-атак.

Перспективи розвитку:

- Додати ML-аналіз поведінки користувачів для виявлення соціальної інженерії.

- Впровадити підтримку quantum-safe криптографії для захисту від квантових атак.
- Розширити систему моніторингу інтеграцією з Threat Intelligence Platform.

Загалом виконана робота засвідчила, що багаторівневий підхід до безпеки вебчату, гарантує стійкість до найрізноманітніших загроз. В результаті сформовано універсальний прототип, який можна адаптувати під різні бізнес-вимоги, інтегрувати в існуючі платформи та масштабувати залежно від обсягів трафіку й навантаження. Якісна захищеність вебчату сприяє підвищенню довіри користувачів і гарантує стабільну роботу сервісу при постійних викликах сьогодення.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. МАТЕРІАЛИ XVI-ої МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ «FREE AND OPEN SOURCE SOFTWARE» 13-14 лютого 2025 р. [Електронний ресурс] – Режим доступу до ресурсу: https://repository.hneu.edu.ua/bitstream/123456789/35624/1/foss-2025-theses.pdf С. 73
2. Сайт CERT-UA. Рекомендації CERT-UA з безпеки вебресурсів.[Електронний ресурс] – Режим доступу до ресурсу: https://cert.gov.ua/recommendation/19
3. Сайт OWASP. Рекомендації щодо безпеки вебдодатків. [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-project-top-ten/
4. Content Security Policy (CSP) [Електронний ресурс] – Режим доступу до ресурсу: https://content-security-policy.com/
5. Сайт OWASP Рекомендації щодо обмеження доступу до API . [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-project-api-security/
6. Сайт Cloudflare. Документація щодо використання WAF [Електронний ресурс] – Режим доступу до ресурсу:https://developers.cloudflare.com/waf/
7. PostgreSQL Official Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs
8. Django Documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.djangoproject.com
9. React Documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://react.dev/reference/react
10. Сайт Cloudflare. Документація щодо використання DDoS Protection [Електронний ресурс] – Режим доступу до ресурсу:https://developers.cloudflare.com/ddos-protection/

11. Сайт Cloudflare. Документація щодо використання захисту від ботів [Електронний ресурс] – Режим доступу до ресурсу: https://developers.cloudflare.com/bots/

12. OWASP Zed Attack proxy documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.zaproxy.org/docs/

13. CORS Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

14. MITRE ATT&CK [Електронний ресурс] – Режим доступу до ресурсу: https://attack.mitre.org/

15. Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org/

16. SSL/TLS in Detail [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc785811(v=ws.10)

17. Сайт WebSocket. Integrate with Django [Електронний ресурс] – Режим доступу до ресурсу: https://websockets.readthedocs.io/en/stable/howto/django.html

18. The WebSocket API [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

19. Національний стандарт України ДСТУ ISO/IEC 27001:2023 "Інформаційні технології. Методи захисту. Системи управління інформаційною безпекою".

20. Cloudflare's 2024 Q3 DDoS report.[Електронний ресурс] – Режим доступу до ресурсу: https://blog.cloudflare.com/ddos-threat-report-for-2024-q3

21. KrLabs. Захист вебдодатків на аудит безпеки [Електронний ресурс] – Режим доступу до ресурсу: https://kr-labs.com.ua/service/cybersecurity/kiberzahyst-sajtiv-i-web-dodatkiv-audyt-kiberbezpeky/

22. The Use of Chatbots in Digital Business Transformation: A Systematic Literature Review / Andrej Miklosik, Nina Evans, Athar Mahmood Ahmed Qureshi [Електронний ресурс] – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/9500127

23. Celery Project. *Celery Documentation*. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.celeryproject.org/en/stable/

24. Розшифровка мережевого трафіку: Поглиблений аналіз протоколів [Електронний ресурс] – Режим доступу до ресурсу: https://hackyourmom.com/osvita/chastyna-5-zlom-merezhi-analiz-merezhevyh-protokoliv/

25. Let's Encrypt. Automated Certificate Authority. [Електронний ресурс] – Режим доступу до ресурсу: https://letsencrypt.org/docs/

26. Prometheus. PostgreSQL Monitoring Guide. [Електронний ресурс] – Режим доступу до ресурсу: https://rezakhademix.medium.com/a-complete-guide-to-monitor-postgresql-with-prometheus-and-grafana-5611af229882

27. pgBackRest. Reliable Backup & Restore for PostgreSQL. [Електронний ресурс] – Режим доступу до ресурсу: https://pgbackrest.org/

28. The Ultimate Guide to API Best Practices [Електронний ресурс] – Режим доступу до ресурсу: https://nordicapis.com/the-ultimate-guide-to-api-best-practices/

29. Zendesk documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://support.zendesk.com/hc/en-us/categories/6191839589274

30. LiveChat Documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://www.livechat.com/features/

31. AWS API Gateway Security Best Practices. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/apigateway/latest/developerguide/security.html

32. JSON Web Token (JWT) RFC. [Електронний ресурс] – Режим доступу до ресурсу: https://datatracker.ietf.org/doc/html/rfc7519

33. OWASP Cheat Sheet: CSRF Prevention. [Електронний ресурс] – Режим доступу до ресурсу: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

34.OWASP Cheat Sheet: XSS Prevention. [Електронний ресурс] – Режим доступу до ресурсу: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

35.SALT. State of API Security Report 2025. [Електронний ресурс] – Режим доступу до ресурсу: https://content.salt.security/state-api-report.html

**Додаток А**

**Фрагмент вихідного коду програми**

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Customer Support Chat</title>
                                                                                                                            <link
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css"
rel="stylesheet">
   <script src="https://cdn.tailwindcss.com"></script>
   <script>
      tailwind.config = {
         theme: {
            extend: {
               colors: {
                  indigo: {
                     50: '#eef2ff',
                     100: '#e0e7ff',
                     200: '#c7d2fe',
                     300: '#a5b4fc',
                     400: '#818cf8',
                     500: '#6366f1',
                     600: '#4f46e5',
                     700: '#4338ca',
                     800: '#3730a3',
                     900: '#312e81',
                  }
               },
               animation: {
                  'bounce-slow': 'bounce-slow 1.5s infinite',
                  'fade-in': 'fadeIn 0.3s ease-in'
               },
               keyframes: {
                  'bounce-slow': {
```

```
                '0%, 100%': { transform: 'translateY(0)' },
```

**Продовження додатку А**

```
              'fadeIn': {
                '0%': { opacity: 0, transform: 'translateY(10px)' },
                '100%': { opacity: 1, transform: 'translateY(0)' },
              }
            }
          }
        }
      }
    </script>
</head>
<body class="bg-gray-100 min-h-screen flex items-center justify-center p-4">
    <div class="max-w-4xl w-full bg-white rounded-xl shadow-xl overflow-hidden">
        <!-- Header -->
        <header class="bg-gradient-to-r from-indigo-600 to-indigo-800 text-white py-10 px-6 text-center">
            <p class="text-indigo-100 max-w-2xl mx-auto text-lg">Our team is ready to assist you 24/7. Click the chat button to get started!</p>
        </header>

        <!-- Main Content -->
        <div class="p-6 md:p-8">
            <div class="text-center mb-12">
                <h2 class="text-2xl md:text-3xl font-bold text-indigo-700 mb-4">How Can We Help You Today?</h2>
                    <p class="text-gray-600 max-w-2xl mx-auto">We're committed to providing exceptional customer service. Our support team is available around the clock to answer your questions and resolve any issues.</p>
            </div>
            <!-- Features -->
            <div class="grid grid-cols-1 md:grid-cols-3 gap-6 mb-12">
                <div class="bg-white border border-indigo-100 rounded-xl p-6 shadow-sm hover:shadow-md transition-shadow">
                    <div class="w-14 h-14 bg-indigo-100 rounded-full flex items-center justify-center mb-4">
                        <i class="fas fa-headset text-indigo-600 text-2xl"></i>
```