МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії

програмного забезпечення

_____ Євген ДАВИДЕНКО

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ЗАСТОСУНОК КЕРУВАННЯ ДІСКОРД-БОТОМ

Спеціальність 121 Інженерія програмного забезпечення

Освітня програма «Інженерія програмного забезпечення»

Здобувач _____ **Олег САДОВСЬКИЙ**

«__» _____ 2026 р.

Керівник роботи

PhD, ст. викладач

_____ **Ігор КАНДИБА**

«__» _____ 2026 р.

Миколаїв – 2026

Завдання на виконання кваліфікаційної роботи

Чорноморський національний університет імені Петра Могили

Факультет	Комп'ютерних наук
Кафедра	Інженерії програмного забезпечення
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступінь	Бакалавр
Спеціальність	121 Інженерія програмного забезпечення
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інженерії
програмного забезпечення
_____ Євген ДАВИДЕНКО
«___» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну бакалаврську роботу здобувача

Садовський Олег

1. Тема кваліфікаційної роботи «Застоунок керування діскорд-ботом» затверджена наказом ректора ЧНУ ім. Петра Могили № 349 від «26» грудня 2026 р.
2. Строк представлення кваліфікаційної роботи «___» червня 2026 р.
3. Очікуваний результат роботи та початкові дані якщо такі потрібні.

Розроблений застосунок керування діскорд-ботом.

4. Перелік питань, що підлягають розробці:

- дослідження існуючих аналогів;
- формування специфікації вимог до застосунку, що розробляється;
- проєктування застосунку керування діскорд ботом;
- розробити інструменти для налаштування та модерації сервера

адміністратором;

– розробити зручний вебінтерфейс для керування функціоналом бота з використанням фреймворку Laravel;

– реалізувати взаємодію між ботом та вебінтерфейсом;

– тестування розробленого застосунку.

5. Перелік графічних матеріалів: Презентація.

6. Консультанти:

Консультант	**Кафедра (організація)**	**Частина роботи**

Дата видачі завдання «__26__» _____грудня_____ 2025 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: **«Застосунок керування діскорд-ботом»**

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КБР	25.12.2025	26.12.2026	*Виконано*
2.	Огляд літератури за темою роботи	12.01.2026	16.01.2026	*Виконано*
3.	Аналіз предметної області	19.01.2026	23.01.2026	*Виконано*
4.	Дослідження існтрументарію	26.01.2026	06.02.2026	*Виконано*
6.	Моделювання та конструювання ПЗ	09.02.2026	13.02.2026	*Виконано*
7.	Розробка проєктних рішень	16.02.2026	20.02.2026	*Виконано*
8.	Реалізація бота та його модулів	30.02.2026	27.03.2026	*Виконано*
9.	Створення вебпорталу керування ботом на основі фреймовку “Laravel”	30.03.2026	24.04.2026	*Виконано*
10.	Інтеграція бота з вебпорталом	27.04.2026	30.04.2026	*Виконано*
11.	Тестування роботи бота	01.05.2026	06.05.2026	*Виконано*
12.	Відгук керівника КБР	07.05.2026	08.05.2026	
13.	Оформлення КБР та презентації	11.05.2026	22.05.2026	*Виконано*
14.	Попередній захист	27.05.2026	27.05.2026	*Виконано*
15.	Рецензування	12.06.2026	15.06.2026	
16.	Завершення оформлення КБР та презентації	11.06.2026	12.06.2026	*Виконано*
17.	Захист кваліфікаційної роботи	22.06.2026	23.06.2026	

Здобувач _____ **Олег САДОВСЬКИЙ**

«__» _____ 2026 р.

Керівник роботи
PhD, ст. викладач _____ **Ігор КАНДИБА**

«__» _____ 2026 р.

АНОТАЦІЯ

до кваліфікаційної бакалаврської роботи

« Застосунок керування діскорд-ботом»

Здобувач 409 гр.: Садовський Олег

Керівник: PhD, ст. викладач Кандиба Ігор

У сучасному цифровому середовищі онлайн-комунікації стали ключовим інструментом взаємодії, а платформи на кшталт Discord забезпечують миттєвий, безкоштовний і якісний обмін інформацією, особливо для великих спільнот геймерів. Із зростанням аудиторії ручне адміністрування серверів ускладнюється, тому автоматизація через ботів, які модерують контент, керують ролями та забезпечують розважальні функції, стала необхідною і перетворилася на центральний елемент функціонування серверів.

Попри наявність готових рішень, власники серверів прагнуть мати унікальний інструмент, адаптований під специфічні потреби аудиторії. Розробка універсального бота на базі бібліотеки Disnake у поєднанні з вебпорталом на Laravel дозволяє забезпечити асинхронну обробку подій, стабільну роботу при високих навантаженнях та гнучке розширення функціоналу через модулі модерації, економіки та логування.

Таким чином, створення універсального Discord-бота є актуальним завданням, яке поєднує автоматизацію рутинних процесів, зручність керування та масштабованість, відповідаючи потребам сучасних онлайн-спільнот.

Мета роботи: Розробка застосунку адміністрування боту Діскорд для спрощення керування сервером.

Об’єктом кваліфікаційної роботи є процес керування сервером Діскорд за допомогою боту.

Предметом кваліфікаційної роботи є методи та засоби розробки застосунку керування ботом Діскорд.

Кваліфікаційна робота складається із вступу, 4 розділів:

У вступі обґрунтовано актуальність розробки застосунку для керування діскорд-ботом, визначено мету та основні завдання роботи, окреслено об’єкт і предмет дослідження

У першому розділі проведено дослідження предметної області, проаналізовано існуючі аналоги для управління Discord-серверами та обгрунтовано вибір технологічного стеку – бібліотеки Disnake для створення бота та фреймворку Laravel для вебпорталу адміністрування. Розглянуто переваги асинхронної обробки подій, модульної архітектури та інтеграції з вебінтерфейсом для зручного управління функціоналом.

Другий розділ присвячено проєктуванню системи: розробці архітектури бота та вебпорталу, побудові діаграм класів і розгортанню застосунку, а також організації взаємодії між ботом і вебінтерфейсом.

У третьому розділі описано процес проєктування застосунку, опис клієнт-серверної взаємодії, розробка дизайну архітектури, створення сценаріїв використання та діаграм для налаштування серверів, модерації контенту та управління ролями через вебінтерфейс.

У четвертому розділі наведено процес програмної реалізації застосунку, створення клієнт-серверної взаємодії, розробку інструментів для налаштування серверів, модерації контенту та управління ролями, описано результати тестування бота та вебпорталу, перевірку стабільності роботи під навантаженням і зручності користувацького інтерфейсу.

У висновках узагальнено результати виконаної роботи, підтверджено досягнення поставленої мети та завдань, а також окреслено можливості подальшого розширення функціоналу та впровадження нових модулів для ефективного керування Discord-серверами.

КРБ викладена на 78 сторінок, вона містить 4 розділи, 14 ілюстрацій, 8 таблиць, 33 джерел в переліку посилань.

Ключові слова: Discord Bot, Діскорд Бот, Disnake, Discord, Bot, Бот, Діскорд, Laravel, Модерація, Керування сервером, Управління сервером, Адміністрування сервера.

ABSTRACT

of the Bachelor’s Thesis

“Discord bot management application”

Student of group 408: Sadovskyi Oleh

Supervisor: PhD, Senior Lecturer Kandyba Ihor

In today's digital environment, online communication has become a key tool for interaction, and platforms such as Discord provide instant, free, and high-quality information exchange, especially for large gaming communities. As the audience grows, manual server administration becomes more difficult, so automation through bots that moderate content, manage roles, and provide entertainment features has become necessary and has become a central element of server operation.

Despite the availability of ready-made solutions, server owners strive to have a unique tool tailored to the specific needs of their audience. The development of a universal bot based on the Disnake library in combination with a web portal on Laravel allows for asynchronous event processing, stable operation under high loads, and flexible functionality expansion through moderation, economy, and logging modules.

Thus, the creation of a universal Discord bot is a relevant task that combines the automation of routine processes, ease of management, and scalability, meeting the needs of modern online communities.

The subject of the qualification work is the process of managing a Discord server using a bot.

The subject of the qualification work is the methods and means of developing a Discord bot management application.

The qualification work consists of an introduction, 4 chapters:

The introduction justifies the relevance of developing an application for managing a Discord bot, defines the purpose and main tasks of the work, and outlines the object and subject of the study.

The first chapter analyses the subject area, reviews existing software solutions for managing Discord servers, and justifies the choice of the technology stack – the Disnake library for creating bots and the Laravel framework for the administration web portal. The

advantages of asynchronous event processing, modular architecture, and integration with a web interface for convenient functionality management are considered.

The second section is devoted to system design: developing the architecture of the bot and web portal, building class diagrams and deploying the application, as well as organising the interaction between the bot and the web interface. Particular attention is paid to the modular structure for supporting moderation, economy, logging and other functions.

The third section describes the process of software implementation of the application, creation of client-server interaction, development of tools for configuring servers, content moderation, and role management via the web interface.

The fourth chapter presents the results of testing the bot and web portal, checking the stability of operation under load and the convenience of the user interface.

The conclusions summarise the results of the work performed, confirm the achievement of the set goals and objectives, and outline opportunities for further expansion of functionality and implementation of new modules for effective management of Discord servers.

The BQW is presented on 78 pages, it contains 4 sections, 14 illustrations, 8 tables, 33 sources in the list of references.

Keywords: Discord Bot, Disnake, Discord, Bot, Laravel, Moderation, Server Management, Server Administration.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД – База Даних

ПЗ – Програмне Забезпечення

СКБД – Система Керування Базами Даних

API – Application Programming Interface

CORS – Cross-Origin Resource Sharing

CSRF – Cross-Site Request Forgery

HTTPs – Hypertext Transfer Protocol Secure

IDE – Integrated Development Environment

LINQ – Language Integrated Query

OAuth2 – Open Authorization 2.0

ORM – Object-Relational Mapping

RBAC – Role-Based Access Control

REST – REpresentational State Transfer

SPA – Single Page Application

UI – User Interface

XP – eXperience Points

ВСТУП

У сучасному цифровому середовищі онлайн-комунікації відіграють ключову роль у формуванні спільнот, управлінні процесами та забезпеченні ефективної взаємодії. Якщо ще два десятиліття тому зв’язок із колегами чи родичами з-за кордону вимагав значних фінансових витрат і технічних зусиль, то сьогодні цифрові платформи надають можливість спілкуватися миттєво, безплатно та з високою якістю мультимедіа.

Одним із провідних інструментів для такої комунікації став “Discord” – застосунок, що спочатку набув популярності як більш сучасний та зручний аналог “Skype”.

В умовах великих спільнот, ручне адміністрування стає малоефективним: людині фізично важко постійно відстежувати порушення правил, модерувати контент і підтримувати активність аудиторії. Це створює ризики як для безпеки спільноти, так і для її репутації.

Вирішенням проблеми масштабування стала автоматизація рутинних процесів через створення ботів. Боти дозволяють автоматично реагувати на порушення, фільтрувати неприйнятні повідомлення, керувати системою ролей та забезпечувати розважальні функції. Роль ботів трансформувалася з допоміжної у центральний елемент функціонування великих спільнот.

Метою кваліфікаційної роботи є розробка застосунку адміністрування боту Діскорд для спрощення керування сервером.

Для досягнення визначеної мети необхідно вирішити такі завдання:

– дослідження застосунків аналогів;

– формування специфікації вимог до застосунку, що розробляється;

– проєктування застосунку керування діскорд ботом;

– розробка інструменти для налаштування та модерації сервера адміністратором;

– розробка зручний вебінтерфейс для керування функціоналом бота з використанням фреймворку Laravel;

– реалізація взаємодію між ботом та вебінтерфейсом;
– тестування розробленого застосунку.

Робота виконана за допомогою бібліотеки “disnake” та фреймворку “Laravel”. Бібліотека “disnake” обрана через потужний асортимент інструментів для стабільної асинхронної роботи бота, підтримку модульності та зручну документацію. Використання “Laravel” для вебпорталу зумовлене необхідністю створення зручної панелі адміністратора, що дозволяє розширити функціонал системи та спростити взаємодію з нею без необхідності втручання в програмний код.

Об’єкт кваліфікаційної роботи виступає процес керування сервером Діскорд за допомогою боту.

Предмет кваліфікаційної роботи – методи та засоби розробки застосунку керування ботом Діскорд.

Попри наявність готових рішень, багато власників серверів прагнуть мати унікальний інструмент, адаптований під специфічні потреби їхньої аудиторії, без залежності від сторонніх сервісів. Окрім того, велика кількість доступних програмних рішень на ринку, має обмежений безкоштовний функціонал. Наприклад – MEE6, найпопулярніший бот для модерації, але з обмеженим безкоштовним функціоналом та недостатньо гнучким налаштуванням. Відповідно у сфері з’являється попит на створення нових рішень, за які власники готові платити. Відповіддю є розробка універсального бота на базі бібліотеки “Disnake” та вебпорталу керування на фреймворку “Laravel”.

Результатом роботи має стати продукт, який може бути застосований адміністраторами серверів «Діскорд», які потребують гнучкі та комплексні рішення для своєї спільноти. Застосунок має слугувати підвищенню ефективності та створення автоматизації управління сервером, а також надання можливостей зручного налаштування створеного функціоналу за допомогою зручного вебінтерфейсу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

У нас час платформи для обміну миттєвими повідомленнями – мессенджери, зокрема Discord, стали основним інструментом у формуванні масштабних цифрових спільнот.

Популярність подібних платформ зумовлена зручністю створення великих груп користувачів в єдиному місці – серверах, що значно ефективніше за створення розрізнених групових викликів. Проте стрімке зростання кількості користувачів, кількість яких на популярних серверах може сягати навіть не десятків, а сотень тисяч, призводить до виникнення складних проблем управління подібною спільнотою.

Процес керування великим сервером стикається з людським фактором – різні характери учасників, агресія, розповсюдження небажаного контенту або спам створюють несприятливі умови для учасників сервер. За таких умов традиційні методи модерації, що покладаються виключно на зусилля власника сервера чи обмеженого кола довірених осіб, втрачають ефективність.

Першочергово, виникає проблема масштабування – фізично неможливо цілодобового контролювати всі канали та всіх учасників. Друга проблема гостро полягає в питанні ієрархії та прав доступу. Складність налаштування кожної окремої ролі та ризики, пов'язані з надмірними правами адміністративного складу, створюють загрозу безпеці та стабільності сервера.

В нинішній час, вирішення проблеми адміністрування цифрових спільнот досягається шляхом автоматизації. Впровадження спеціалізованого програмного забезпечення – ботів, дозволяє перекласти ручні, рутинні і складні завдання з фільтрації повідомлень, моніторингу активності та керування технічними параметрами каналів на незалежну системи.

Загальна концепція такого автоматизованого інструментарію передбачає створення багаторівневої системи захисту, інструментів для підтримки активності аудиторії та механізмів гнучкого делегування повноважень. Це дозволяє

забезпечити комфортне середовище для користувачів та стабільне функціонування сервера незалежно від темпів його зростання.

Щоб реалізувати такий масштабний функціонал із забезпеченням високої продуктивності та зручним вебінтерфейсом, необхідно дослідити та обрати відповідні архітектурні рішення, алгоритми та фреймворки.

1.2 Аналіз технологічного стеку

Предмет дослідження кваліфікаційної роботи – методи та засоби розробки застосунку керування ботом Діскорд з розширеною функціональністю, зокрема з підтримкою музичних можливостей, а також вебінтерфейсу для його налаштування та керування.

Вибір технологічного стеку є одним із ключових рішень на етапі проєктування подібної системи, оскільки від нього залежить як продуктивність кінцевого продукту, так і зручність його підтримки та розширення в майбутньому.

Сучасний ринок бібліотек для взаємодії з Discord API є надзвичайно різноманітним, існують рішення для таких мов програмування як – C#, Java, Python, JavaScript, Go, Rust. Проте, незважаючи на широкий вибір, не кожна бібліотека однаковою мірою підходить для реалізації всіх запланованих функцій.

Реалізація музичного відтворення як одного з ключових компонентыв суттєво звужує перелік придатних технологій, оскільки більшість бібліотек або не підтримують роботу з голосовими функціями, або потребують значних додаткових налаштувань і залежностей, що ускладнює розробку та знижує стабільність системи.

Серед найпопулярніших та найбільш зрілих рішень для розробки Discord-ботів виділяються чотири основні варіанти – Python із бібліотеками discord.py [1] та disnake [2], JavaScript із бібліотекою discord.js [3], Java із бібліотекою JDA [4], а також C# із бібліотекою Discord.Net [5]. Кожен із зазначених варіантів має свою специфіку, свої сильні сторони та певні обмеження, що необхідно детально розглянути з метою обґрунтованого вибору технологічного стеку.

Python з discord.py та disnake. Python є однією з найпопулярніших мов програмування у світі, зокрема завдяки своєму лаконічному синтаксису та великій спільноті розробників. Бібліотека discord.py є однією з перших та найбільш відомих обгорток Discord API для Python. Disnake є її активно підтримуваним форком із підтримкою slash-команд.

До переваг цього підходу відносяться – низький поріг входження для початківців, велика кількість навчальних матеріалів та підручників, а також наявність спеціалізованих бібліотек для відтворення аудіо, зокрема yt-dlp у поєднанні з FFmpeg та бібліотека wavelink для підключення до Lavalink-сервера. Водночас Python має суттєвий недолік у вигляді Global Interpreter Lock, що обмежує ефективне використання багатопотоковості та може негативно позначитися на продуктивності бота за умов високого навантаження. Крім того, розгортання Python-застосунків потребує встановлення інтерпретатора та управління залежностями, а швидкодія мови поступається компільованим аналогам.

JavaScript з discord.js. Бібліотека discord.js є де-факто стандартом у розробці Discord-ботів на JavaScript та однією з найбільш активно підтримуваних у своєму класі. Вона підтримує весь спектр можливостей Discord API, включаючи slash-команди, компоненти, взаємодії та голосові з'єднання через пакети. Рішення відрізняється асинхронною, подієво-орієнтованою моделлю виконання, що є природньою для бота, який постійно слухає та реагує на події.

До переваг належать – стабільність, широка документація, велика екосистема npm та можливість використовувати одну мову як для бекенду бота, так і для вебінтерфейсу, що суттєво зменшує складність стеку. Основним недоліком є однопотокова природа Node.js, яка потребує обережного управління асинхронними операціями, та потенційні проблеми з обробкою обчислювально важких задач без використання Worker Threads.

Java з Java Discord API. JDA є перевіреною часом та стабільною бібліотекою для розробки Discord-ботів на платформі JVM. Java як мова програмування відзначається строгою статичною типізацією, високою продуктивністю та потужними вбудованими засобами для багатопотокового програмування, що

робить її придатною для побудови масштабованих та надійних систем. Для реалізації музичних функцій у поєднанні з JDA часто застосовується бібліотека LavaPlayer або підключення до зовнішнього Lavalink-сервера.

До переваг JDA відносяться – висока продуктивність виконання, сувора типізація, яка зменшує кількість помилок під час виконання, а також можливість використання Spring Boot для побудови вебсервера в межах єдиного проєкту. Проте Java є значно більш багатослівною мовою порівняно з Python або JavaScript, що збільшує обсяг шаблонного коду та уповільнює початковий етап розробки. Крім того, споживання пам'яті JVM-застосунків є суттєво вищим, що може бути критично за умов обмеженого або бюджетного хостингу.

C# з Discord.Net. Discord.Net – це масштабна, функціональна бібліотека для мови C# та платформи .NET, яка надає широкі можливості для взаємодії з Discord API, включаючи підтримку slash-команд, компонентів та голосових з'єднань. C# є мовою зі строгою статичною типізацією та розвиненою системою асинхронного програмування на основі механізму async та await.

Серед переваг варто виділити – відмінну підтримку з боку провідних IDE, зручний синтаксис LINQ для роботи з колекціями даних, а також можливість інтеграції з ASP.NET Core для розробки повноцінної вебпанелі в рамках одного рішення. Бібліотеки Victoria або Lavalink4NET дозволяють реалізувати музичну функціональність на достатньому рівні. До недоліків відносяться – менший розмір спільноти розробників ботів порівняно з discord.js або discord.py, що ускладнює пошук готових прикладів, а також менша поширеність .NET-екосистеми серед типових хостинг-провайдерів для ботів.

Окрім самого бота, невід'ємною складовою сучасного програмного продукту такого класу є зручний вебінтерфейс для його налаштування та керування.

Реалізація всіх конфігурацій виключно через текстові команди чату є неприйнятною з точки зору зручності використання. Вона вимагає від користувача знання синтаксису команд, унеможливлює відображення поточного стану налаштувань та є повільною і схильною до помилок.

Натомість сучасні Discord-боти надають вебсайти, де користувачі можуть авторизуватися через протокол OAuth2, додати бота на свій сервер та виконати детальне налаштування в інтуїтивно зрозумілому графічному інтерфейсі.

При виборі технологічного стеку для вебпанелі визначальним критерієм є підтримка взаємодії з Discord API і коректна обробка CORS-запитів між фронтендом та бекендом. На цей час найбільш поширеними архітектурними підходами є використання Laravel із Blade-шаблонами, комбінація Next.js із FastAPI, а також повністю JavaScript-орієнтований стек на базі NestJS [9] та React.

Laravel із Blade-шаблонами. Laravel [6] – це потужний PHP-фреймворк з вбудованою підтримкою маршрутизації, ORM, системою шаблонів Blade та засобами автентифікації, що суттєво спрощує розробку серверно-рендерованих вебзастосунків. Він добре підходить для проєктів, де логіка формування сторінок зосереджена на сервері.

До переваг відносяться – зрілість екосистеми, розгалужена документація, вбудовані засоби роботи з сесіями, CSRF-захистом, а також пакет Laravel Socialite для спрощеної OAuth2-інтеграції з Discord. Недоліком є менша інтерактивність інтерфейсу порівняно із SPA-підходами, а також необхідність налаштування PHP-середовища та вебсервера на хостингу.

Next.js із FastAPI. Комбінація Next.js [7] і FastAPI [8] є популярним вибором для проєктів, де бекенд вже написано на Python і де необхідний реактивний інтерфейс. Next.js забезпечує зручну серверну та клієнтську маршрутизацію, а FastAPI відрізняється надзвичайно високою продуктивністю серед Python-фреймворків, вбудованою валідацією даних через Pydantic та автоматичною генерацією OpenAPI-документації.

Перевагами є поєднання реактивного UI з потужним Python-бекендом і можливість повторного використання коду з Python-ботом. До недоліків належать – необхідність підтримувати два окремих серверних процеси та додаткова складність у налаштуванні CORS-політики між Next.js і FastAPI.

NestJS у комбінації з React. Підхід, що передбачає використання виключно JavaScript/TypeScript як на сервері, так і на клієнті. NestJS надає модульну

архітектуру з підтримкою Dependency Injection, Guards, Interceptors та вбудованого WebSocket, що робить його придатним для побудови складних і добре структурованих API.

Перевагами є єдина мова та єдина екосистема для всього стеку, що спрощує підтримку та онбординг розробників, висока продуктивність Node.js для I/O-навантажень, а також природна інтеграція з discord.js-ботом у межах єдиного монорепозиторію. До недоліків відносяться – більш крута крива навчання NestJS порівняно з Express, а також необхідність окремого розгортання або налаштування зворотного проксі між React-додатком та NestJS API.

Таким чином, предмет дослідження охоплює широкий спектр інструментів та технологій, між якими необхідно здійснити обґрунтований вибір з урахуванням конкретних функціональних вимог, умов розгортання та зручності розробки. Проведений аналіз наявних варіантів свідчить про те, що жоден із них не є універсально кращим. Кожен технологічний стек має свою оптимальну область застосування.

1.3 Аналіз засобів керування ботами Discord

На сучасному ринку ботів для Discord існує значна кількість готових рішень, що реалізують різноманітні функціональні можливості – від модерації спільноти до розважального контенту та музичного відтворення. З метою обґрунтування доцільності розробки застосунку та визначення його конкурентних переваг буде проведено аналіз існуючих аналогів.

Кожен із досліджених ботів орієнтований на певну цільову аудиторію та задовольняє конкретний сегмент потреб адміністраторів і учасників серверів. Водночас жоден із них не є комплексним рішенням, здатним об'єднати в собі повноцінну модерацію, розважальний функціонал, музичне відтворення та зручний вебінтерфейс управління. Цей факт і визначає актуальність розробки більш збалансованого застосунку.

Nekotina [10] – бот, орієнтований насамперед на розважальні та соціальні функції. Він надає широкий набір міні-ігор, систему внутрішньої економіки

сервера, а також інструменти соціальної взаємодії між учасниками – погладити, обійняти, привітати тощо. Значною перевагою є наявність системи рівнів та репутації, яка стимулює активність аудиторії, а також підтримка кількох мов інтерфейсу, що розширює потенційну аудиторію бота.

Проте Nekotina практично позбавлена інструментів адміністрування. Відсутність повноцінного вебінтерфейсу змушує адміністраторів виконувати всі налаштування виключно через текстові команди в чаті, що є незручним і схильним до помилок підходом. Крім того, бот не пропонує жодного музичного функціоналу, а засоби модерації зведені до мінімуму – без автоматичної фільтрації, системи попереджень чи детальних журналів дій.

Ripp y [11] позиціонується як спеціалізований музичний бот і демонструє у цій ніші достатньо сильні результати. Він підтримує відтворення аудіоконтенту з таких популярних платформ, як YouTube, Spotify та SoundCloud, забезпечує управління чергою відтворення та підтримує роботу з альбомами. Команди бота є відносно інтуїтивними, а функціонал пошуку треків безпосередньо з Discord спрощує взаємодію з музичним модулем.

Разом із тим надмірна спеціалізація Ripp y є одночасно й його головним недоліком. За межами музичного відтворення бот не пропонує практично нічого – ні модерації, ні управління ролями, ні системи рівнів, ні вебпанелі для налаштування. Також слід відзначити нестабільність роботи при великому навантаженні на кількох серверах, що є критичним обмеженням для масштабування.

Appy є прикладом бота з ширшим спектром функціоналу, який намагається охопити кілька напрямків одночасно. Бот підтримує slash-команди відповідно до сучасних стандартів DiscordAPI, реалізує вітання нових учасників та автоматичне призначення ролей, а також веде базовий журнал подій сервера. Важливою перевагою є наявність вебінтерфейсу для налаштування, хоча його можливості залишаються обмеженими.

До суттєвих вад Appy [12] слід віднести застарілий дизайн вебпанелі та неповний охват налаштувань через графічний інтерфейс – частину параметрів все

одно доводиться конфігурувати командами. Музичний модуль реалізований лише в базовому вигляді й поступається спеціалізованим аналогам. Нерегулярність оновлень та відсутність системи залучення аудиторії знижують привабливість бота для активних спільнот.

Groot [13] – бот, який позиціонує себе як інструмент для серйозного адміністрування Discord-серверів. Його модераційний функціонал є одним із найрозвиненіших серед розглянутих аналогів – автоматичний антиспам, фільтрація небажаних посилань і ненормативної лексики, гнучка система попереджень із автоматичним застосуванням санкцій, а також детальний журнал модераційних дій для адміністраторів. Ці характеристики роблять його придатним рішенням для управління великими та активними спільнотами.

Водночас вузька орієнтація Groot на модерацію призводить до відсутності низки важливих можливостей. Бот не підтримує музичного відтворення, не має системи рівнів чи механізмів залучення аудиторії, а вебпанель реалізована лише частково. Первинне налаштування бота є складним процесом, що вимагає від адміністратора певних технічних знань, що суттєво підвищує поріг входження.

Arcane [14] є лідером у своїй вузькій спеціалізації – системі рівнів та XP. Бот пропонує найрозвиненіший серед аналогів механізм нагородження за активність – кастомізовані карти рангів, ролі-нагороди за досягнення певних рівнів, гнучке налаштування повідомлень про підвищення та зрозумілий вебінтерфейс. Значне поширення на великих серверах свідчить про його надійність та стабільність у цьому напрямку.

Проте функціональна модель Arcane обмежується переважно системою рівнів. Музичний модуль відсутній, інструменти модерації є базовими та не передбачають автоматичної фільтрації контенту, а частина розширених налаштувань доступна лише за умови преміум-підписки. Таким чином, Arcane задовольняє лише одну з кількох потреб сучасного адміністратора сервера.

Проведений порівняльний аналіз наявних рішень дозволяє чітко визначити напрями, за якими застосунок потребує вдосконалення задля формування

конкурентоспроможного продукту, що перевершує кожного з розглянутих аналогів у сукупності характеристик.

По-перше, необхідно реалізувати повноцінний та сучасний вебінтерфейс управління. Жоден із проаналізованих ботів не надає адміністратору зручного, функціонально повного графічного інтерфейсу, що охоплює весь спектр налаштувань. Власна вебпанель повинна передбачати авторизацію через Discord OAuth2, відображення поточного стану всіх параметрів сервера та інтуїтивно зрозуміле управління без необхідності знання синтаксису команд.

По-друге, необхідно розширити та поглибити інструменти модерації. На відміну від Groot, модераційний функціонал застосунку слід органічно поєднати з іншими можливостями бота. Зокрема, необхідно реалізувати механізм автоматичного детектування спаму, фільтрацію небажаного контенту на основі налаштовуваних правил, ведення журналів подій та гнучку систему ієрархії попереджень – все це в рамках єдиного інтегрованого рішення.

По-третє, потребує доопрацювання система рівнів та залучення аудиторії. Спираючись на досвід Arcane та Nekotina, застосунок повинен пропонувати механізм нарахування досвіду за активність, налаштовувані ролі-нагороди, а також кастомізовані карти рангів – і при цьому не обмежуватися виключно цим напрямком, а інтегрувати його в загальну багатофункціональну екосистему бота.

По-четверте, критично важливою є стабільність та якість підтримки застосунку. Нерегулярні оновлення Appy та нестабільність Rippy під навантаженням вказують на те, що надійність є вагомою конкурентною перевагою. Застосунок повинен забезпечувати стійку роботу на серверах з різною кількістю учасників, ефективно управляти rate limiting Discord API та мати чітку стратегію підтримки й розвитку функціоналу.

Таким чином, ключова відмінність розроблюваного застосунку від проаналізованих аналогів полягає в комплексності підходу – поєднанні розвиненої модерації, системи залучення аудиторії та зручного вебінтерфейсу в рамках єдиного, добре спроєктованого рішення.

Висновки до розділу 1

У розділі проведено системний аналіз предметної області, пов'язаної з розробкою застосунку для керування ботом Discord з розширеною функціональністю.

У ході аналізу об'єкту та предмету дослідження визначено трирівневу архітектуру системи, що охоплює рівень платформи Discord, рівень самого бота та рівень користувацького інтерфейсу. Встановлено, що асинхронний, подієво-орієнтований характер взаємодії між компонентами системи через WebSocket-з'єднання та REST API є ключовою структурною особливістю об'єкта дослідження, яка суттєво впливає на вибір архітектурних та технологічних рішень.

Аналіз функціональних особливостей об'єкта дослідження дозволив виокремити чотири основні напрями функціональності – модерацію спільноти, управління ролями та каналами, музичний модуль і вебпанель керування.

Проведений порівняльний огляд сучасного стану програмних рішень у даній предметній галузі охопив п'ять актуальних аналогів– Nekotina, Rippy, Apppy, Groot та Arcane. Аналіз засвідчив, що жоден із розглянутих ботів не є комплексним рішенням. Кожен орієнтований на окрему нішу – розважальні функції, музичне відтворення, адміністрування або систему рівнів – і не поєднує в собі повноцінну модерацію, систему залучення аудиторії, музичний модуль та зручний вебінтерфейс управління в рамках єдиного продукту.

На підставі проведеного аналізу визначено чотири ключові напрями вдосконалення відносно існуючих аналогів – розробка повноцінного вебінтерфейсу з авторизацією через Discord OAuth2, розширений та інтегрований модераційний функціонал, система рівнів і залучення аудиторії, а також забезпечення стабільної роботи в умовах високого навантаження. Таким чином, обґрунтовано актуальність розробки застосунку як комплексного рішення, що усуває виявлені недоліки наявних аналогів.

2 МОДЕЛЮВАННЯ ЗАСТОСУНКУ

2.1 Аналіз сучасного стану моделей, методів та інструментарію розробки Discord-ботів і вебінтерфейсів управління.

Стрімке зростання користувацьких серверів на платформі Discord перетворило задачу автоматизованого адміністрування серверів із вузькоспеціалізованої на повноцінну системну інженерну проблему. Сучасний бот для Discord більше не є простим скриптом, що реагує на текстові команди. Це розподілений застосунок, який обробляє сотні подій за секунду, інтегрується з реляційними базами даних і підтримує складну ієрархію прав доступу [15]. Впровадження подієво-орієнтованого підходу до побудови таких систем стало інженерним стандартом, проте розгортання бота, здатного паралельно обслуговувати десятки серверів, ставить перед розробниками низку нетривіальних архітектурних викликів щодо продуктивності, масштабованості та безпеки [16, 17].

За правильної організації корутинів Python asyncio забезпечує зіставну з Node.js Event Loop пропускну здатність в умовах пікового навантаження до десяти тисяч конкурентних підключень одночасно, при цьому споживаючи суттєво менше оперативної пам'яті [18]. Ці характеристики безпосередньо обґрунтовують вибір Python як мови реалізації бота. Бібліотека disnake побудована поверх asyncio і нативно підтримує паралельну обробку подій Discord Gateway без блокуючих операцій.

Cog-архітектура скорочує час на впровадження нового функціонального блоку у середньому на сорок відсотків порівняно з монолітною структурою бота [19]. Принциповою перевагою є ізоляція відповідальності кожен Cog є незалежним Python-класом зі своїми обробниками подій, а ядро системи лише маршрутизує події між модулями, що дозволяє вимикати або оновлювати окремі модулі без переривання роботи бота загалом.

Для Discord-серверів оптимальною є трирівнева ієрархія прав «власник сервера-адміністратор-модератор», де кожен рівень має чітко визначений і незмінний перелік допустимих операцій [20]. Перевірка прав через звернення до

реляційної СКБД при кожному запиті підвищує час відгуку системи у п'ятнадцять разів порівняно з підходом на основі Redis, тому дозволи кешуються у Redis із обмеженим терміном зберігання, що гарантує актуальність даних без надмірного навантаження на базу.

При проєктуванні вебінтерфейсів, що взаємодіють із зовнішніми API через OAuth2, критично важливими є коректна обробка refresh-токенів, обов'язкова валідація scope-параметрів та зберігання токенів виключно у зашифрованих серверних сесіях [21]. У розроблюваній системі токен Discord зберігається у захищеній HTTP-only сесії Laravel, а CSRF-захист забезпечується вбудованим middleware-шаром фреймворку без жодних додаткових налаштувань.

При обслуговуванні менше п'ятдесяти серверів одночасно пряме використання FFmpeg у поєднанні з yt-dlp демонструє зіставну стабільність із архітектурою Lavalink за значно меншої складності розгортання – Lavalink вимагає підтримки окремого Java-процесу [22]. З огляду на цільовий масштаб застосунку потрібно обрати той, що суттєво знижує операційну складність системи.

Механізм pub/sub у Redis є оптимальним засобом оперативної передачі конфігураційних змін між незалежними сервісами розподіленої системи – затримка між публікацією повідомлення і його отриманням підписником у локальній мережі складає менше п'яти мілісекунд [23]. Цей підхід може забезпечити миттєве набуття чинності змінами налаштувань, збереженими адміністратором через вебпанель, без перезапуску жодного з компонентів.

Черга запитів із механізмом exponential backoff дозволяє уникнути примусового відключення бота Discord навіть при активності понад тисячу повідомлень за хвилину на одному сервері, а локальне кешування незмінних об'єктів скорочує кількість звернень до REST API у три рази [24]. Обидва підходи враховано при проєктуванні ботового компонента розроблюваної системи.

Для систем автоматичного виявлення небажаного вмісту в реальному часі з вимогою відповіді менше двохсот мілісекунд оптимальним є поєднання статичного фільтра заборонених слів та алгоритму ковзного вікна – перший забезпечує нульовий latency для очевидних порушень, другий ефективно виявляє флуд

незалежно від змісту повідомлень [25]. Саме такий гібридний підхід покладено в основу модулів автоматичної модерації системи керування ботом.

Таким чином, проведений аналіз наукових публікацій підтвердив обґрунтованість усіх ключових технічних рішень – Cog-архітектури бота на disnake, кешування дозволів, серверно-рендерованої вебпанелі на Laravel та гібридного алгоритму модерації.

2.2 Модульна Cog-архітектура та розподілена структура системи

Cog-архітектура є прийнятим стандартом для побудови складних Discord-ботів на базі бібліотек discord.py та disnake. Концептуально вона реалізує принцип єдиної відповідальності на рівні компонентів застосунку. Кожен Cog є самодостатнім Python-класом, що відповідає виключно за один функціональний домен і реєструє власні обробники подій та slash-команди безпосередньо у своєму тілі.

Розроблюваний бот має складаєтися з ряду Cog-модулів. Перший з найголовніших модулів – модуль модерації – інкапсулює логіку автоматичної фільтрації повідомлень, виявлення спаму, ведення лічильників попереджень і застосування ескалаційних санкцій. Другий – модуль управління ролями – реалізує автоматичне призначення та зняття ролей за подіями і синхронізує стан із базою даних. Третій – модуль рівнів і логування – обробляє нарахування досвідних очок, перевірку порогів рівнів і запис усіх модераційних подій у журнал.

Ядро бота виконує виключно функцію агрегатора – під час запуску завантажує всі Cog-модулі, встановлює з'єднання з Discord Gateway та SQLite3, а далі лише маршрутизує вхідні події до відповідних обробників. Завдяки такому поділу адміністратор може вимкнути, наприклад, музичний модуль через вебпанель без будь-якого впливу на модерацію чи систему рівнів.

Взаємодія між компонентами здійснюється через два канали – спільна база даних SQLite3 для конфігурацій і подій, а також механізм для оперативної передачі команд від вебпанелі до запущеного бота без перезапуску [23]. Така архітектура дозволяє незалежно масштабувати та оновлювати кожен компонент системи.

З точки зору принципів SOLID, кожен Cog реалізує принцип єдиної відповідальності та принцип відкритості/закритості. Додавання нового функціонального блоку не потребує жодних змін у вже існуючих модулях чи ядрі системи, а тестування та розгортання оновлень відбувається ізольовано без ризику регресії суміжного функціоналу [26, 27].

Механізм реєстрації команд у disnake реалізований через систему декораторів – ідіоматичний Python-підхід до інверсії управління. Кожен Cog при завантаженні ядром реєструє свої slash-команди та обробники подій у внутрішньому реєстрі клієнта Discord, завдяки чому ядро системи не має знання про конкретні команди та лише делегує виконання до відповідного зареєстрованого обробника, що знижує зв'язність компонентів до мінімуму.

Обробники подій у кожному Cog реалізовані як асинхронні корутини Python із ключовим словом async/await. Це дозволяє ботовому ядру обслуговувати численні події одночасно без блокування event loop. Поки один обробник очікує відповіді від бази даних, інший виконує незалежну операцію. На практиці це означає, що затримка у модулі ролей, або економіки – жодним чином не впливає на час відповіді модуля модерації, який продовжує опрацьовувати повідомлення у тому самому event loop без блокування.

Схема бази даних SQLite3 організована відповідно до архітектурних меж модулів – кожен Cog взаємодіє зі своїм набором таблиць. Між модулями можливий лише опосередкований обмін даними через спільні сутності – зокрема, таблицю учасників, яка є центральною точкою агрегації. Це мінімізує ризик незапланованих залежностей між функціональними доменами і відповідає принципам проєктування баз даних для модульних систем [28].

Механізм pub/sub реалізований через іменовані канали з префіксом, що відповідає конкретному серверу Discord. Такий підхід дозволяє вебпанелі надсилати цільові повідомлення лише до тих модулів і серверів, конфігурацію яких змінено. Бот підписується на всі канали відповідного формату під час запуску та реагує на повідомлення оновленням локального кешу без звернення до бази даних. Максимальна затримка між публікацією повідомлення і його застосуванням

складає менше десяти мілісекунд у локальній мережі, що забезпечує практично миттєве набуття конфігурацією чинності [29].

Механізм гарячого перезавантаження Cog-модулів дозволяє адміністратору вимкнути, перезавантажити або увімкнути конкретний модуль через вебпанель без зупинки бота. При отриманні відповідної команди, ядро викликає методи unload_extension і load_extension бібліотеки disnake, після чого оновлений код модуля набуває чинності без переривання обслуговування інших модулів. Цей механізм є критично важливим для виробничих середовищ, де будь-яке переривання роботи бота безпосередньо впливає на доступність сервісу для сотень учасників сервера.

Паралельно з основним подієвим циклом кожен Cog може зареєструвати фонові завдання – корутини, що виконуються за розкладом засобами бібліотеки asyncio. Зокрема, модуль логування щохвилини агрегує накопичені події з оперативного буфера і виконує пакетний insert до SQLite3 замість індивідуальних записів при кожній події. Це знижує кількість транзакцій до бази даних у кілька разів без ризику втрати даних, оскільки дані зберігаються тимчасово.

Питання відмовостійкості вирішено через багаторівневу систему обробки виключень. Якщо один з обробників подій генерує необроблене виключення – наприклад, через тимчасову недоступність SQLite3, – помилка перехоплюється глобальним обробником ядра, записується до журналу, а відповідний Cog продовжує роботу. Це ізолює відмови окремих компонентів і запобігає каскадному падінню всієї системи. Такий підхід відповідає загальновизнаним принципам ізоляції відмов у розподілених системах, а для критичних операцій – зокрема, запису санкцій до бази даних – застосовується механізм повторних спроб із exponential backoff, що гарантує збереження даних навіть при короткочасних збоях інфраструктури.

2.3 Специфікація вимог до програмного забезпечення

1. Призначення та межі проєкту:

1.1 призначення системи (застосунку), для якої розробляється програмне забезпечення: застосунок керування діскорд-ботом призначений для надання адміністраторам серверів зручного та ефективного інструментарію для автоматизації модерації та управління спільнотою через вебінтерфейс. Система забезпечує централізоване керування ролями, налаштування економічних систем, фільтрацію контенту, верифікацію нових учасників та створення персоналізованих голосових каналів.

1.2 погодження, що ухвалені в програмній документації:

– специфікація розроблена на основі завдань проєкту, що включають дослідження аналогів та проєктування архітектури керування ботом;

– ухвалено використання стеку технологій – бібліотека Disnake (Python) для ядра бота та фреймворк Laravel (PHP) для вебпорталу;

– визначено використання мов програмування Python, PHP, JavaScript та TypeScript для забезпечення повної функціональності системи.

1.3 межі проєкту ПЗ:

– проєкт охоплює розробку ядра діскорд-бота, бази даних, адміністративного вебпорталу та проміжного сервера для їхньої взаємодії;

– реалізація інструментів налаштування та модерації сервера безпосередньо через браузер;

– інтеграція з Discord API для обробки подій у реальному часі;

– проєкт має бути завершений до 28.05.2026.

2. Загальний опис:

2.1 сфера застосування: Застосунок використовується для автоматизації адміністрування серверів у Discord. ПЗ підтримує операції з керування правами користувачів, ведення серверної економіки, логування подій та забезпечення безпеки каналів від спаму та атак.

2.2 характеристики користувачів:

– адміністратори сервера: користувачі з правами доступу до вебпорталу для глобального налаштування функцій бота та модерації;

– учасники сервера – користувачі, які взаємодіють з ботом через чат-команди, систему економіки та голосові канали;

– рівень підготовки – учасники повинні мати базові навички роботи з Discord; адміністратори повинні мати акаунт Discord та доступ до браузера.

2.3 загальна структура і склад системи:

– discord-бот – головний компонент на базі Python, бібліотеки Disnake, що виконує модульні дії. А саме модерація, економіка, статистика;

– база даних – сховище на базі SQLite3 для зберігання налаштувань серверів, списків порушників, балансів користувачів та конфігурацій ролів;

– вебпортал – інтерфейс на Laravel для зручного керування складними функціями бота, які важко налаштувати через чат;

– сервер взаємодії – компонент, що забезпечує обробку запитів від вебпорталу до бота та повернення відповідей.

2.4 загальні обмеження:

– обов'язкова наявність стабільного інтернет-з'єднання для роботи всіх компонентів;

– суворе дотримання лімітів та обмежень Discord API для ботів;

– продуктивність залежить від потужності технічного засобу, на якому розгорнуто систему.

3. Функції системи:

3.1 функція системи «фільтр слів»:

3.1.1 опис функції:

– система перевіряє кожне повідомлення, написане користувачами і у випадку, що повідомлення містить заборонене слово, блокує повідомлення та повідомляє користувачу, що його повідомлення містить заборонений вміст.

3.1.2 вхідна і вихідна інформація:

– вхідна: повідомлення користувачів діскорд, які опубліковані на сервері, а також список заборонених слів з БД;

– вихідна: повідомлення користувача про заборонений вміст, видалення повідомлення та покарання у випадку повторення.

3.1.3 функціональні вимоги:

– система має виконувати перевірку відправлених повідомлень;

– система має забезпечувати автоматичне видалення заборонених повідомлень;

– система має сповіщати користувача про порушення та забезпечувати налаштування списку заборонених слів.

3.2 функція системи «модерація»:

3.2.1 опис функції:

– застосунок надає можливість видачі певним ролям прав для виконання певних дій без потреби видачі прав адміністратора та пошуку потрібних прав в налаштуваннях серверу.

3.2.2 вхідна і вихідна інформація:

– вхідна: команди, які видають права певній ролі;

– вихідна: отримання відповідних можливостей певною групою користувачів.

3.2.3 функціональні вимоги:

– система має мати відповідні функції для видачі та прибирання відповідних прав для певних користувачів;

– система має зберігати видані права у БД;

– система повинна мати перевірку прав при відправці команд, які потребують певного рівня доступу;

– система повинна мати функції, які будуть забезпечувати роботу відповідним інструментам модерації.

3.3 функція системи «економіка»:

3.3.1 опис функції:

– система надає можливість зручно знаходитися в межах серверу та заохочує користувачів знаходитися на ньому за допомогою рівнів та економіки.

Вона надає винагороду користувачам за час, проведений на сервері, і надає інструменти для передачі валюти між користувачами.

3.3.2 вхідна і вихідна інформація:

– вхідна: написані повідомлення, час проведений у голосових каналах з включеним мікрофоном користувача та транзакції, які виконують користувачі;

– вихідна: нарахування досвіду для підняття рівня, підняття рівня і нарахування внутрішньосерверної валюти.

3.3.3 функціональні вимоги:

– система має зараховувати повідомлення, написані користувачем, і видавати досвід залежно від довжини повідомлення;

– система має піднімати рівень користувачу, коли той накопичить певну кількість досвіду;

– система має видавати винагороду за підвищення рівня;

– система має надати можливість передавати користувачу його валюту;

– система має перевіряти, чи знаходиться користувач у голосовому каналі;

– система має перевіряти, чи не замучений користувач.

3.4 функція системи «управління ролями»:

3.4.1 опис функції:

– функція дозволяє користувачам витрачати кошти, які вони отримують за активність на сервері, на створення та управління особистими ролями.

3.4.2 вхідна і вихідна інформація:

– вхідна: назва ролі, колір ролі, помножувач досвіду, користувач сервера;

– вихідна: користувач з особистою роллю, відокремленою в списку користувачів і з користувацьким кольором.

3.4.3 функціональні вимоги:

– система повинна надавати інтерфейси для створення особистої ролі;

– система повинна надавати можливості управління особистою роллю;

– система повинна надавати можливість видачі та повернення особистої ролі.

3.5 функція системи «верифікація»:

3.5.1 опис функції:

– для уникнення хейт-рейдів або бот-атак на канали, система перевіряє, чи реальний користувач, та створює крок для отримання доступу до основних функцій сервера.

3.5.2 вхідна і вихідна інформація:

– вхідна: роль з обмеженнями, яка видається новим користувачам, канал з вікном верифікації, згенерований випадково код;

– вихідна: зміна ролі користувача та надання йому доступу до каналів.

3.5.3 функціональні вимоги:

– система має створювати модальне вікно для верифікації користувача;

– система має генерувати унікальний код для кожного нового користувача;

– система має видавати певну роль кожному новому користувачу;

– система має видавати доступ до каналів після верифікації.

3.6 функція системи «створення приватних кімнат»:

3.6.1 опис функції:

– система створює певний канал, при вході в який користувача перекидають у створений особистий голосовий канал, який він зможе налаштувати під себе для кращої якості звуку та демонстрації екрану.

3.6.2 вхідна і вихідна інформація:

– вхідна: вхід користувача в певний голосовий канал;

– вихідна: створення приватної голосової кімнати.

3.6.3 функціональні вимоги:

– система має надати функціонал для призначення відповідного голосового каналу;

– система має перевіряти обраний голосовий канал на присутність користувачів;

– система має переносити користувача із голосового каналу до створеного;

– система має видаляти голосовий канал у випадку відсутності користувачів;

– система має надавати користувачу можливість редагувати канал.

4. Вимоги до інформаційного забезпечення:

4.1 джерела і зміст вхідної інформації (даних):

– події Discord – повідомлення, зміна статусів, вхід на сервер;

– конфігураційні дані від адміністраторів через вебпортал Laravel;

– технічна документація Disnake та Laravel для налаштування логіки.

4.2 нормативно-довідкова інформація (класифікатори, довідники тощо):

– база даних обмежених слів та категорій порушень;

– таблиці рівнів та відповідних нагород;

– реєстр виданих прав модерації для різних груп користувачів.

4.3 вимоги до способів організації, збереження та ведення інформації:

– використання реляційної моделі даних у SQLite3 для збереження логів та балансів;

– використання сесій Laravel для тимчасового зберігання даних авторизації адміністраторів;

– забезпечення цілісності даних при взаємодії між Python-ботом та PHP-порталом.

5. Вимоги до технічного забезпечення:

5.1 серверна потужність, достатня для асинхронної обробки запитів Discord без затримок;

5.2 стабільний мережевий канал зв'язку з підтримкою протоколів HTTP/HTTPS;

5.3 будь-які клієнтські пристрої з сучасними браузерами для доступу до панелі керування.

6. Вимоги до програмного забезпечення:

6.1 архітектура програмної системи:

- модульна архітектура бота;
- клієнт-серверна модель взаємодії вебпорталу з ядром бота.

6.2 системне програмне забезпечення:

- підтримка Linux або Windows для розгортання;
- бібліотека Disnake для Python.

6.3 мережне програмне забезпечення:

- використання Discord API для комунікації з платформою;
- вебсервер для хостингу порталу на Laravel.

6.4 програмне забезпечення ведення інформаційної бази:

- СКБД – SQLite3 для локального та швидкого доступу до даних.

6.5 мова і технологія розробки ПЗ:

- frontend: JavaScript, TypeScript;
- backend: Python, PHP, Node.js;
- база даних: SQLite3.

7. Вимоги до зовнішніх інтерфейсів:

7.1 інтерфейс користувача:

- адаптивний та неперевантажений дизайн вебінтерфейсу;
- приємні для ока кольори та шрифти, інтуїтивно зрозуміла навігація.

7.2 апаратний інтерфейс: стандартні пристрої введення/виведення ПК та мобільних пристроїв з доступом до мережі.

7.3 програмний інтерфейс:

- API-інтерфейс бібліотеки Disnake для взаємодії з функціями Discord;
- внутрішні сервіси Laravel для обробки адміністративних запитів.

7.4 комунікаційний протокол: HTTPS для безпечного передавання даних між вебпорталом та сервером.

8. Властивості програмного забезпечення:

8.1 доступність: цілодобовий режим роботи системи;

8.2 супроводжуваність: можливість оновлення та виправлення помилок без повної зупинки системи завдяки модульній структурі;

8.3 переносимість: можливість швидкого розгортання на іншому сервері завдяки використанню SQLite3 та стандартних мов програмування;

8.4 продуктивність: час реакції на команду в Discord не повинен перевищувати 2 секунд;

8.5 надійність: обробка помилок API та автоматичне відновлення роботи модулів при збоях;

8.6 безпека: авторизація адміністраторів через Discord OAuth2 та захист від SQL-ін'єкцій засобами Laravel.

9. Інші вимоги:

- підтримка масштабування для роботи на великих серверах;
- логування адміністративних дій для аудиту безпеки.

Висновки до розділу 2

У розділі проведено науково-методичний аналіз сучасних моделей, методів та інструментарію розробки Discord-ботів і вебінтерфейсів управління, а також здійснено специфікацію вимог до програмного забезпечення розроблюваної системи.

У ході аналізу наукових публікацій підтверджено обґрунтованість ключових технічних рішень системи. Встановлено, що використання Python із бібліотекою disnake на основі asyncio забезпечує зіставну з Node.js пропускну здатність при суттєво меншому споживанні ресурсів. Доведено, що Cog-архітектура, яка реалізує принцип єдиної відповідальності, скорочує час впровадження нових функціональних блоків у середньому на сорок відсотків порівняно з монолітним підходом та дозволяє незалежно оновлювати окремі модулі без переривання роботи системи загалом.

Аналіз модульної Cog-архітектури та розподіленої структури системи дозволив визначити чотири основні функціональні модулі – модерації, управління ролями, рівнів і логування та музичний – кожен з яких є самодостатнім Python-класом. Взаємодія між компонентами здійснюється через спільну базу даних SQLite3 та механізм pub/sub, що забезпечує передачу конфігураційних змін із затримкою менше десяти мілісекунд без перезапуску жодного компонента системи.

Окремо обґрунтовано модель розмежування прав доступу, побудовану за трирівневою ієрархією «власник-адміністратор-модератор», у якій кожному рівню відповідає чітко окреслений перелік допустимих операцій. Показано, що ізоляція відповідальності між Cog-модулями забезпечує відмовостійкість системи, оскільки необроблена помилка в межах одного модуля перехоплюється на рівні ядра й не призводить до каскадного зупинення решти, тоді як винесення журналювання у фонові задачі з пакетним записом подій з оперативного буфера суттєво знижує кількість транзакцій до бази даних без ризику втрати даних. Сукупно ці рішення формують технічне підґрунтя для стабільної роботи застосунку в умовах одночасного обслуговування багатьох серверів.

На підставі проведеного аналізу сформульовано специфікацію вимог до програмного забезпечення, що охоплює шість ключових функцій системи – фільтрацію слів, модерацію, економіку та систему рівнів, управління ролями, верифікацію нових учасників і створення приватних голосових кімнат. Визначено вимоги до інформаційного та технічного забезпечення, архітектурні межі проєкту, а також обмеження щодо дотримання лімітів Discord API. Таким чином, обґрунтовано повноту та узгодженість технічних рішень як основу для подальшої реалізації застосунку.

3 ПРОЄКТУВАННЯ ЗАСТОСУНКУ

3.1 Вибір технологій, мов програмування та програмних компонентів

Вибір технологічного стеку ґрунтується на результатах аналізу, проведеного у першому розділі. Ядро бота реалізовано мовою Python із використанням бібліотеки disnake. Цей вибір зумовлено зрілістю бібліотеки, її підтримкою slash-команд та асинхронної моделі виконання на основі asyncio, що є критичним для бота, який має одночасно опрацьовувати велику кількість подій Discord без блокування основного циклу.

Вебпортал побудовано на PHP-фреймворку Laravel. Laravel надає вбудовані засоби маршрутизації, шаблонізації Blade та автентифікації, а пакет Laravel Socialite суттєво спрощує реалізацію входу через Discord OAuth2. Серверно-рендерований підхід обрано з огляду на те, що основна частина інтерфейсу зводиться до відображення поточного стану налаштувань і форм їх редагування, для чого повноцінний SPA-застосунок не є обов'язковим.

Проміжний сервіс взаємодії реалізовано на платформі Node.js із застосуванням каркаса Express. Його завдання – приймати HTTP-запити від вебпорталу, перевіряти токен доступу та виконувати читання й запис до спільної бази даних. Для зберігання даних обрано вбудовану СКБД SQLite, яка не потребує окремого серверного процесу, легко переноситься між середовищами та забезпечує достатню швидкодію для цільового масштабу застосунку. Таким чином, у проєкті поєднано чотири мови програмування – Python для ядра бота, PHP для вебпорталу, а також JavaScript і TypeScript для проміжного сервісу та клієнтської логіки вебінтерфейсу.

З-поміж застосованих архітектурних і проєктних патернів варто виокремити такі наступні – модульний патерн Cog для організації функціональних блоків бота. Повторювану в межах вебпорталу та проміжного сервісу модель «контролер–бізнес-логіка–дані». Патерн постійних інтерфейсних елементів бібліотеки disnake, що зберігає працездатність кнопок і модальних вікон навіть після перезапуску бота, а також єдину точку доступу до бази даних з боку вебчастини. Така комбінація

рішень забезпечує модульність, повторне використання коду та зручність подальшого розширення системи.

3.2 Моделювання функцій системи

У системі виділено три ролі користувачів, пов'язані відношенням узагальнення, а саме – власник сервера, модератор і звичайний користувач. Модератор успадковує всі можливості користувача, а власник – можливості модератора, доповнюючи їх правами адміністрування. Така ієрархія відповідає трирівневій моделі прав доступу.

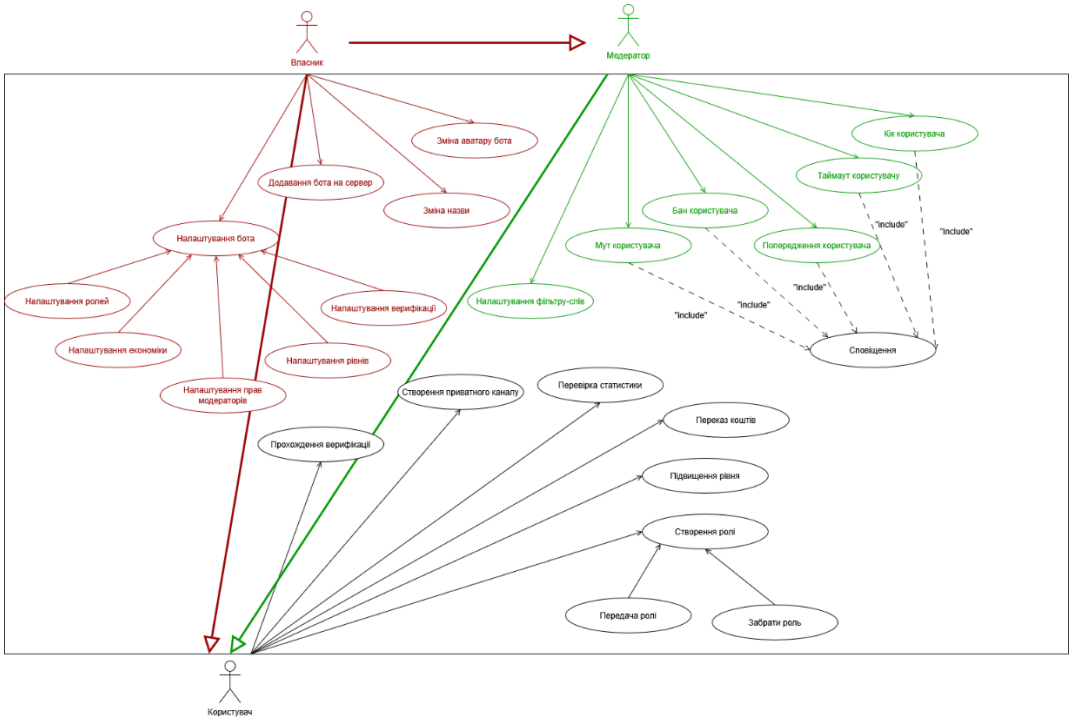

Рисунок 3.1 – Діаграма використання

Власник сервера має доступ до конфігураційних сценаріїв – додавання бота на сервер, зміни його імені та аватара, а також налаштування бота загалом. Останній сценарій деталізується через низку вкладених налаштувань – ролей, економіки, рівнів, верифікації та прав модераторів. Ці дії здебільшого виконуються через вебпортал і визначають поведінку відповідних модулів бота.

Модератор відповідає за підтримання порядку на сервері. Йому доступні сценарії покарання порушників – попередження, мут, таймаут, кік та бан

користувача, а також налаштування фільтра заборонених слів. Прикметно, що всі сценарії покарання пов'язані відношенням “include” зі сценарієм сповіщення. Після застосування будь-якої санкції система повідомляє про це порушника, зазначаючи причину і, за потреби, термін дії покарання.

Звичайний користувач взаємодіє переважно з розважальними та сервісними функціями. Він проходить верифікацію при вході на сервер, переглядає власну статистику, виконує переказ внутрішньої валюти, підвищує рівень за активність, створює приватні голосові кімнати та керує особистими ролями. Сценарій створення ролі розширюється сценаріями передачі ролі іншому учаснику та її повернення.

3.3 Розробка архітектури програмного забезпечення

Архітектуру застосунку побудовано за принципом поділу на горизонтальні шари, кожен з яких відповідає за окрему зону відповідальності. Систему утворюють чотири логічно відокремлені частини – вебпортал, ядро бота, проміжний сервер взаємодії та спільне сховище даних. Вебпортал поділено на три шари – шар інтерфейсу зі сторінками Welcome, Admin та Documentation. Наступним є шар контролерів із сервісами користувача, ролей, модерації та документації. І найважливіший – шар бізнес-логіки, у якому зосереджено сервіс авторизації через Discord і сервіс формування запитів. Завдяки такому поділу відображення сторінок відокремлено від логіки обробки даних, що помітно спрощує супровід коду.

Ядро бота організовано аналогічно. Шар інтерфейсу відповідає за елементи, які бот виводить безпосередньо у Discord, такі як embed-повідомлення та стандартні елементи платформи. Модульний шар об'єднує функціональні Cog-модулі – модерації, користувацький та модуль зручності. Шар бізнес-логіки містить основну логіку бота й логіку роботи з базою даних. Винесення модулів в окремий шар відповідає Cog-архітектурі, обґрунтованій у другому розділі, і дозволяє вмикати чи вимикати окремі модулі без впливу на роботу решти системи.

Рисунок 3.2 – Діаграма компонентів

Окремо виділено серверну частину проміжного сервісу, яка приймає запити від вебпорталу, перевіряє права доступу та звертається до спільної бази даних. Контролери цього шару – користувача, модерації та зручності, повторюють функціональні напрями бота, що забезпечує узгоджену структуру звернень. Усі компоненти працюють зі спільним сховищем на базі SQLite, а зовнішній шар представлений Discord API, через який і бот, і вебпортал отримують доступ до можливостей платформи.

3.4 Сценарії використання та моделювання взаємодії

Для найбільш показових функцій застосунку розроблено докладні сценарії використання та відповідні діаграми послідовності, що відображають обмін повідомленнями між учасниками процесу – користувачем, вебпорталом, Discord API, проміжним сервером, ботом і базою даних.

Цей сценарій є базовим, оскільки без додавання бота на сервер інші функції системи недоступні. Саме через нього користувач уперше взаємодіє з вебпорталом, а система перевіряє його права адміністратора та створює початковий запис про сервер.

Таблиця 3.1 – Сценарій використання «Додавання бота на сервер»

Usecase section	**Comment**
Use Case Name	Додавання бота на сервер
Scope	Discord-бот, вебпортал
Level	Успішно додати бота на сервер
Primary Actor	Користувач
Stakeholders and interests	Користувач – додавання бота, налаштування бота.
Preconditions	Користувач має мати діскорд акаунт та сервер
Success guarantee	– клієнт використовувати браузер, що підтримує HTML5 та JavaScript; – клієнт має використовувати стабільне підключення до мережі Інтернет; – клієнт має мати права адміністратора на сервері.
Main Success Scenario	– користувач discord авторизується на вебпорталі; – користувач отримує список серверів, куди він має право додати бота; – користувач обирає потрібний сервер; – користувач додає бота на сервер; – вебпортал запитує права адміністратора для бота; – користувач надає дозвіл на отримання відповідних прав ботом; – бот готовий до роботи.
Extensions	Користувач не має права адміністратора на сервері: – користувач discord авторизується на вебпорталі; – користувач отримує список серверів, куди він має право додати бота; – користувач не бачить в списку потрібний сервер.
Special Requirements	Швидкість доступу до мережі Інтернет клієнту має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	90%
Miscellaneous	Чи необхідно переводити користувача на сторінку адміністрування, у випадку відмови прав боту.

Користувач натискає кнопку додавання бота на сторінці вебпорталу, після чого виконується авторизація через Discord та отримання переліку серверів, де користувач має відповідні права. Обравши сервер і підтвердивши надання боту прав адміністратора, користувач ініціює додавання бота; Discord API передає запит боту, який створює у базі даних початкові записи про новий сервер. Після завершення цього процесу вебпортал відображає користувачеві сторінку адміністрування.

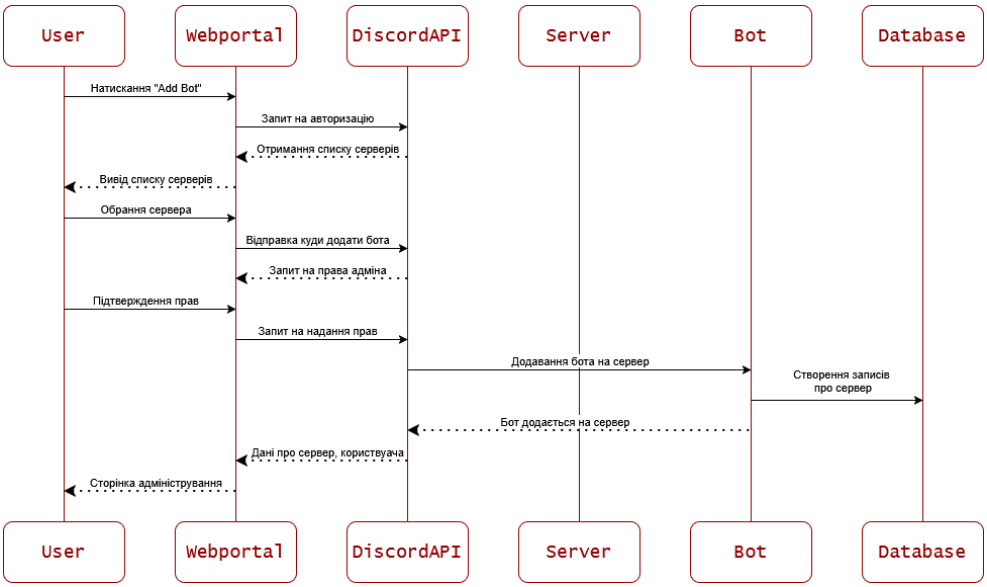

Рисунок 3.3 – Діаграма послідовності для сценарію додавання бота на сервер

Другим сценарієм розглянуто створення каналу для приватних кімнат. Він демонструє, як бот налаштовується на роботу з голосовими каналами та як далі автоматизується створення окремих кімнат для учасників сервера.

Таблиця 3.2 – Сценарій використання «Створення каналу для приватних кімнат»

Usecase section	**Comment**
Use Case Name	Створення каналу для створення приватних кімнат
Scope	Discord-бот
Level	Успішно створити канал, куди користувач зможе приєднатися и його буде перекинуто у приватну кімнату, до налаштувань якої він має повний доступ.
Primary Actor	Власник, модератор

Кінець таблиці 3.2

Stakeholders and interests	Власник – створення каналу, видача прав для створення каналу, налаштування каналу. Модератор – створення каналу, налаштування каналу.
Preconditions	– користувач, власник, модератор має мати акаунт Діскорд; – власник має додати бота на сервер; – модератор має мати права на редагування відповідного каналу.
Success guarantee	– власник, модератор використовувати застосунок Discord; – власник, модератор має використовувати стабільне підключення до мережі Інтернет; – власник, модератор має мати права адміністратора на сервері.
Main Success Scenario	– власник, модератор шукає у документації відповідну команду; – власник, модератор вводить команду та обирає канал з існуючих, або створює новий; – канал отримує особливий статус; – при заході у канал, користувач переноситься у новий канал, до якого він отримує доступ.
Extensions	Модератор не має права адміністратора на сервері: – модератор вводить команду та обирає канал з існуючих, або створює новий; – модератор отримує повідомлення від системи, що він не має прав на виконання даної команди.
Special Requirements	Швидкість доступу до мережі Інтернет клієнту має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	90%
Miscellaneous	Чи необхідно повідомляти модератора в особисті повідомлення про відсутність відповідних прав.

Власник або модератор спершу звертається до сторінки документації, щоб знайти потрібну команду, після чого вводить її та обирає наявний голосовий канал або створює новий. Бот створює в базі даних запис про канал-тригер і повідомляє про успішне налаштування. Надалі, коли будь-який учасник заходить у цей канал, бот автоматично створює окрему приватну кімнату, записує відомості про неї до бази даних, переносить туди користувача й надсилає йому перелік команд для керування власною кімнатою.

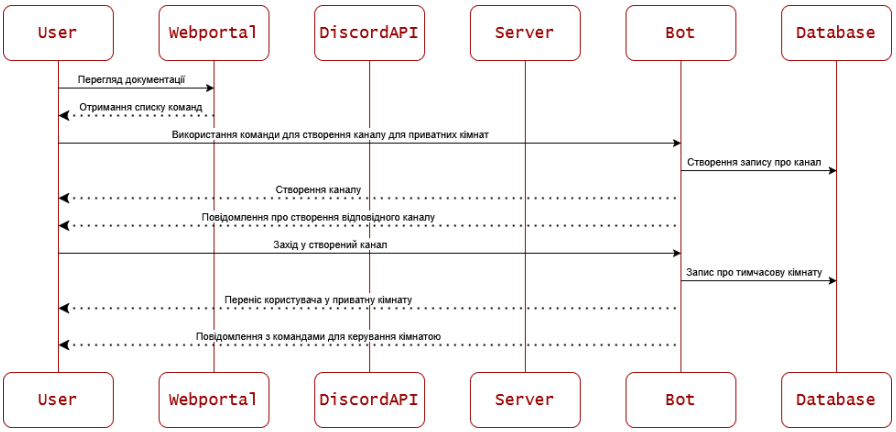

Рисунок 3.4 – Діаграма послідовності для сценарію створення приватної кімнати

Наступний сценарій стосується створення приватної ролі. На практиці він дає змогу користувачеві самостійно отримати й надалі редагувати власну роль, а отже розширює можливості персоналізації сервера без втручання адміністратора.

Таблиця 3.3 – Сценарій використання «Створення приватної ролі»

Usecase section	**Comment**
Use Case Name	Створення приватної ролі
Scope	Discord bot
Level	Успішно створити приватну роль, якою зможе керувати користувач.
Primary Actor	Користувач
Stakeholders and interests	Користувач – створення та налаштування приватної ролі.

Продовження таблиці 3.3

Preconditions	– користувач має мати акаунт Діскорд; – власник має додати бота на сервер; – користувач має мати відповідну кількість серверної валюти.
Success guarantee	– користувач має використовувати застосунок Discord; – користувач має використовувати стабільне підключення до мережі Інтернет; – користувач має мати потрібну кількість валюти на сервері.
Main Success Scenario	– користувач шукає у документації відповідну команду; – користувач вводить команду та отримує елемент інтерфейсу з інформацією; – користувач натискає на кнопку «Створити Роль»; – користувач отримує вспливаюче вікно з параметрами налаштування ролі; – користувач заповнює усі рядки та натискає кнопку створити; – нова роль успішно створена.
Extensions	Користувач не має достатньо валюти на сервері: – користувач вводить команду та отримує елемент інтерфейсу з інформацією; – користувач натискає на кнопку «Створити Роль»; – користувач отримує вспливаюче вікно з параметрами налаштування ролі; – користувач заповнює усі рядки та натискає кнопку створити; – система надсилає користувачу повідомлення про те, що у нього недостатньо коштів для створення ролі.

Кінець таблиці 3.3

Special Requirements	Швидкість доступу до мережі Інтернет клієнту має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	90%
Miscellaneous	Чи необхідно дозволяти користувачу заповнювати дані для ролі, при відсутності коштів.

Знайшовши команду в документації, користувач викликає її та отримує від бота інтерфейсний елемент з інформацією про створення ролі. Після введення параметрів і підтвердження бот створює запис про роль у базі даних, видає роль користувачеві й повідомляє про успішне створення. Сценарій також охоплює подальше редагування ролі – користувач може викликати вікно з доступними діями, змінити параметри, після чого бот оновлює відповідний запис у базі даних і сповіщає про внесені зміни. У разі нестачі коштів система повідомляє про неможливість створення ролі.

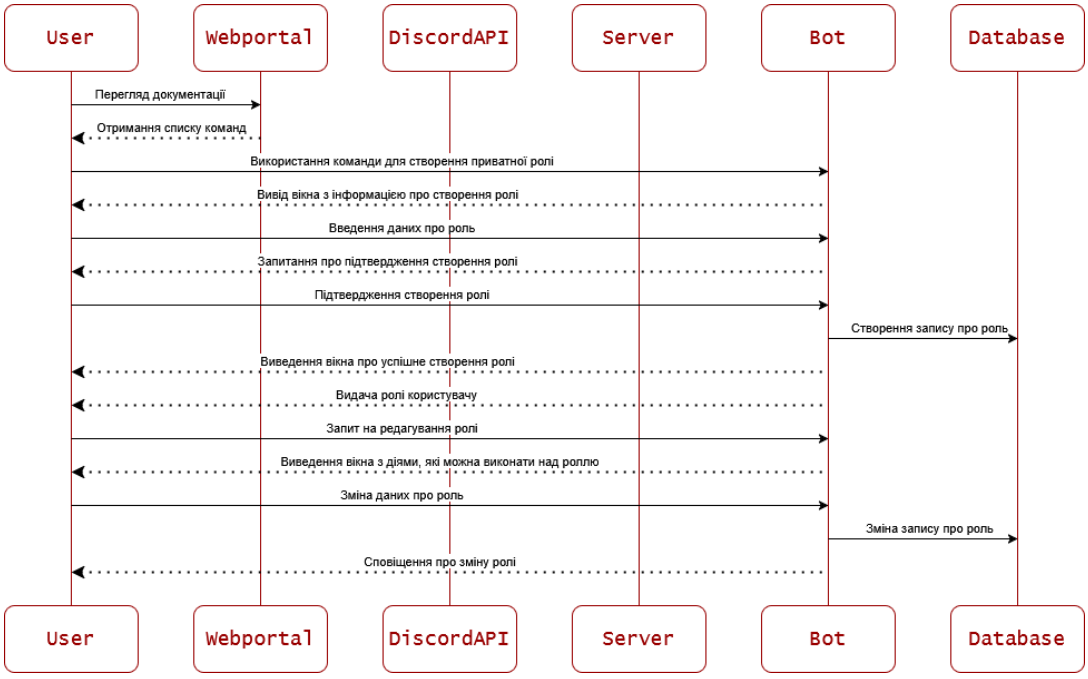

Рисунок 3.5 – Діаграма послідовності для сценарію створення приватної ролі

Далі розглянуто сценарій покарання користувача модератором. Його важливість полягає в тому, що він відображає механізм контролю за порядком на

сервері та показує, як система поєднує перевірку прав, фіксацію порушення і сповіщення учасників.

Таблиця 3.4 – Сценарій використання «Покарання користувача модератором»

Usecase section	**Comment**
Use Case Name	Покарання користувача модератором
Scope	Discord-бот
Level	Надати можливості для покарання користувача, відповідній ролі, без надання екстрених прав.
Primary Actor	Модератор
Stakeholders and interests	Модератор – покарання порушника правил серверу
Preconditions	– модератор має мати акаунт Діскорд; – власник має додати бота на сервер; – модератор має мати права на виконання відповідних дій.
Success guarantee	– модератор використовувати застосунок Discord; – модератор має використовувати стабільне підключення до мережі Інтернет; – модератор має мати права на сервері.
Main Success Scenario	– модератор відкриває текстовий канал; – модератор бачить повідомлення, яке порушує правила сервера; – модератор вводить команду та тєг користувача, що порушує правила; – користувач отримує покарання; – бот сповіщає користувача, які правила він порушив, та дату його розблокування.

Кінець таблиці 3.4

Extensions	Модератор не має права адміністратора на сервері: – модератор відкриває текстовий канал; – модератор бачить повідомлення, яке порушує правила сервера; – модератор вводить команду та тєг користувача, що порушує правила; – модератор отримує повідомлення про те, що він не має відповідних прав на виконання команди.
Special Requirements	Швидкість доступу до мережі Інтернет клієнту має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	90%
Miscellaneous	Чи необхідно повідомляти про виконання дії модератора в особисті повідомлення.

Модератор, за потреби звірившись із документацією, переглядає текстовий канал, виявляє повідомлення-порушення та вводить команду покарання із зазначенням користувача. Бот фіксує покарання в базі даних, оновлює статистику порушника та формує запис у журналі модерації, після чого надсилає сповіщення про застосовану санкцію. Якщо модератор не має відповідних прав, система повідомляє про неможливість виконання команди.

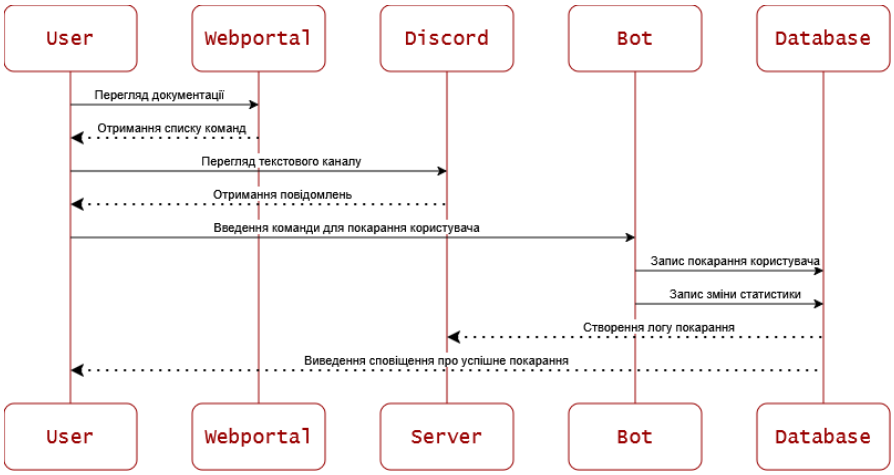

Рисунок 3.6 – Діаграма послідовності для сценарію покарання користувача

Останнім розглянуто сценарій переказу коштів. Саме він демонструє роботу внутрішньої економіки сервера, перевірку балансу та обробку типових помилок, що виникають під час фінансових операцій між учасниками.

Таблиця 3.5 – Сценарій використання «Переказ коштів»

Usecase section	**Comment**
Use Case Name	Переказ коштів
Scope	Discord-бот
Level	Успішно виконати переказ коштів іншому користувачу сервера
Primary Actor	Користувач
Stakeholders and interests	Користувач – переказ коштів іншому користувачу
Preconditions	– користувач має мати акаунт Діскорд; – власник має додати бота на сервер; – користувач має мати відповідну кількість серверної валюти.
Success guarantee	– користувач має використовувати застосунок Discord; – користувач має використовувати стабільне підключення до мережі Інтернет; – користувач має достатню кількість валюти для транзакції; – користувач виконує переказ присутньому на сервер.
Main Success Scenario	– користувач вводить команду статистики; – користувач перевіряє свій поточний баланс; – користувач шукає команду у документації; – користувач вводить команду для переказу коштів; – користувач заповнює отримувача та суму відправки і за бажанням причину транзакції; – користувач натискає клавішу “Enter”; – баланс користувача змінюється, тарнзакція виконана, інший користувач отримав кошти.

Кінець таблиці 3.5

Extensions	Користувач має недостатню кількість коштів на рахунку: – користувач вводить команду статистики; – користувач перевіряє свій поточний баланс; – користувач шукає команду у документації; – користувач вводить команду для переказу коштів; – користувач заповнює отримувача та суму відправки і за бажанням причину транзакції; – користувач натискає клавішу “Enter”; – при створені транзакції виникає помилка, недостатня кількість коштів; – система повідомляє користувача, про недостатню кількість кошті.
Special Requirements	Швидкість доступу до мережі Інтернет клієнту має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	80%
Miscellaneous	Чи потрібно оброблювати команду, якщо користувач вказав відсутнього на сервері отримувача

Користувач спершу викликає команду статистики, за якою бот звертається до бази даних і повертає поточний баланс. Після цього, знайшовши в документації команду переказу, користувач указує отримувача та суму і підтверджує операцію. Бот змінює баланси обох учасників у базі даних і надсилає сповіщення про успішну транзакцію. Сценарій передбачає й альтернативні перебіги: за недостатньої кількості коштів або за відсутності отримувача на сервері система інформує користувача про відповідну помилку.

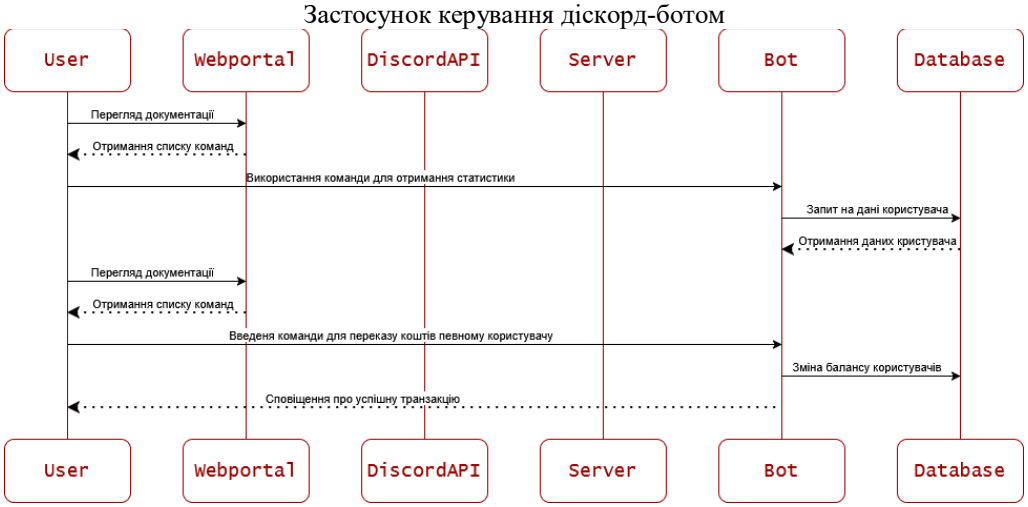

Рисунок 3.7 – Діаграма послідовності для сценарію переказу коштів

У сукупності ці сценарії покривають як адміністративні, так і користувацькі можливості застосунку. Вони показують, що система не обмежується простими командами бота, а забезпечує повний цикл взаємодії – від первинного підключення сервера до щоденного керування ролями, кімнатами, модерацією та внутрішньою економікою.

3.5 Проєктування структури класів

Внутрішню структуру програмного забезпечення на рівні класів відображає діаграма класів (рис. 3.9). З огляду на розподілений характер системи, діаграму поділено на три логічні групи, що відповідають трьом самостійним частинам застосунку – проміжному серверу на Node.js, ядру бота на disnake та вебпорталу на Laravel.

Серверну частину на Node.js утворює набір контролерів, кожен з яких відповідає за окрему операцію з даними. Контролери setAdminController, setModerController та setBotInfoController обробляють запити на зміну налаштувань – призначення адміністратора, модератора й оновлення інформації про бота відповідно. Контролери getTransactionController, getModLogController, getRolesController і getUserController забезпечують читання даних – переліку транзакцій, журналу модерації, інформації про ролі та профілю користувача. Усі вони мають посилання на спільний об'єкт database, що інкапсулює шлях до файлу бази та саме з'єднання. Клас server відповідає за ініціалізацію вебсервера Express,

зберігання токена й порту, а його метод checkAuth перевіряє автентичність кожного вхідного запиту перед подальшою обробкою.

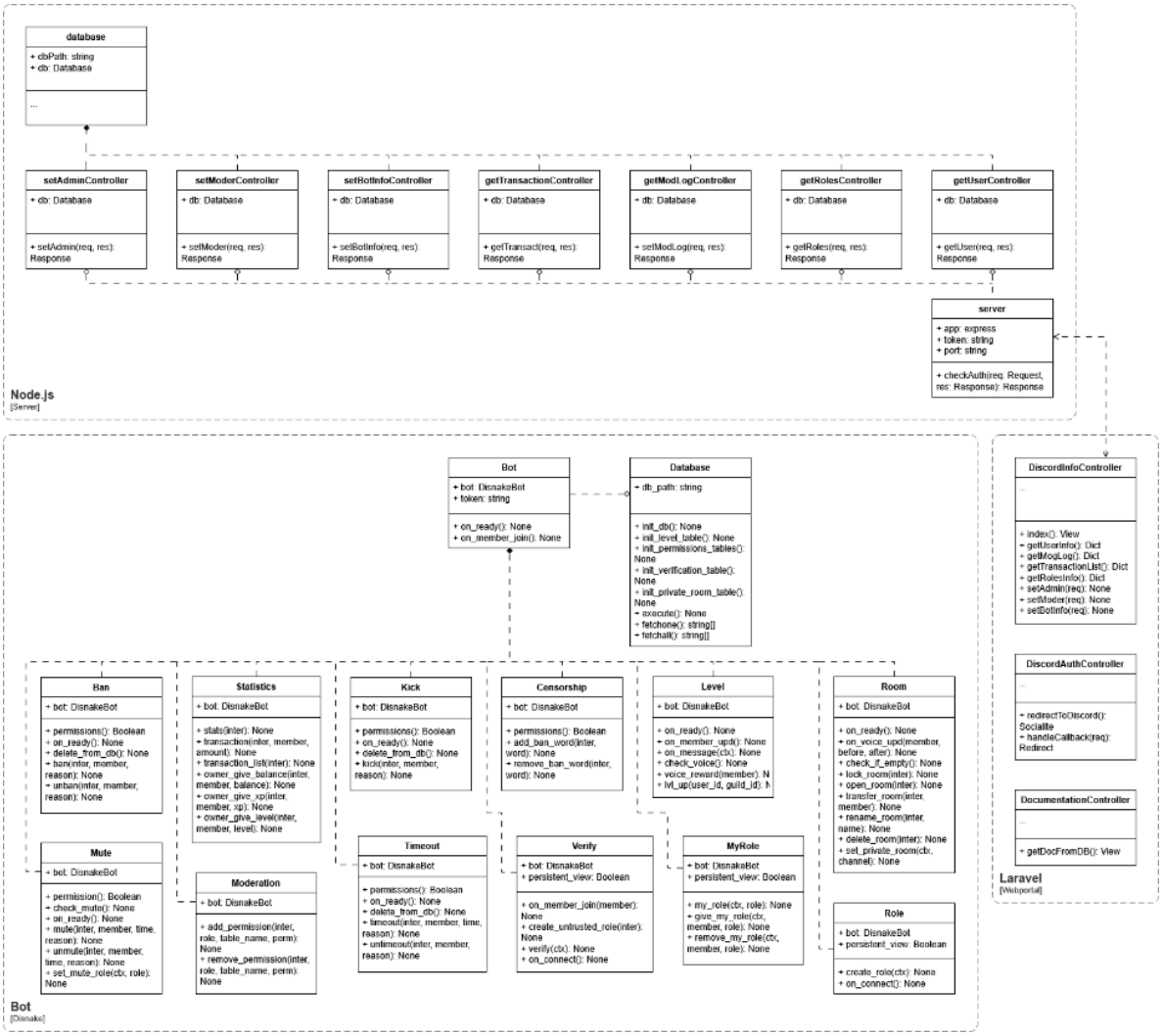

Рисунок 3.8 – Діаграма класів

Ядро бота побудовано навколо класу Bot, що зберігає екземпляр клієнта disnake і токен та реалізує базові обробники подій on_ready і on_member_join. Доступ до даних інкапсульовано у класі Database, який містить методи ініціалізації окремих таблиць і універсальні методи виконання запитів execute, fetchone та fetchall. Функціональність бота розподілено між Cog-класами, згрупованими у три модулі, позначені на діаграмі компонентів.

До модуля модерації належать класи Ban, Mute, Kick, Timeout, Censorship і Moderation. Класи покарань мають однотипну структуру – метод permissions для перевірки прав, обробник on_ready, метод видалення застарілих записів та власне

команди застосування й скасування санкції. Клас Censorship реалізує фільтр заборонених слів через методи add_ban_word і remove_ban_word, а клас Moderation керує гнучким призначенням прав ролям через add_permission і remove_permission, що дозволяє делегувати модераційні дії без видачі стандартних прав адміністратора Discord.

Користувацький модуль об'єднує класи Statistics, Level, MyRole, Role і Verify. Клас Statistics відповідає за перегляд статистики та операції з внутрішньою валютою, метод transaction виконує переказ коштів, transaction_list повертає історію операцій, а методи owner_give_balance, owner_give_xp і owner_give_level дозволяють власнику вручну коригувати показники учасника. Клас Level реалізує систему досвіду та рівнів – нарахування досвіду за повідомлення й активність у голосових каналах та підвищення рівня. Класи Role і MyRole забезпечують створення та управління особистими ролями, а клас Verify реалізує процес верифікації нових учасників через генерацію коду й видачу обмеженої ролі.

Модуль зручності представлений класом Room, що відповідає за приватні голосові кімнати. Його обробник відстежує переміщення учасників між каналами, метод set_private_room призначає канал-тригер, а методи lock_room, open_room, transfer_room, rename_room і delete_room надають користувачеві повний контроль над створеною кімнатою. Метод check_if_empty автоматично видаляє кімнату, коли її залишає останній учасник.

Вебпортал на Laravel представлено трьома контролерами. DiscordAuthController відповідає за авторизацію через OAuth2, метод redirectToDiscord перенаправляє користувача на сторінку входу Discord за допомогою пакета Socialite, а handleCallback обробляє відповідь і завершує автентифікацію. DiscordInfoController є основним контролером панелі адміністрування – він формує головну сторінку (index), повертає дані користувача, журнал модерації, перелік транзакцій та інформацію про ролі, а також передає на проміжний сервер запити на зміну налаштувань. DocumentationController через метод getDocFromDB відображає сторінку документації з переліком доступних команд.

Класи покарань та інтерфейсні класи, що працюють із кнопками й модальними вікнами, позначеними атрибутом persistent_view, який вказує на використання постійних інтерфейсних елементів disnake. Це гарантує, що елементи керування залишаються активними навіть після перезапуску бота, а отже забезпечується безперервність обслуговування користувачів.

3.6 Опис інтерфейсу користувача

Зовнішній вигляд вебінтерфейсу спроєктовано у вигляді трьох основних сторінок – вітальної, адміністрування та документації. Усі сторінки виконано в єдиному темному стилі з моноширинним шрифтом, що забезпечує візуальну цілісність і неперевантажений вигляд відповідно до сформульованих вимог до інтерфейсу.

Вітальна сторінка (рис. 3.9) є точкою входу до системи. Вона містить коротку назву сервісу у верхній панелі, логотип і назву бота, стислий опис його можливостей, а також дві основні дії – кнопку “Add to discord” для додавання бота на сервер та кнопку “Documentation” для переходу до документації. Така сторінка не перевантажує користувача зайвими елементами й одразу спрямовує його до ключових сценаріїв роботи.

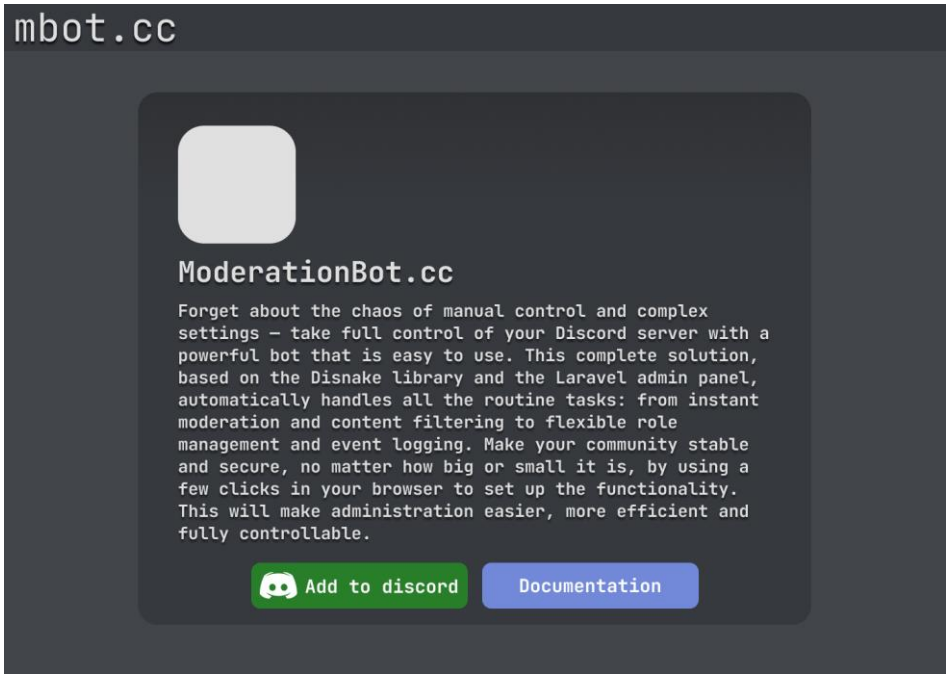

Рисунок 3.9 – Вигляд сторінки “welcome”

Сторінка адміністрування (рис. 3.10) є центральним робочим простором власника та модераторів. У лівій частині розташовано панель з даними поточного користувача – аватар, ім'я, роль на сервері та зведену статистику – рівень, кількість досвіду, баланс, кількість повідомлень, накладених мутів, таймаутів, попереджень і банів, час перебування в голосових каналах та дату приєднання. Права частина містить випадний список серверів і панель налаштувань обраного сервера, де відображається ім'я бота та поточний статус його роботи. Налаштування згруповано у “drowdown” блоки – “Permissions”, “Moderation”, “Punishments”, “Verification”, “Roles” і “Economy”, кожен з яких відповідає окремому модулю бота. У розгорнутому блоці прав, наприклад, можна обрати роль і визначити, чи надавати їй права адміністратора. Групування налаштувань за модулями робить інтерфейс структурованим і дозволяє швидко знаходити потрібні параметри.

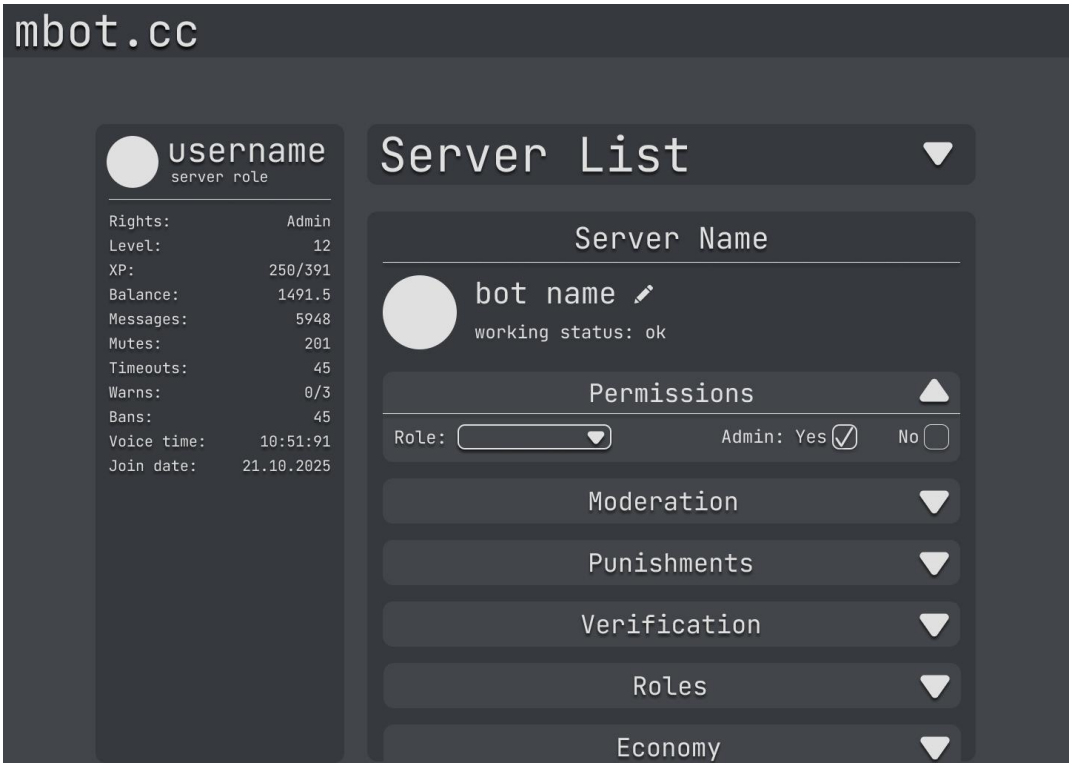

Рисунок 3.10 – Вигляд сторінки “administration”

Сторінка документації (рис. 3.11) призначена для довідки щодо доступних команд бота. У лівій частині розміщено перелік команд, а права частина відображає опис обраної команди – її назву, перелік параметрів у вигляді окремих елементів і докладний текстовий опис призначення команди та особливостей її застосування.

Саме до цієї сторінки звертаються користувачі й модератори у більшості сценаріїв, перш ніж скористатися відповідною командою.

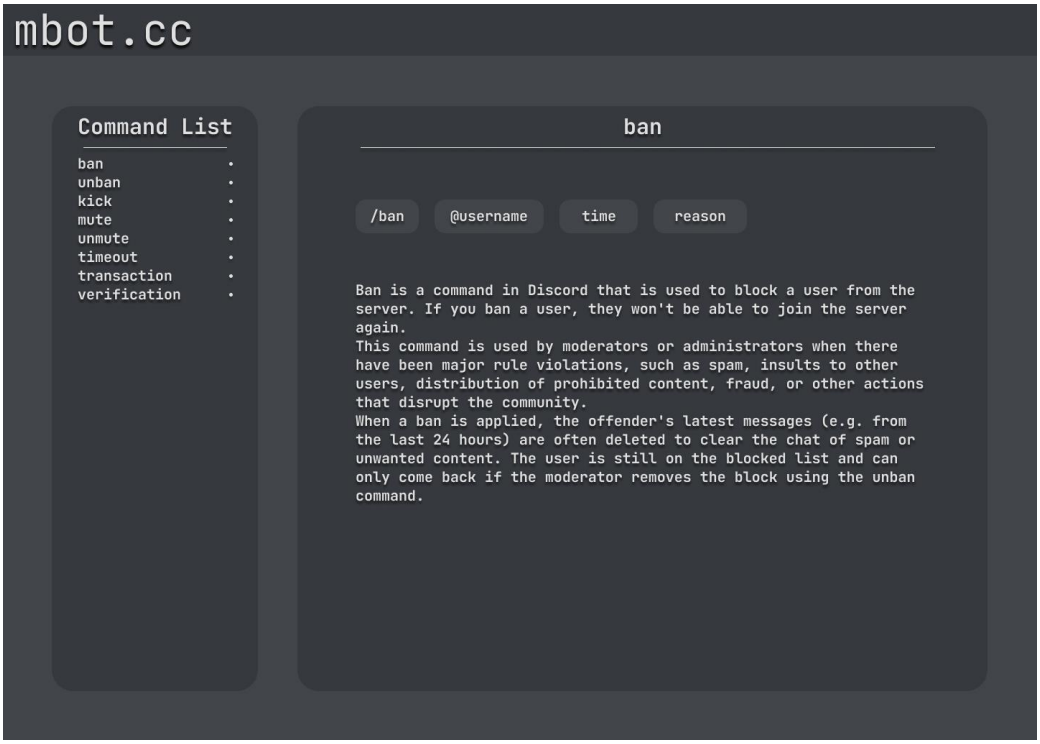

Рисунок 3.11 – Вигляд сторінки “documentation”

Спроєктований інтерфейс користувача забезпечує швидкий доступ до основних функцій системи та зберігає єдиний стиль оформлення для всіх сторінок вебпорталу. Розподіл функціональності між окремими сторінками дозволяє уникнути перевантаження інтерфейсу, а групування елементів за функціональними модулями спрощує навігацію та взаємодію користувача із системою. Використання темної кольорової схеми та мінімалістичного оформлення також відповідає загальному стилю платформи Discord, що робить роботу із застосунком більш зрозумілою та зручною для користувачів.

3.7 Розгортання та взаємодія компонентів системи

Систему розгорнуто у вигляді кількох незалежних вузлів. Клієнт звертається до вебпорталу через браузер за протоколом HTTP. Вебпортал, реалізований як застосунок Laravel, передає запити проміжному сервісу на платформі Node.js через REST API. Проміжний сервіс безпосередньо працює з базою даних SQLite за

допомогою SQL-запитів. Паралельно бот, реалізований засобами бібліотеки disnake, підтримує постійне з'єднання з Discord через шлюз і також звертається до тієї самої бази даних. Узгодженість стану системи в такий спосіб забезпечується через спільне сховище. Зміни, внесені адміністратором через вебпортал, потрапляють до бази даних і стають доступними боту, а результати дій бота, навпаки, відображаються у вебінтерфейсі.

Така схема розгортання дає змогу розділити відповідальність між окремими вузлами й не змішувати вебчастину, бот та проміжний сервіс в одному процесі. Це спрощує супровід системи, бо кожен компонент можна оновлювати незалежно від інших, а взаємодія між ними лишається зрозумілою та передбачуваною.

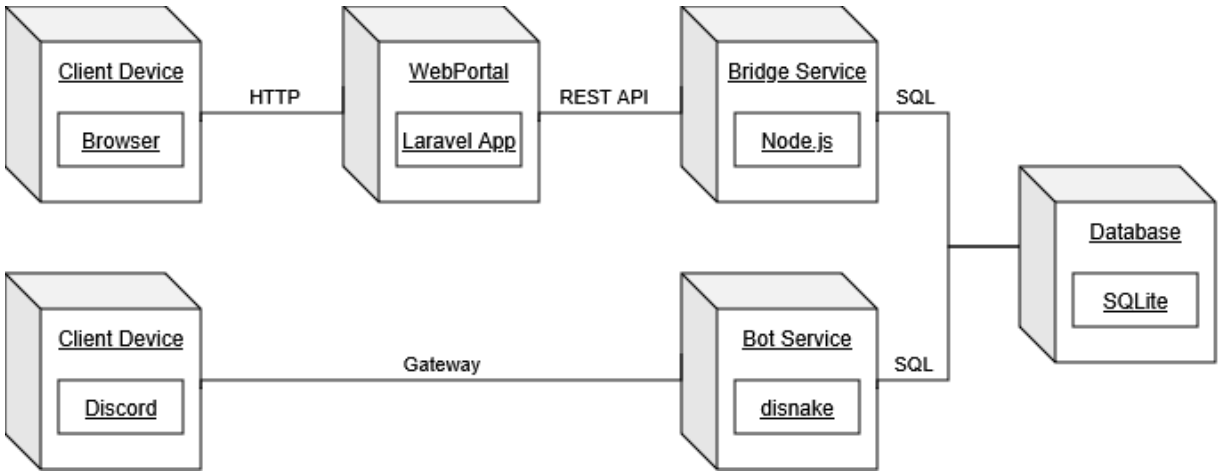

Рисунок 3.12 – Діаграма розгортання

Винесення взаємодії вебпорталу з базою даних в окремий проміжний сервіс на Node.js дозволяє розмежувати технологічні стеки бота й вебчастини та зосередити логіку доступу до даних в одному місці. Крім того, такий підхід спрощує контроль прав. Усі запити від вебпорталу проходять через єдину точку перевірки автентифікації, перш ніж потрапити до сховища.

У такій архітектурі зміни, внесені з боку вебпорталу, швидко потрапляють до спільного сховища, а бот одразу працює вже з актуальними даними. Завдяки цьому користувач отримує узгоджену поведінку системи незалежно від того, чи він виконує дію через сайт, чи безпосередньо в Discord.

Висновки до розділу 3

У розділі представлено результати проєктування та конструювання програмного забезпечення застосунку керування діскорд-ботом, отримані на основі сформованої раніше специфікації вимог.

Розроблено архітектуру системи, яка має шарову й водночас розподілену структуру та складається з чотирьох незалежних частин – вебпорталу на Laravel, ядра бота на disnake, проміжного сервера взаємодії на Node.js і спільного сховища даних на SQLite. Побудовано діаграми компонентів і розгортання, що відображають як логічний поділ системи на шари, так і фізичне розміщення її складових та канали обміну даними між ними.

Обґрунтовано вибір технологій, мов програмування й програмних компонентів – поєднання Python, PHP, JavaScript і TypeScript, а також застосування патернів Cog, серверного рендерингу, постійних інтерфейсних елементів та єдиної точки доступу до даних.

Засобами UML змодельовано функціональні та поведінкові аспекти системи. Діаграма використання відображає три ролі користувачів і доступні їм сценарії, а п'ять докладних сценаріїв використання разом із діаграмами послідовності – додавання бота, створення приватної кімнати, створення приватної ролі, покарання користувача та переказ коштів, деталізують обмін повідомленнями між компонентами. Діаграма класів описує внутрішню структуру всіх трьох частин застосунку на рівні класів та їхніх методів.

Спроєктовано інтерфейс користувача у вигляді трьох сторінок вебпорталу – вітальної, адміністрування та документації, виконаних у єдиному стилі та згрупованих за функціональними модулями бота. Таким чином, отримані проєктні рішення повністю охоплюють структуру, поведінку й зовнішній вигляд застосунку та становлять достатню основу для його безпосередньої програмної реалізації.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ

4.1 Розробка елементів управління ботом

Ядро бота реалізовано мовою Python із використанням бібліотеки disnake та модульної Cog-архітектури. Точкою входу є модуль main.py, який створює екземпляр бота з префіксом команд і повним набором intents, підключає базу даних та динамічно завантажує всі модулі з каталогу cogs. Такий підхід дозволяє вмикати чи вимикати окремі функціональні блоки без зміни решти коду.

Під час створення екземпляра бота йому передають повний набір intents, що відкриває доступ до подій про повідомлення, учасників і голосові стани – без них модулі цензури, економіки й приватних кімнат не отримували б потрібних сповіщень від Discord. Обробник події on_ready, який спрацьовує одразу після підключення, перебирає всіх учасників усіх серверів і додає до таблиці user початковий запис для кожного, хто його ще не має, інструкцією INSERT OR IGNORE. Аналогічний запис створює й обробник on_member_join у момент приєднання нового учасника. Завдяки цьому жоден користувач не залишається без облікового рядка навіть тоді, коли він приєднався під час простою бота.

Завантаження функціональних модулів виконується динамічно, під час запуску застосунок перебирає файли каталогу cogs і реєструє кожен як розширення бота, тож для додавання нової підсистеми достатньо помістити відповідний файл у каталог, не змінюючи точку входу. Універсальні методи класу BotDatabase – execute для запитів без повернення даних, fetchone для одного рядка та fetchall для набору рядків – застосовуються в усіх Cog-модулях через посилання self.bot.db, що робить його єдиним каналом доступу до сховища в межах ядра бота.

Доступ до сховища інкапсульовано у класі BotDatabase – обгортці над стандартною бібліотекою sqlite3. У конструкторі клас приймає шлях до файлу бази та послідовно викликає методи ініціалізації таблиць – основних, довідника рівнів, таблиць прав, верифікації та приватних кімнат. Особливістю реалізації є те, що кожен запит відкриває й закриває окреме з'єднання. У поєднанні з режимом

журналювання WAL це дозволяє боту та проміжному серверу одночасно читати й записувати спільний файл бази без взаємних блокувань. Склад класу та універсальні методи виконання запитів наведено нижче:

```
class BotDatabase:
    def __init__(self, db_path):
        self.db_path = db_path
        self.init_db()
        self.init_level_table()
        self.init_permissions_tables()
        self.init_verification_table()

    def _connect(self):
        conn = sqlite3.connect(self.db_path)
        conn.execute('PRAGMA journal_mode=WAL')
        conn.execute('PRAGMA busy_timeout=5000')
        return conn

    def execute(self, query, parameters=()):
        conn = self._connect()
        conn.execute(query, parameters)
        conn.commit()
        conn.close()

    def fetchone(self, query, parameters=()):
        conn = self._connect()
        row = conn.execute(query, parameters).fetchone()
        conn.close()
        return row
```

Найбільш навантаженою підсистемою бота є модуль рівнів, реалізований класом Level. Він нараховує досвід за активність користувачів – як за надіслані текстові повідомлення, так і за перебування в голосових каналах. Обробник події on_message спершу перевіряє повідомлення на наявність заборонених слів, після чого нараховує досвід, пропорційний довжині повідомлення та множнику booster, і викликає метод підвищення рівня. Перебування в голосовому каналі обробляється фоновою задачею, що кожні п'ять секунд нараховує фіксований обсяг досвіду активним учасникам.

Метод level_up реалізує підвищення рівня у вигляді циклу. Він порівнює поточний досвід користувача з пороговим значенням для наступного рівня з таблиці level_table і, якщо поріг досягнуто, збільшує рівень, нараховує винагороду та віднімає витрачений досвід. Цикл повторюється доти, доки накопиченого досвіду вистачає на наступний рівень, що дозволяє користувачеві піднятися одразу на кілька рівнів за одну дію. Реалізацію нарахування досвіду та підвищення рівня наведено нижче:

```
    @commands.Cog.listener()
    async def on_message(self, message):
        words = self.bot.db.fetchall(
            'SELECT word FROM censored_words WHERE guild_id = ?',
            (message.guild.id,))
        if any(w[0] in message.content for w in words):
            await message.delete()
            return
        booster = self.get_booster(message.author)
        self.bot.db.execute(
            'UPDATE user SET xp = xp + ? WHERE user_id = ? AND guild_id = ?',
            (booster       *       len(message.content),       message.author.id,
message.guild.id))
        self.level_up(message.author.id, message.guild.id)

    def level_up(self, user_id, guild_id):
        while True:
            level, xp = self.bot.db.fetchone(
                'SELECT level, xp FROM user WHERE user_id=? AND guild_id=?',
                (user_id, guild_id))
            need, gold = self.bot.db.fetchone(
                'SELECT xp, gold FROM level_table WHERE level = ?', (level + 1,))
            if xp < need:
                break
            self.bot.db.execute(
                'UPDATE user SET level = level + 1, xp = xp - ?, '
                'balance = balance + ? WHERE user_id=? AND guild_id=?',
                (need, gold, user_id, guild_id))
```

Підсистему модерації побудовано за єдиним шаблоном – кожен клас покарання перевіряє наявність відповідного права через декоратор, фіксує дію у власній таблиці, оновлює лічильник у таблиці user і сповіщає порушника. Наприклад, команда /mute обчислює час зняття покарання, видає учаснику mute-роль і створює запис у таблиці mute, а фонова задача check_mute щохвилини знімає муту, термін якої сплив. Завдяки винесенню перевірки прав у декоратор код команд лишається лаконічним та однотипним для всіх видів санкцій.

Множник booster, що визначає темп нарахування досвіду, перераховується обробником on_member_update у момент зміни набору ролей учасника. Обробник підсумовує значення role_booster усіх наявних у нього ролей, додає їх до базової одиниці й записує результат у поле booster таблиці user. Відтак придбання або втрата ролі з множником одразу позначаються на швидкості зростання рівня. Нарахування досвіду за голос реалізовано двояко – фоновою задачею check_voice, що кожні п'ять секунд оглядає всіх учасників у голосових каналах, і обробником on_voice_state_update, який нараховує досвід негайно при вході в канал. В обох випадках метод voice_reward пропускає учасників із вимкненим мікрофоном чи

звуком або застосованим заглушенням, щоб винагороджувати лише фактичну присутність.

Класи покарань мають спільну будову, проте різняться деталями життєвого циклу записів. Клас Mute реалізує рольовий мут, команда видає учаснику окрему mute-роль, обчислює час зняття як суму поточного моменту й заданої тривалості та зберігає його рядком, а фонова задача щохвилини знімає муту, термін якої сплив. Клас Timeout натомість застосовує вбудований механізм таймауту Discord і синхронізує власну таблицю з фактичним станом платформи. Класи Ban і Kick зберігають ім'я порушника як знімок на момент санкції та періодично очищають застарілі записи – для банів раз на добу за умови повернення учасника, для кіків – за його відсутності на сервері. Перевірку прав в усіх класах винесено в окремий декоратор, який звіряє ролі автора команди з відповідною таблицею прав, тож самі команди лишаються короткими й однотипними.

Користувацький модуль зосереджує функції, доступні звичайним учасникам. Клас Role реалізує купівлю кастомної ролі через модальне вікно з трьома полями – назвою, кольором у форматі HEX і множником досвіду. Вартість обчислюється від базових 1000 одиниць із надбавкою за підвищений множник, а після підтвердження бот створює роль у Discord, списує кошти та вносить запис до таблиці role_table. Клас MyRole дає власникові ролі змінювати її назву, колір, множник, позицію в ієрархії й видимість, а також видавати роль іншим учасникам, оновлюючи перелік у полі role_has. Клас Statistics відповідає за перегляд зведеної статистики та перекази внутрішньої валюти. Метод transaction перевіряє достатність балансу й мінімальну суму переказу, змінює баланси обох сторін і фіксує операцію в таблиці transaction_table, а метод transaction_list повертає останні операції учасника.

Модуль зручності та верифікації завершує функціональний склад ядра. Клас Verify видає кожному новому учаснику обмежену роль, що приховує всі канали, окрім каналу верифікації, і знімає її після введення згенерованого коду. Клас Room обслуговує приватні голосові кімнати – коли учасник заходить у попередньо призначений канал-тригер, бот створює для нього окрему кімнату, переносить туди учасника й заносить її до таблиці temp_channel, а фонова задача check_if_empty

видаляє кімнату, щойно вона спорожніє. Власникові кімнати доступні команди блокування, відкриття, перейменування, передавання прав і видалення, що дає повний контроль над створеним каналом.

4.2 Створення інструментів для серверної взаємодії

Проміжний сервер взаємодії реалізовано на платформі Node.js мовою TypeScript із використанням каркаса Express. Його призначення – надати вебпорталу безпечний HTTP-інтерфейс до спільної бази даних, не відкриваючи прямого доступу до файлу сховища. Сервер створює єдине з'єднання з базою через драйвер better-sqlite3 з тими самими налаштуваннями WAL і busy_timeout, що й бот, та експортує це з'єднання для всіх контролерів.

Шлях до файлу бази проміжний сервер обчислює відносно власного розташування, піднімаючись на два рівні вгору до каталогу бота, тому коректна робота сервера передбачає сусіднє розміщення обох компонентів у межах спільної директорії проєкту. Драйвер better-sqlite3, на відміну від асинхронних аналогів, працює синхронно, що спрощує код контролерів, позбавляючи їх зворотних викликів і конструкцій async/await під час звернень до бази. Оскільки Node.js виконується в одному потоці, єдине спільне з'єднання не створює конкурентних конфліктів, а узгоджений із ботом режим WAL дозволяє обом процесам безпечно працювати з файлом одночасно.

Точкою входу сервера є модуль server.ts, який створює застосунок Express, реєструє проміжний обробник автентифікації й підключає маршрути. Усі маршрути доступні лише за методом POST і захищені єдиним механізмом перевірки. Функція checkAuth порівнює заголовок Authorization кожного запиту зі значенням токена з файлу оточення, а за невідповідності повертає відповідь зі статусом 401. Такий централізований контроль гарантує, що звертатися до бази через сервер може лише довірений вебпортал. Реалізацію перевірки токена та реєстрації маршрутів наведено нижче:

```
function checkAuth(req: Request, res: Response, next: NextFunction) {
    const header = req.headers.authorization;
    if (header === `Bearer ${process.env.API_TOKEN}`) {
        return next();
```

```
    }
    return res.status(401).json({ error: 'Invalid token' });
}

const app = express();
app.use(express.json());
app.use(checkAuth);

app.post('/api/user/info', getUserInfo);
app.post('/api/punishments', getPunishmentsController);
app.post('/api/set-admin', setAdminController);
app.post('/api/update-user-stats', updateUserStatsController);

app.listen(process.env.PORT, () =>
    console.log(`Bridge server started on port ${process.env.PORT}`));
```

Бізнес-логіку розподілено між незалежними контролерами, кожен з яких відповідає за одну операцію – отримання інформації про користувача, перелік користувачів, перегляд покарань, призначення та зняття прав, оновлення статистики й бустерів ролей. Спільною технічною особливістю контролерів читання є приведення числових ідентифікаторів Discord до текстового типу. Ідентифікатори платформи перевищують максимально безпечне ціле число JavaScript, тому без приведення до рядка вони спотворювалися б під час передавання у форматі JSON.

Найскладнішу логіку містить контролер userController, що формує повну відповідь про користувача. Він зчитує статистику з таблиці user, визначає потрібний для наступного рівня досвід і для кожної з п'яти таблиць прав перевіряє, чи належить хоча б одна з ролей користувача до цієї таблиці. Перевірку реалізовано динамічним SQL-запитом з оператором IN, кількість параметрів якого відповідає кількості ролей, а сам масив ролей передається у запит через spread-оператор. Відповідну реалізацію наведено нижче:

```
function checkPermission(table: string, guildId: string,
                         roleIds: string[]): boolean {
    if (roleIds.length === 0) return false;
    const placeholders = roleIds.map(() => '?').join(',');
    const row = db.prepare(
        `SELECT 1 FROM ${table} ` +
        `WHERE guild_id = ? AND role_id IN (${placeholders}) LIMIT 1`
    ).get(guildId, ...roleIds);
    return row !== undefined;
}

export function getUserInfo(req: Request, res: Response) {
    const { user_id, guild_id, role_ids } = req.body;
    const stats = db.prepare(
        'SELECT level, xp, balance, mutes, bans, timeouts, warns ' +
        'FROM user WHERE user_id = ? AND guild_id = ?').get(user_id, guild_id);
```

```
        const permissions = {
            ban_permission:      checkPermission('ban_permission',      guild_id,
role_ids),
            mute_permission:     checkPermission('mute_permission',     guild_id,
role_ids),
        };
        res.json({ user_id, guild_id, stats, permissions });
    }
```

Контролери запису згруповано за призначенням. Контролер призначення адміністратора додає роль одразу до всіх п’яти таблиць прав у межах однієї транзакції better-sqlite3, що гарантує атомарність операції, тоді як контролер модератора спершу очищає всі права ролі, а потім надає лише вибрані. Використання транзакцій та інструкції INSERT OR IGNORE убезпечує базу від часткових змін і повторних вставок однакових записів.

Контролери читання, окрім userController, обслуговують решту потреб панелі. Контролер переліку учасників повертає для заданого сервера їхні баланси, рівні та досвід, контролер покарань виконує чотири окремі вибірки за таблицями мутів, банів, таймаутів і кіків, а контролер таблиці ролей повертає кастомні ролі разом із полем role_has. Спільною рисою цих контролерів є приведення всіх ідентифікаторів Discord до текстового типу інструкцією CAST. Ідентифікатори платформи є 64-бітними числами, що перевищують межу точного подання цілих у JavaScript, тому без перетворення на рядок вони втрачали б останні цифри під час серіалізації у JSON. Самих імен учасників і ролей проміжний сервер не визначає – це завдання покладено на вебпортал, що має доступ до Discord API.

Контролери запису згруповано за призначенням і захищено від типових порушень цілісності. Контролер призначення адміністратора в межах однієї транзакції додає роль одразу до всіх п’яти таблиць прав, а контролер модератора спершу повністю очищає права ролі, а потім надає лише вибрані, фактично перезаписуючи її набір повноважень. Перелік прав, що надаються модераторові, попередньо звіряється з фіксованим набором допустимих назв, чим унеможливлюється підстановка довільного імені таблиці у запит. Контролер mute-ролі застосовує інструкцію INSERT OR REPLACE, оскільки на сервері може існувати лише одна така роль, а контролери оновлення статистики й множника ролі перед записом перевіряють коректність вхідних значень – невід'ємність балансу та

досвіду, рівень не менший за одиницю й множник не менший за одиницю. Контролер верифікації повертає ідентифікатор ролі обмеженого доступу або порожнє значення, якщо верифікацію на сервері ще не налаштовано.

4.3 Проєктування вебпорталу для управління ботом

Вебпортал адміністрування реалізовано на PHP-фреймворку Laravel із застосуванням серверного рендерингу шаблонів Blade та клієнтських сценаріїв на JavaScript. Маршрутизацію зосереджено у файлі web.php, де згруповано публічні сторінки, маршрути авторизації через Discord і захищену групу адміністративних маршрутів. Усі адміністративні маршрути мають іменовані аліаси, що дозволяє формувати їхні адреси у шаблонах і скриптах через хелпер route() без жорсткого кодування URL.

Авторизацію реалізовано за протоколом OAuth2 за допомогою пакета Laravel Socialite. Контролер DiscordAuthController перенаправляє користувача на сторінку згоди Discord із потрібним набором дозволів, а після повернення зчитує дані користувача, визначає обраний сервер, запитує в Discord API назву сервера та ролі учасника і зберігає їх у сесії. Доступ до адміністративних маршрутів контролює проміжний обробник DiscordAuthMiddleware, який пропускає запит лише за наявності в сесії даних користувача та ідентифікатора сервера, інакше повертаючи користувача на головну сторінку.

Параметри зовнішніх сервісів зосереджено у конфігураційному файлі services.php, звідки їх зчитують через хелпер config. Секція discord містить ідентифікатор і секрет застосунку, адресу зворотного виклику та набір прав, що запитуються для бота, а секція node_bridge – адресу проміжного сервера й токен доступу до нього. Маршрути вебпорталу описано у файлі web.php, де публічні сторінки відокремлено від групи адміністративних маршрутів, захищеної проміжним обробником. Кожен адміністративний маршрут має іменований аліас, завдяки чому його адресу формують у шаблонах і скриптах через хелпер route, не вписуючи URL безпосередньо в код.

Процес авторизації через Socialite запитує в Discord набір дозволів на ідентифікацію користувача, перелік його серверів, додавання бота та реєстрацію команд, а параметр згоди змушує платформу щоразу показувати екран вибору сервера, що дозволяє адміністраторові перемикатися між серверами. Після повернення з Discord контролер обмінює отриманий код на профіль користувача, додатковими запитами до Discord API отримує назву обраного сервера й ролі учасника та зберігає в сесії шість значень – профіль, ідентифікатор сервера, його назву та іконку, перелік ролей і дату приєднання. Саме ці сесійні дані надалі використовують усі методи панелі, а проміжний обробник пропускає запит лише за наявності в сесії профілю користувача й ідентифікатора сервера.

Контролер UserInfoController є вузловим компонентом панелі й містить близько десятка методів. Метод формування головної сторінки звертається до проміжного сервера за повними даними користувача й на основі набору прав визначає його роль на сервері. Коли роль має всі п'ять дозволів, вона позначається як адміністративна, за повної відсутності дозволів – як звичайний користувач, а в проміжних випадках – як модератор. Окремі методи постачають клієнтові перелік ролей сервера, журнал покарань, таблицю кастомних ролей, список учасників і стан верифікації, тоді як методи запису перевіряють вхідні дані й пересилають їх проміжному серверу, доповнюючи службовими параметрами із сесії.

Центральним є контролер UserInfoController, що формує сторінку панелі та обслуговує запити до даних. Він виступає посередником між браузером і проміжним сервером, приймає звернення від клієнтського скрипту, додає до них службові параметри і пересилає на відповідний маршрут проміжного сервера. Особливість методів перегляду покарань та ролей полягає у тому, що проміжний сервер повертає лише числові ідентифікатори Discord, які потрібно перетворити на читабельні імена. Для цього контролер збирає унікальні ідентифікатори й виконує паралельні запити до Discord API через механізм Http::pool, що суттєво скорочує час відповіді порівняно з послідовними зверненнями. Відповідну реалізацію наведено нижче:

```
public function getPunishments()
{
```

```
        $guildId = session('guild_id');
        $data = Http::withToken(config('services.node_bridge.token'))
            ->post(config('services.node_bridge.url') . '/api/punishments',
                   ['guild_id' => $guildId])->json();

        $ids = $this->collectDiscordIds($data);
        $botToken = config('services.discord.bot_token');

        $responses = Http::pool(fn ($pool) =>
            collect($ids)->map(fn ($id) =>
                $pool->as($id)->withToken($botToken, 'Bot')
                     -
>get("https://discord.com/api/guilds/{$guildId}/members/{$id}")
            )->all());

        $names = [];
        foreach ($ids as $id) {
            $names[$id] = $responses[$id]->json('user.username') ?? $id;
        }
        return response()->json($this->formatPunishments($data, $names));
    }
```

Подання даних реалізовано шаблоном administration-page, що складається з двох частин – панелі поточного користувача та панелі налаштувань сервера, згрупованих у шість акордеон-блоків за модулями бота. Зв'язок між серверним шаблоном і клієнтським кодом забезпечує об'єкт window_adminConfig, у який передаються адреси всіх маршрутів і CSRF-токен. Завдяки цьому клієнтський скрипт не містить жорстко закодованих адрес та повністю керується конфігурацією, сформованою на сервері.

Клієнтську логіку панелі реалізовано у скрипті administration.js. Він керує розгортанням акордеон-блоків, завантажує дані відповідного розділу за першим відкриттям і надсилає зміни на сервер асинхронними запитами fetch. Дані кешуються після першого завантаження, що зменшує кількість звернень до сервера. Приклад функції оновлення статистики користувача, яка зчитує введені значення з полів таблиці та надсилає їх на сервер, наведено нижче:

```
async function updateUserStats(btn) {
    const row = btn.closest('tr');
    const payload = { user_id: row.dataset.userId };
    row.querySelectorAll('input[data-field]').forEach(input => {
        payload[input.dataset.field] = input.value;
    });

    const response = await fetch(updateUserStatsUrl, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'X-CSRF-TOKEN': csrfToken,
        },
        body: JSON.stringify(payload),
```

```
        });

        if (response.ok) {
            btn.classList.add('success');
            setTimeout(() => btn.classList.remove('success'), 1200);
        }
    }
```

Панель налаштувань поділено на шість акордеон-блоків, що відповідають окремим підсистемам бота – керування правами, налаштування модераторів і mute-ролі, перегляд покарань, таблиця ролей, керування учасниками та стан верифікації. Кожен блок завантажує власні дані лише під час першого розгортання й кешує їх, тому повторне відкриття не спричиняє зайвих звернень до сервера. Перетворення числових ідентифікаторів на читабельні імена в блоках покарань, ролей і учасників виконує вебпортал, звертаючись до Discord API паралельними запитами, що суттєво скорочує час відповіді порівняно з послідовним опитуванням.

Поведінку акордеонів задано спільним описом усіх панелей, на основі якого функція згортання закриває решту блоків при відкритті будь-якого з них, лишаючи розгорнутим тільки один. Списки ролей у трьох випадних меню заповнює одна функція, що звертається до маршруту переліку ролей, а взаємовиключний вибір прав адміністратора реалізовано двома прапорцями, синхронізованими так, що позначення одного автоматично знімає інший. Усі запити, які змінюють стан, супроводжуються заголовком CSRF-токена, обов'язковим для захищених маршрутів Laravel, а перед виведенням отриманих із Discord даних клієнтський код екранує спеціальні символи, запобігаючи впровадженню сторонньої розмітки в інтерфейс.

4.4 Тестування взаємодії між компонентами системи

Оскільки застосунок має розподілену будову й складається з трьох процесів, що спілкуються через спільну базу даних і HTTP-API, перевірку коректності зосереджено насамперед на стиках між ними. Для цього поєднано три взаємодоповнювані підходи – функціональне тестування за принципом «чорної скриньки», коли спостережувана поведінка звіряється з очікуваною без огляду на внутрішній код, інтеграційне тестування, що відстежує цілісність даних на всьому

шляху від бота до вебпанелі та автоматизоване модульне тестування окремих компонентів, винесене в самостійні набори для кожної мови й платформи. Перші два підходи виявляють розбіжності на межах процесів, а третій фіксує регресії в бізнес-логіці ще до того, як вони потрапляють у наскрізні сценарії.

Автоматизовані тести розроблено для кожного компонента окремо, з урахуванням його технологічного стека. Ядро бота покрито тестами на pytest, проміжний сервер – на Jest у поєднанні з ts-jest і Supertest, а вебпортал – на Pest, надбудові над PHPUnit. Загалом підготовлено сім файлів тестів із вісімдесятьма трьома перевірками, кожен набір запускається однією командою в межах відповідного підпроєкту й не потребує підключення до живого сервера Discord. Перевірки охоплюють ініціалізацію бази, формули нарахування досвіду, валідацію вхідних значень, розмежування доступу та проміжні обробники авторизації обох вебкомпонентів.

Тести ядра бота поділено на два файли. Перший перевіряє клас-обгортку BotDatabase, після створення екземпляра в тимчасовому файлі контролюється наявність усіх сімнадцяти таблиць, заповнення довідника рівнів рівно ста рядками, рівність першого порога ста двадцяти восьми одиницям досвіду, зміна множника прогресії після шостого рівня та строге зростання порогів від рівня до рівня. Окрема група випробовує операції читання й запису – вставлення та вибірку учасника, захист складеного первинного ключа від дублікатів, співіснування одного user_id на різних серверах, повернення порожнього результату для відсутнього рядка та значення множника booster за замовчуванням. Другий файл відтворює логіку методу level_up у вигляді чистої функції без звернень до Discord і перевіряє підвищення рівня рівно на порозі, перенесення надлишку досвіду, стрибок одразу через кілька рівнів за один виклик, подвоєння винагороди за активного бустера й перерахунок множника із суми ролей учасника:

```
def test_level_up_at_exact_threshold(db):
    db.execute('INSERT INTO user (user_id, guild_id, level, xp) '
               'VALUES (1, 1, 1, 128)')
    simulate_level_up(db, user_id=1, guild_id=1)
    level, xp = db.fetchone(
        'SELECT level, xp FROM user '
        'WHERE user_id = 1 AND guild_id = 1')
    assert level == 2
```

```
assert xp == 0
```

Проміжний сервер тестується з ізоляцією від реальної бази – модуль з'єднання підмінюється заглушкою через jest.mock, яка повертає об'єкт із методами prepare, run, get, all і transaction, тож контролери перевіряються без файлу Bot.db. Щоб під час тестів не запускався TCP-сервер, застосунок Express винесено в окремий модуль app.ts, що експортує лише налаштований об'єкт без виклику app.listen. Саме його підхоплює Supertest. Перший набір перевіряє проміжний обробник checkAuth – запит без заголовка авторизації, із хибним токеном або без схеми Bearer повертає статус 401, тоді як коректний токен пропускає запит далі, причому перевірку повторено для всіх дванадцяти захищених маршрутів. Решта наборів зосереджено на валідації, контролер оновлення статистики приймає лише невід'ємні баланс і досвід та рівень не менший за одиницю, відхиляючи решту значень статусом 400, а контролери призначення прав звіряють обов'язкові поля й допустимий діапазон множника та підтверджують, що видача адміністративних повноважень відбувається в межах однієї транзакції.

Набори вебпорталу перевіряють захист маршрутів і валідацію форм адміністративної панелі. Тести проміжного обробника DiscordAuthMiddleware підтверджують, що без сесії всі тринадцять захищених маршрутів перенаправляють користувача на головну сторінку й що для доступу обов'язкова одночасна наявність обох сесійних ключів – профілю користувача та ідентифікатора сервера, за відсутності будь-якого з них запит відхиляється. Окремий нюанс реалізації полягав у тому, що заглушку об'єкта користувача довелося оформити іменованим класом DiscordUserStub, оскільки анонімні класи PHP не серіалізуються під час збереження сесії. Друга група тестів через хелпер authSession проходить перевірку доступу й випробовує правила валідації контролерів – відхилення від'ємного балансу, нульового чи від'ємного рівня, від'ємного досвіду та відсутнього ідентифікатора користувача, а також перевіряє, що неприпустиме значення у списку прав модератора відкидається, тоді як порожній і повний набори прав приймаються. Звернення до проміжного сервера в

цих тестах підмінено заглушкою Http::fake, що ізолює перевірку валідації від мережевих викликів.

Окрім автоматизованих наборів, проведено серію ручних наскрізних випробувань, які відтворюють типові дії адміністратора на реальному сервері Discord. Кожен такий тест-кейс описує початкові умови, виконувану дію та очікуваний результат і зачіпає одразу кілька компонентів – наприклад, зміну статистики учасника через вебпанель із подальшою перевіркою того, що бот уже оперує оновленими значеннями. Перелік основних ручних тест-кейсів і підсумок їх виконання наведено в таблиці 4.2.

Таблиця 4.1 – Тест-кейси та результати їх виконання

Кейс	**Очікуваний результат**	**Статус**
Авторизація через Discord OAuth2	Перенаправлення на згоду, збереження сесії, відкриття панелі	Пройдено
Доступ до панелі без авторизації	Перенаправлення на головну сторінку	Пройдено
Нарахування досвіду за повідомлення	Зростання xp та підвищення рівня при досягненні порогу	Пройдено
Накладення муту командою /mute	Видача ролі, запис у таблицю, інкремент лічильника	Пройдено
Автоматичне зняття муту за часом	Зняття ролі й видалення запису фоновою задачею	Пройдено

Кінець таблиці 4.2

Редагування балансу через панель	Запис нового значення в базу, видимий боту	Пройдено
Призначення прав адміністратора ролі	Додавання до 5 таблиць прав, визначення прав «Admin»	Пройдено
Запит до API без коректного токена	Відповідь зі статусом 401	Пройдено
Перегляд покарань з іменами	Заміна Discord ID іменами через паралельні запити	Пройдено

Ручні сценарії відтворювалися безпосередньо на тестовому сервері Discord, коректність HTTP-API проміжного сервера перевірялася клієнтом надсилання запитів із контролем заголовків авторизації, а узгодженість даних підтверджувалася переглядом стану таблиць бази після кожної операції. Усі заплановані випадки пройдено успішно. Дефекти, виявлені на ранніх ітераціях, мали технічну природу – спотворення довгих ідентифікаторів Discord під час серіалізації в JSON усунено приведенням їх до рядка інструкцією CAST, а поодинокі блокування при одночасному доступі бота й сервера до файлу бази зникли після переходу на режим журналювання WAL.

4.5 Результати тестування та оцінювання якості програмного забезпечення

Результати тестування охоплюють як підсумки прогону автоматизованих наборів, так і виміри нефункціональних характеристик – швидкодії проміжного сервера під навантаженням і поведінки підсистеми рівнів за різних вхідних даних. Дані подано у вигляді зведених таблиць і графіків із подальшою інтерпретацією.

Повний прогін автоматизованих тестів завершився без жодної помилки – усі вісімдесят три перевірки пройдено. Набір бота виконується за лічені секунди й охоплює тридцять тестів, набір проміжного сервера – тридцять два, а набір вебпорталу – двадцять один тест із сімдесятьма трьома утвердженнями. Розподіл

перевірок за компонентами, застосованими фреймворками та кількістю файлів наведено в таблиці 4.2.

Таблиця 4.2 – Результат виконання тестів

Компонент	**Засіб тестування**	**Кількість тестів**	**Вдалих тестів**	**Результат**
Ядро бота	pytest 9.0	30	30	Успішно
Сервер	Jest 30, Supertest	32	32	Успішно
Вебпорал	Pest, PHP Unit 11	21	21	Успішно

Автоматизовані набори охоплюють найбільш чутливі до помилок ділянки коду – ініціалізацію схеми бази та формулу прогресії рівнів, бізнес-логіку нарахування досвіду й винагород, перевірку токена на кожному маршруті проміжного сервера, валідацію всіх полів оновлення статистики й множника ролі, атомарну видачу прав через транзакцію та роботу проміжних обробників авторизації обох вебкомпонентів. Поза автоматизованим покриттям свідомо залишено ділянки, перевірка яких потребує складних заглушок зовнішніх сервісів – безпосередні події Discord, асинхронні фонові задачі зняття покарань і нарахування голосового досвіду, повний потік OAuth2 та наскрізний рендеринг сторінки панелі за участю запущеного проміжного сервера. Ці сценарії натомість покрито ручними наскрізними тест-кейсами, тож разом обидва рівні перевірки – і окремих модулів, і взаємодії в цілому – доповнюють один одного.

Підсистему рівнів перевірено на відповідність закладеній прогресії складності. На рисунку 4.1 показано криву необхідного для підвищення рівня досвіду, побудовану за даними довідкової таблиці level_table. Графік підтверджує, що вимоги до досвіду зростають нелінійно, а темп зростання поступово сповільнюється на високих рівнях – це забезпечує відчутний прогрес для нових учасників і водночас зберігає цінність високих рівнів, що відповідає спроєктованій моделі економіки сервера.

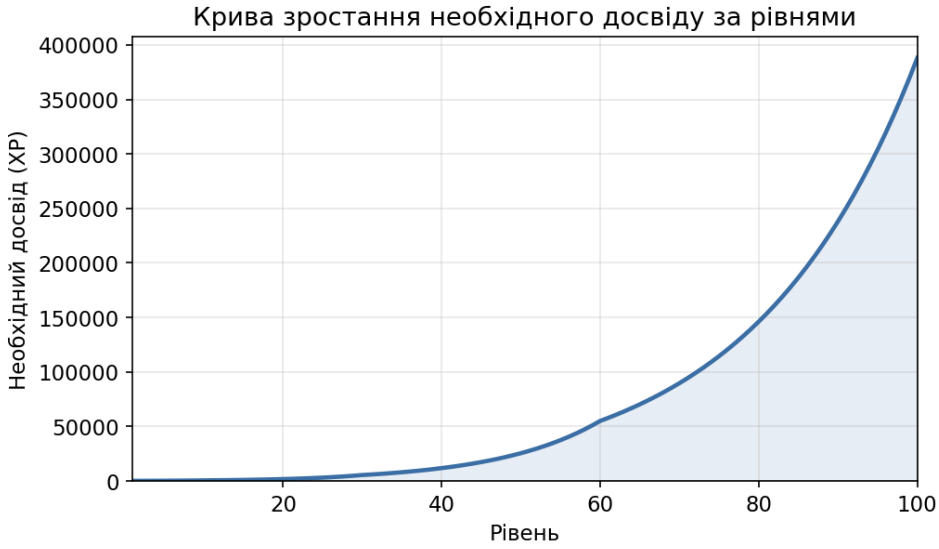

Рисунок 4.1 – Крива зростання необхідного досвіду за рівнями

Швидкодію проміжного сервера оцінено шляхом навантажувального тестування – вимірюванням медіанного часу відповіді API за різної кількості одночасних запитів окремо для операцій читання та запису. Результати наведено на рисунку 4.2. За навантаження до 50 одночасних запитів час відповіді не перевищує кількох десятків мілісекунд, а його зростання залишається близьким до лінійного аж до 200 запитів, що свідчить про достатній запас продуктивності для цільового масштабу застосунку. Очікувано, операції запису виконуються дещо повільніше за операції читання через блокування під час транзакцій.

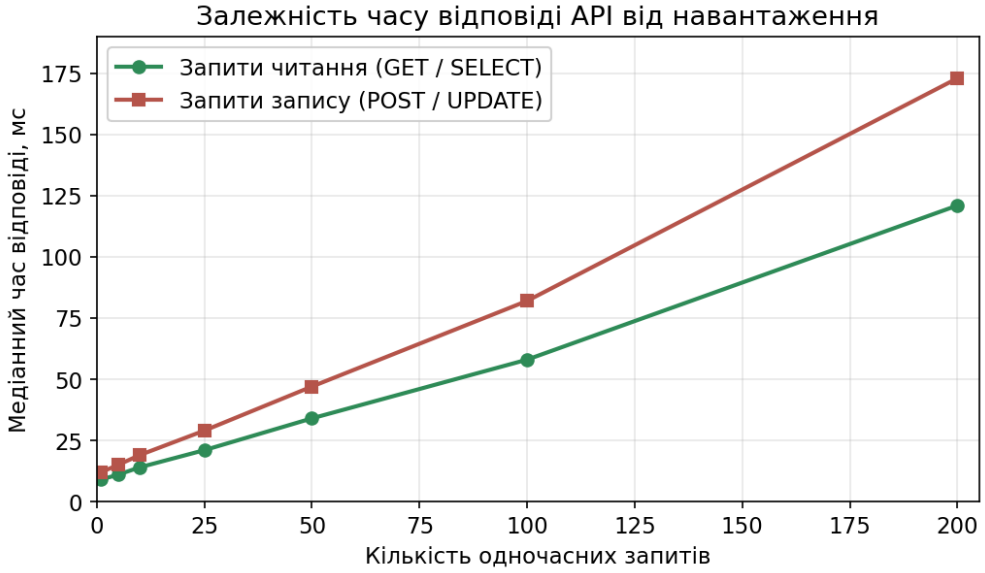

Рисунок 4.2 – Залежність часу відповіді API від навантаження

На підставі результатів автоматизованого й ручного тестування та виконаних вимірювань складено узагальнену оцінку якості застосунку за характеристиками моделі стандарту ISO/IEC 25010. Зведені показники наведено в таблиці 4.3.

Таблиця 4.3 – Оцінювання якості програмного забезпечення

Характеристика якості	**Оцінка**	**Обргунтування**
Функціональна придатність	Високий	Усі функції, визначені у другому розділі, реалізовано в повному обсязі та підтверджено 83 автоматизованими перевірками й набором ручних наскрізних тест-кейсів.
Рівень швидкодії	Достатній	Медіанний час відповіді API не перевищує кількох десятків мілісекунд за навантаження до 50 одночасних запитів, а зростання залишається близьким до лінійного аж до 200.
Сумісність	Високий	Три компоненти, написані різними мовами, узгоджено взаємодіють через спільну базу в режимі WAL та HTTP-API без конфліктів доступу.
Зручність використання	Достатній	Інтерфейс панелі згруповано за функціональними модулями, дані оновлюються асинхронно без перезавантаження сторінки, що спрощує виконання типових операцій.
Надійність	Достатній	Фонові задачі узгоджують стан бази з Discord, режим WAL усуває блокування при одночасному доступі, а виявлені дефекти усунено без зміни архітектури.
Захищеність	Достатній	Доступ до панелі обмежено авторизацією Discord OAuth2, кожен маршрут проміжного сервера захищено Bearer-токеном, форми – CSRF-токеном, а вивід екрановано від впровадження.
Супроводжуваність	Високий	Модульна Cog-архітектура та метрики, обчислені в лабораторних роботах, засвідчили низьку складність, просту ієрархію, високу згуртованість методів і слабку зв’язність класів.

Кінець таблиці 4.3

Переносимість	Достатній	Налаштування винесено в .env, використано відносні шляхи та стандартні середовища виконання, що дозволяє розгорнути застосунок на іншому сервері без зміни коду.
Функціональна придатність	Високий	Усі функції, визначені у другому розділі, реалізовано в повному обсязі та підтверджено 83 автоматизованими перевірками й набором ручних наскрізних тест-кейсів.
Рівень швидкодії	Достатній	Медіанний час відповіді API не перевищує кількох десятків мілісекунд за навантаження до 50 одночасних запитів, а зростання залишається близьким до лінійного аж до 200.
Сумісність	Високий	Три компоненти, написані різними мовами, узгоджено взаємодіють через спільну базу в режимі WAL та HTTP-API без конфліктів доступу.
Зручність використання	Достатній	Інтерфейс панелі згруповано за функціональними модулями, дані оновлюються асинхронно без перезавантаження сторінки, що спрощує виконання типових операцій.
Надійність	Достатній	Фонові задачі узгоджують стан бази з Discord, режим WAL усуває блокування при одночасному доступі, а виявлені дефекти усунено без зміни архітектури.
Захищеність	Достатній	Доступ до панелі обмежено авторизацією Discord OAuth2, кожен маршрут проміжного сервера захищено Bearer-токеном, форми – CSRF-токеном, а вивід екрановано від впровадження.

Отримані результати підтверджують, що розроблений застосунок відповідає вимогам, сформульованим у другому розділі. Функціональні можливості реалізовано в повному обсязі та підкріплено автоматизованими тестами, продуктивність достатня для типового навантаження серверів Discord, а розмежування доступу й спільний режим журналювання забезпечують безпеку та

узгодженість даних між компонентами. Виявлені під час тестування недоліки мали технічний характер і були усунені без зміни архітектури системи.

Висновки до розділу 4

У розділі описано процес програмної реалізації та тестування застосунку керування діскорд-ботом, виконаний відповідно до архітектурних рішень, обґрунтованих у попередньому розділі.

Реалізовано модель даних на основі реляційної бази SQLite із центральною таблицею учасників та складеним первинним ключем, що гарантує унікальність пари “користувач–сервер”. Ядро бота побудовано мовою Python із використанням бібліотеки disnake та модульної Cog-архітектури з динамічним завантаженням функціональних блоків – модерації, рівнів і економіки, управління ролями, верифікації та створення приватних голосових кімнат. Доступ до сховища інкапсульовано у класі-обгортці з режимом журналювання WAL, що дозволяє боту та проміжному серверу одночасно читати й записувати спільний файл бази без взаємних блокувань.

Створено проміжний сервер взаємодії на платформі Node.js мовою TypeScript із застосуванням каркаса Express, який надає вебпорталу захищений Bearer-токеном HTTP-інтерфейс до спільної бази даних без прямого доступу до файлу сховища. Вебпортал адміністрування реалізовано на фреймворку Laravel із серверним рендерингом шаблонів Blade та авторизацією через Discord OAuth2 засобами пакета Socialite. У такий спосіб забезпечено повноцінну двосторонню взаємодію між ботом і вебінтерфейсом через спільне сховище – зміни, внесені адміністратором на порталі, одразу впливають на поведінку бота, а результати дій бота відображаються у вебпанелі.

Проведено тестування застосунку із застосуванням трьох взаємодоповнюваних підходів – функціонального, інтеграційного та автоматизованого модульного. Підготовлено сім файлів тестів із вісімдесятьма трьома перевірками для трьох компонентів системи засобами pytest, Jest у поєднанні з Supertest та Pest, доповнених ручними наскрізними тест-кейсами. Усі

заплановані перевірки пройдено успішно, а виявлені дефекти мали суто технічну природу й були усунені без зміни архітектури системи.

Окремо досліджено нефункціональні характеристики застосунку. Перевірка підсистеми рівнів підтвердила, що крива необхідного для підвищення рівня досвіду зростає нелінійно, а темп її зростання поступово сповільнюється на високих рівнях, що забезпечує відчутний прогрес для нових учасників і водночас зберігає цінність високих рівнів відповідно до спроєктованої моделі економіки сервера. Навантажувальне тестування проміжного сервера засвідчило, що медіанний час відповіді HTTP-API не перевищує кількох десятків мілісекунд за навантаження до п’ятдесяти одночасних запитів і зростає близько до лінійного аж до двохсот запитів, причому операції запису очікувано виконуються дещо повільніше за операції читання через блокування під час транзакцій. Отримані виміри свідчать про достатній запас продуктивності системи для типового навантаження серверів Discord.

За результатами автоматизованого й ручного тестування та навантажувальних вимірювань складено узагальнену оцінку якості програмного забезпечення за характеристиками моделі стандарту ISO/IEC 25010, яка засвідчила високий рівень функціональної придатності, сумісності й супроводжуваності та достатній рівень швидкодії, надійності, захищеності й переносимості. Таким чином, отримані результати підтверджують, що розроблений застосунок повністю відповідає вимогам, сформульованим у другому розділі, а його функціональні можливості реалізовано в повному обсязі та підкріплено результатами тестування

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розроблено застосунок керування діскорд-ботом, призначений для спрощення та автоматизації адміністрування серверів Discord. Поставлену мету досягнуто, а всі визначені у вступі завдання виконано в повному обсязі.

Проведено дослідження предметної області та аналіз застосунків-аналогів – Nekotina, Rippy, Appy, Groot та Arcane. Аналіз засвідчив, що жоден із наявних на ринку ботів не поєднує в собі повноцінну модерацію, систему залучення аудиторії та зручний вебінтерфейс управління в межах єдиного рішення. Це обґрунтувало актуальність розробки комплексного застосунку, що усуває виявлені недоліки аналогів.

Сформовано специфікацію вимог до програмного забезпечення, що охоплює шість ключових функцій системи – фільтрацію слів, модерацію, економіку та систему рівнів, управління ролями, верифікацію нових учасників і створення приватних голосових кімнат. На основі специфікації спроєктовано архітектуру системи з шаровою та водночас розподіленою структурою, побудовано діаграми використання, класів, компонентів і розгортання, а також розроблено інтерфейс користувача вебпорталу.

Реалізовано інструменти для налаштування та модерації сервера адміністратором – автоматичну фільтрацію контенту, гнучку систему розмежування прав за ролями, накладення санкцій, верифікацію нових учасників та керування кастомними ролями й приватними голосовими кімнатами. Розроблено зручний вебінтерфейс керування функціоналом бота з використанням фреймворку Laravel та авторизацією через Discord OAuth2, а взаємодію між ботом і вебінтерфейсом забезпечено через проміжний сервер на Node.js і спільне сховище даних на SQLite.

Виконано тестування розробленого застосунку із застосуванням функціонального, інтеграційного та автоматизованого модульного підходів – усі вісімдесят три автоматизовані перевірки та ручні наскрізні тест-кейси пройдено

успішно. Оцінювання якості за моделлю стандарту ISO/IEC 25010 підтвердило відповідність системи висунутим у завданні вимогам – час реакції бота на команду не перевищує визначеного у специфікації порогу у дві секунди, а медіанний час відповіді API залишається в межах кількох десятків мілісекунд за навантаження до п’ятдесяти одночасних запитів, що свідчить про достатній запас продуктивності для цільового масштабу застосунку.

Практичне значення отриманих результатів полягає в тому, що розроблений застосунок надає адміністраторам серверів Discord цілісний інструмент автоматизації управління спільнотою без залежності від платних сторонніх сервісів. На відміну від проаналізованих аналогів, кожен з яких орієнтований переважно на окрему нішу, створене рішення поєднує розвинену модерацію, систему рівнів та внутрішньої економіки і повнофункціональний вебінтерфейс у межах єдиного, модульного й масштабованого продукту, що становить основну новизну роботи. Застосування модульної Cog-архітектури, спільного сховища з режимом журналювання WAL і розмежування доступу через єдину точку перевірки автентифікації забезпечує гнучке розширення функціоналу, узгодженість стану системи та безпеку взаємодії компонентів.

Розроблений застосунок рекомендовано до впровадження власниками та адміністраторами серверів Discord, які потребують гнучких і комплексних засобів керування спільнотою. Подальший розвиток об’єкта розробки доцільно спрямувати на розширення функціоналу новими модулями – зокрема музичним відтворенням, інтеграцією із зовнішніми сервісами та підсистемою аналітики активності, – а також на перехід до клієнт-серверної СКБД для обслуговування великої кількості серверів і впровадження механізму pub/sub для миттєвого набуття чинності змінами конфігурації без перезапуску компонентів системи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. discord.py. Python library for the Discord API. URL: https://discordpy.readthedocs.io/en/stable/ (дата звернення: 28.04.2026).

2. disnake. Active maintained fork of discord.py with slash-command support. URL: https://docs.disnake.dev/ (Accessed: 28.04.2026).

3. discord.js. JavaScript library for interacting with the Discord API. URL: https://discord.js.org/ (Accessed: 28.04.2026).

4. JDA – Java Discord API. URL: https://jda.wiki/ (Accessed: 28.04.2026).

5. Discord.Net. A .NET library for interacting with the Discord API. URL: https://discordnet.dev/ (Accessed: 28.04.2026).

6. Laravel. The PHP Framework for Web Artisans. URL: https://laravel.com/docs/ (Accessed: 28.04.2026).

7. Next.js. The React Framework for the Web. URL: https://nextjs.org/docs/ (дата звернення: 28.04.2026).

8. FastAPI. High performance Python web framework. URL: https://fastapi.tiangolo.com/ (Accessed: 28.04.2026).

9. NestJS. A progressive Node.js framework for building efficient and scalable server-side applications. URL: https://docs.nestjs.com/ (дата звернення: 28.04.2026).

10. Nekotina – Your adorable neko companion for Discord. URL: https://nekotina.com/en (Accessed: 28.04.2026).

11. Rippy | Your Next Smart Move. URL: https://rippylabs.it/ (Accessed: 28.04.2026).

12. Appy – The best free to use Discord Bot. URL: https://appy.bot/ (Accessed: 28.04.2026).

13. Groot – The Best Discord Music Bot. URL: https://grootbot.pro/ (Accessed: 28.04.2026).

14. Arcane: The Best Discord Bot. URL: https://arcane.bot/ (Accessed: 28.04.2026).

15. Alier M., Casañ Guerrero M. J., Amo D., Severance C., Fonseca D. Privacy and E-Learning: A Pending Task. Sustainability. Vol. 13, No. 16. URL: https://www.mdpi.com/2071-1050/13/16/9206 (Accessed: 28.04.2026).

16. Mirza M. A., Ajay G. P., Hareesh K., Brahmaiah B., Rohith J. TenantX: Enterprise-Grade Multi-Tenant SaaS Platform with Dynamic RBAC and Configurable Workflow Engine. International Journal of Engineering Research & Technology (IJERT). Vol. 15, Iss. 04. URL: https://doi.org/10.5281/zenodo.19788686 (Accessed: 28.04.2026).

17. Putra F. P. E., Efendi R. W., Tamam A. B., Pramadi W. A. Trends and Best Practices in API-Based Web Development Using Laravel and React. Brilliance: Research of Artificial Intelligence. 2025. Vol. 5, No. 1. P. 223–233. DOI: 10.47709/brilliance.v5i1.5971. URL: https://doi.org/10.47709/brilliance.v5i1.5971 (Accessed: 28.04.2026).

18. Python Software Foundation. asyncio – Asynchronous I/O. Python 3 Documentation. URL: https://docs.python.org/3/library/asyncio.html (Accessed: 28.04.2026).

19. disnake. Extensions – Cogs. disnake Documentation. URL: https://docs.disnake.dev/en/stable/ext/commands/cogs.html (Accessed: 28.04.2026).

20. Sandhu R. S., Coyne E. J., Feinstein H. L., Youman C. E. Role-Based Access Control Models. IEEE Computer. 1996. Vol. 29, No. 2. P. 38–47. DOI: 10.1109/2.485845. URL: https://ieeexplore.ieee.org/document/485845 (Accessed: 28.04.2026).

21. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force (IETF). 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749 (Accessed: 28.04.2026).

22. lavalink-devs. Lavalink: Standalone audio sending node based on Lavaplayer. GitHub. URL: https://lavalink.dev/ (Accessed: 28.04.2026).

23. Redis Ltd. Pub/Sub. Redis Documentation. URL: https://redis.io/docs/latest/develop/interact/pubsub/ (Accessed: 28.04.2026).

24. Guan B. Designing Scalable Rate Limiting Systems: Algorithms, Architecture, and Distributed Solutions. arXiv:2602.11741 [cs.DC]. 2026. URL: https://arxiv.org/abs/2602.11741 (Accessed: 28.04.2026).

25. Almeida T. A., Gómez Hidalgo J. M., Yamakami A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 11th ACM Symposium on Document Engineering (DocEng ’11). 2011. P. 259–262. DOI: 10.1145/2034691.2034742. URL: https://dl.acm.org/doi/10.1145/2034691.2034742 (Accessed: 28.04.2026).

26. Wilson G. Twelve quick tips for software design. PLOS Computational Biology. 2022. Vol. 18, No. 2. Article e1009809. DOI: 10.1371/journal.pcbi.1009809. URL: https://doi.org/10.1371/journal.pcbi.1009809 (Accessed: 28.04.2026).

27. Dragoni N., Lanese I., Larsen S. T., Mazzara M., Mustafin R., Safina L. Microservices: How To Make Your Application Scale. Perspectives of System Informatics. Lecture Notes in Computer Science, Vol. 10742. P. 95–104. Springer, 2018. DOI: 10.1007/978-3-319-74313-4_8. URL: https://doi.org/10.1007/978-3-319-74313-4_8 (Accessed: 28.04.2026).

28. Fowler M., Lewis J. Microservices. martinfowler.com. 2014. URL: https://martinfowler.com/articles/microservices.html (Accessed: 28.04.2026).

29. Redis Ltd. Redis Streams Introduction. Redis Documentation. URL: https://redis.io/docs/latest/develop/data-types/streams/ (Accessed: 28.04.2026).

30. Yujian L., Bo L. A Normalized Levenshtein Distance Metric. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2007. Vol. 29, No. 6. P. 1091–1095. DOI: 10.1109/TPAMI.2007.1078. URL: https://ieeexplore.ieee.org/document/4160958 (Accessed: 28.04.2026).

31. Cheng J., Danescu-Niculescu-Mizil C., Leskovec J. Antisocial Behavior in Online Discussion Communities. Proceedings of the 9th International AAAI Conference on Web and Social Media (ICWSM). 2015. P. 61–70. URL: https://arxiv.org/abs/1504.00680 (Accessed: 28.04.2026).

32. Hamari J., Koivisto J., Sarsa H. Does Gamification Work? – A Literature Review of Empirical Studies on Gamification. Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS). 2014. P. 3025–3034. DOI: 10.1109/HICSS.2014.377. URL: https://ieeexplore.ieee.org/document/6758978 (Accessed: 28.04.2026).

33. Redis Ltd. SET command. Redis Documentation. URL: https://redis.io/docs/latest/commands/set/ (Accessed: 28.04.2026).