Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: Розроблення Discord-бота для інформаційної системи користувача на платформі Faceit

Виконав: студент II курсу групи **КНС - 21**
спеціальності
122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Гладун В.О
(прізвище та ініціали)

Керівник: Лукащук Б. С.
(прізвище та ініціали)

Керівник: Думанський О.І.
(прізвище та ініціали)

Рецензент Сторонук О.І.
(прізвище та ініціали)

Львів – 2025 року

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
_____ Борецька І. Б.
"10" червня 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Гладуну Володимиру Олеговичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення Discord-бота для інформаційної системи користувача на платформі Faceit

керівник роботи *Лукащук Богдан Сергійович, асистент*

керівник роботи Думанський Остап Іванович, к.ф.-м.н, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від 15 11 2024 року № С-582

2. Термін подання студентом роботи 10 червня 2025 року

3. Вихідні дані до роботи Розроблення Discord-бота для інформаційної системи користувача на платформі Faceit

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Стан проблемної області, інформаційне та математичне забезпечення, програмне та технічне забезпечення, висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Підготовка матеріалу до відповіді.

6. Дата видачі завдання 18.11.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз аналогів і джерел щодо ботів та API статистики	19.11.24 – 29.02.25	Виконано
2	Аналіз предметної області та постановка задачі	29.02.25 – 11.03.25	Виконано
3	Проєктування архітектури Discord-бота та API-інтеграцій	11.03.25 – 22.03.25	Виконано
4	Реалізація функцій отримання статистики через FACEIT API	22.03.25 – 31.04.25	Виконано
5	Розробка системи обробки Webhook та візуалізації статистики	31.04.25 – 24.05.25	Виконано
6	Оформлення пояснювальної записки та підготовка до захисту	24.05.25 – 09.06.25	Виконано

Студент _____ **Гладун В.О.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Лукащук Б. С.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Думанський О.І.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота містить 62 сторінок, 1 таблицю, 29 ілюстрацій, 5 додатків, 12 джерел.

У роботі описано етапи розробки та впровадження інтегрованої системи для отримання та візуалізації ігрової статистики з платформи FACEIT у середовище Discord. Система складається з веб-сервісу на базі ASP.NET Core, консольного Discord-бота створеного за допомогою .NET Framework, та бази даних на основі PostgreSQL, що забезпечують ефективну взаємодію з користувачами, збір та обробку статистичних даних, а також генерацію графічних звітів.

Ключові слова: ASP.NET Core, Discord-бот, мікросервісна архітектура, webhook, PuppeteerSharp, генерація інфографіки, PostgreSQL, REST API, інтеграція FACEIT, C#.

ABSTRACT

The bachelor's thesis consists of 62 pages, 1 table, 29 illustrations, 5 appendices, and 12 references.

The thesis describes the stages of development and implementation of an integrated system for retrieving and visualizing gaming statistics from the FACEIT platform into the Discord environment. The system consists of a web service based on ASP.NET Core, a console Discord bot developed using the .NET Framework, and a PostgreSQL-based database. Together, they ensure efficient user interaction, data collection and processing, and graphical report generation.

Keywords: ASP.NET Core, Discord bot, microservice architecture, webhook, PuppeteerSharp, infographic generation, PostgreSQL, REST API, FACEIT integration, C#.

ТЕХНІЧНЕ ЗАВДАННЯ

Завдання дипломного проєкту полягає в розробці програмного забезпечення у вигляді Discord-бота під назвою FaceitHelper, призначеного для взаємодії з відкритим API платформи FACEIT. Даний бот повинен надавати користувачам можливість отримувати інформацію про статистику гравців, результати останніх матчів, деталі окремих ігор, а також виводити зведену статистику у вигляді зручної інфографіки.

Розробка програмного продукту передбачає реалізацію наступних функціональних складових: реалізувати підключення до FACEIT API, для отримання детальної статистики про проведені ігри гравця, інформації про гравця, включаючи рівень, ELO, розробити систему обробки подій за допомогою технології Webhook, яка дозволяє в реальному часі отримувати інформацію про створення матчів, зміну їхнього статусу, завершення та доступність демо-файлів. Реалізувати Базу даних для збереження статистики даних гравця. реалізувати автоматичне оброблення, логування та надсилання у Discord.

Для зручності користувачів реалізовано набір Slash-команд у Discord, серед яких: /faceitstats (базова статистика гравця), /matchstats (останні матчі), /matchinfo (дані по ID матчу), /playerstats (агреговані показники), /avg (середнє число вбивств), /statimage (генерація зображення з інфографікою).

Реалізувати візуалізації статистичних даних за допомогою HTML-шаблону та CSS-оформлення та бібліотеки PuppeteerSharp.

Програму реалізувати за допомогою мови програмування C# з використанням бібліотек DSharpPlus та Newtonsoft.Json Реалізувати можливість зручної конфігурації програми через JSON.

Очікуваним результатом дипломної роботи є повнофункціональний Discord-бот, який поєднує текстовий і графічний вивід статистики з FACEIT, підтримує обробку вебхуків, працює у реальному часі та може бути використаний окремими гравцями для аналізу своєї ігрової ефективності.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface – інтерфейс прикладного програмування.

БД – база даних.

CS2 – Counter-Strike 2 – відеогра, для якої збирається статистика.

Discord API – набір методів для взаємодії із серверами Discord.

JSON – JavaScript Object Notation – текстовий формат для представлення структурованих даних.

JWT – JSON Web Token – спосіб безпечної авторизації та обміну інформацією.

PuppeteerSharp – C#-бібліотека для керування headless-версією Chromium.

HTML – HyperText Markup Language – мова розмітки гіпертексту.

CSS – Cascading Style Sheets – таблиці стилів для оформлення вебсторінок.

HTTP – HyperText Transfer Protocol – протокол передавання гіпертекстових документів.

UML – Unified Modeling Language – уніфікована мова моделювання.

Webhook – механізм, який дозволяє отримувати повідомлення про події через HTTP.

ELO – умовна оцінка навичок гравця в ігрових системах рейтингу.

K/D – Kill/Death Ratio – співвідношення кількості вбивств до смертей у грі. HS% – Headshot Percentage – відсоток пострілів у голову від загальної кількості вбивств.

ВСТУП

У сучасному цифровому світі онлайн-ігри та платформи для змагань, такі як FACEIT, набули широкого поширення серед геймерської спільноти. Кіберспорт стає не лише формою розваги, а й професійною діяльністю, що потребує глибокої аналітики, моніторингу показників гравців та оперативного прийняття рішень на основі статистики. У цьому контексті зростає потреба в інструментах, які дозволяють швидко отримувати важливу інформацію про гравців та матчі в зручному, інтегрованому середовищі.

Актуальність теми зумовлена необхідністю створення ефективного засобу для збирання, аналізу та візуального представлення статистичних даних з платформи FACEIT. Замість постійного переходу між сервісами, користувачі можуть отримувати всю потрібну інформацію безпосередньо у Discord – платформі, яка є основним середовищем спілкування для геймерів. Візуалізація статистики у вигляді інфографіки робить аналітику доступнішою, зрозумілішою та зручнішою для сприйняття, що значно підвищує її практичну цінність.

Об'єктом дослідження є процес інтеграції API-даних із зовнішньої ігрової платформи у середовище Discord.

Предметом дослідження є засоби і методи реалізації Discord-бота для автоматизованого отримання, обробки та візуалізації статистичних даних із платформи FACEIT.

Метою роботи є розробка програмного засобу у вигляді Discord-бота, який дозволяє користувачам отримувати персональну статистику, результати матчів та інші дані з FACEIT API у форматі текстових повідомлень і графічних зображень.

Для досягнення мети було поставлено наступні завдання:

- проаналізувати доступні засоби інтеграції з Discord API та FACEIT API; реалізувати обробку подій FACEIT через Webhook;
- реалізувати набір Slash-команд для отримання статистики гравця;
- реалізувати модуль візуалізації статистичних даних у вигляді зображення; забезпечити конфігураційність проєкту через JSON-файл;

- провести тестування та верифікацію функціоналу бота.

Практична цінність роботи полягає у створенні гнучкого інструменту, який може бути використаний гравцями та командами для ефективного моніторингу ігрової активності. Розроблений бот спрощує доступ до персональної статистики, дозволяє реагувати на матчеві події у режимі реального часу, підвищує рівень командної комунікації, а також має потенціал до розширення функціоналу. Враховуючи активне використання Discord у геймерському середовищі, реалізований проєкт має прикладну цінність і може знайти практичне застосування у кіберспорті.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Сучасний стан ринку ігрової аналітики та Discord-інтеграцій.

У сучасному цифровому середовищі ринок і відеоігор та інтегрованих комунікаційних платформ, зокрема Discord, переживає динамічний розвиток. За прогнозами дослідницької компанії MarketsandMarkets, обсяг ринку ігрової аналітики сягне 5,3 мільярда доларів США до 2026 року, зростаючи із середньорічним темпом понад 19,5% CAGR. Такий розвиток обумовлений не лише зростанням кількості геймерів, а й підвищеним попитом на персоналізовану статистику, метрики продуктивності, а також змагальний дух в eSports-середовищі.

Рисунок 1.1 – Розмір доходу на ринку відеоігор

Особливе місце в цій екосистемі займає платформа Discord, яка за останні роки трансформувалася з простої VoIP-програми в повноцінне середовище для управління спільнотами, спілкування, автоматизації й навіть бізнес-рішень [1]. Станом на 2024 рік, Discord налічує понад 200 мільйонів активних користувачів щомісяця, з яких

значна частина припадає на геймерську аудиторію.

Боти для Discord стали ефективними інструментами автоматизації. Вони дозволяють користувачам взаємодіяти з API сторонніх сервісів, отримувати інформацію, генерувати графіку та навіть будувати повноцінні аналітичні панелі без необхідності залишати середовище Discord [1].

Однією з найпопулярніших дисциплін, що потребує такої аналітики, є Counter-Strike 2 (CS2). Турнірна платформа FACEIT, яка забезпечує інфраструктуру для змагань і рейтингових матчів, надає API-доступ до даних гравців [3], що відкриває широкі можливості для інтеграції та візуалізації статистики. Попри це, офіційний інтерфейс FACEIT часто не задовольняє вимог щодо зручності, гнучкості чи інформативності.

На цьому фоні зростає попит на кастомні рішення — такі як FaceitHelperBot — які дозволяють інтегрувати функції статистики, аналітики та графічної візуалізації у Discord-сервери. Завдяки використанню бібліотек типу SkiaSharp, PuppeteerSharp та API Discord, такі боти можуть автоматично обробляти статистику матчів, генерувати звіти й надавати гравцям індивідуальну аналітику в інтерактивному та зручному форматі.

До основних чинників зростання цієї ніші можна віднести:

- високий рівень залучення гравців у змагальний процес;
- потребу у самопокращенні та командному аналізі;
- розвиток eSports-індустрії;
- розширення інфраструктури Discord для кастомних ботів;
- попит на інтерактивну й адаптивну візуалізацію статистики.

Таким чином, ринок аналітики в сфері відеоігор, особливо в контексті Discord-інтеграцій, демонструє високий потенціал розвитку. Рішення, що поєднують зручність, графіку та актуальні дані — як-от розроблена система FaceitHelper — мають вагоме прикладне значення як для аматорів, так і для професійних гравців.

1.2. Використання Discord-ботів для розвитку геймерських спільнот.

У сучасному цифровому середовищі Discord став ключовою платформою для спілкування, координації та обміну інформацією серед геймерів. Одним із найцінніших напрямів використання Discord-ботів є аналітика ігрової продуктивності — функціональність, яка дозволяє гравцям відстежувати свій прогрес, отримувати рекомендації для поліпшення та зміцнювати командну взаємодію.

Discord-боти, що використовують API популярних ігрових платформ — таких як FACEIT [3], Steam, Valorant API — здатні витягувати дані про гравця в реальному часі. Вони надають користувачам доступ до:

- історії матчів;
- особистої статистики (вбивства, смертей, асистів, HS%);
- рейтингів (ELO, Skill Level);
- графіків прогресу;
- кращих виступів тощо.

Особливістю ботів, таких як FaceitHelper, є інтерактивність та візуальна привабливість. Завдяки використанню HTML+CSS-шаблонів у поєднанні з бібліотеками наприклад PuppeteerSharp [12] чи SkiaSharp, користувач може отримати повноцінне зображення статистики, яке автоматично рендериться на запит. Це дозволяє швидко ділитися результатами без необхідності вручну заходити на сторонні сайти чи скріншотити власні профілі.

Основні переваги використання Discord-ботів зі статистикою:

- Автоматизація: усі дані оновлюються у фоновому режимі на запит без втручання користувача.
- Мотивація: наочні графіки та аналітика стимулюють до покращення результатів.
- Тренування команд: тренери або лідери спільнот можуть аналізувати гру всіх учасників.
- Зручність: бот працює прямо в Discord, не потребуючи сторонніх застосунків.

- Реальна економія часу: більше ніяких пошуків даних вручну через вебінтерфейс FACEIT.

Геймерські спільноти**,** особливо ті, що активно беруть участь у змаганнях або створюють власні турніри, отримують завдяки таким інструментам реальні конкурентні переваги. Можливість бачити не лише результат, а й динаміку розвитку гравця, дозволяє формувати кращі склади, приймати тактичні рішення й розвивати командну синергію.

Окрім цього, боти з функцією зведених звітів по всіх матчах, ТОП виступів чи графіків прогресу надають повну картину ігрового потенціалу користувача. Це корисно як для самих гравців, так і для тренерів, організаторів турнірів або навіть стрімерів, що ведуть власну аналітику.

Таким чином, використання Discord-ботів для ігрової аналітики — це не просто зручність, а необхідність у професійному середовищі, де кожна деталь може вплинути на результат. Саме тому рішення на зразок FaceitHelper стають базовим інструментом для багатьох команд і спільнот, які прагнуть рости та досягати нових результатів.

1.3. Аналіз конкурентних рішень

З розвитком індустрії кіберспорту та ростом популярності платформ, таких як FACEIT, зросла й потреба у високоякісних аналітичних інструментах, які дозволяють гравцям моніторити свою ігрову статистику, аналізувати прогрес та покращувати ігрові показники. Окреме місце у цьому ланцюжку займають Discord-боти, оскільки Discord став де-факто стандартом для комунікації у геймерських спільнотах.

Сьогодні ринок рішень у цій сфері поділяється на два основні напрямки:

- Глобальні комерційні сервіси, що надають аналітику через API або вебінтерфейс.
- Спеціалізовані Discord-боти, які реалізують інтеграцію з платформами типу FACEIT і виводять статистику безпосередньо в чат.

Популярні конкуренти та їх аналіз.

Tracker Network (tracker.gg)

Tracker Network є одним із найбільших сервісів, що агрегує статистику з безлічі ігор: CS2, Valorant, Apex Legends, Fortnite та інших. Сервіс має власних Discord-ботів, які надають короткі текстові звіти на запит користувача.

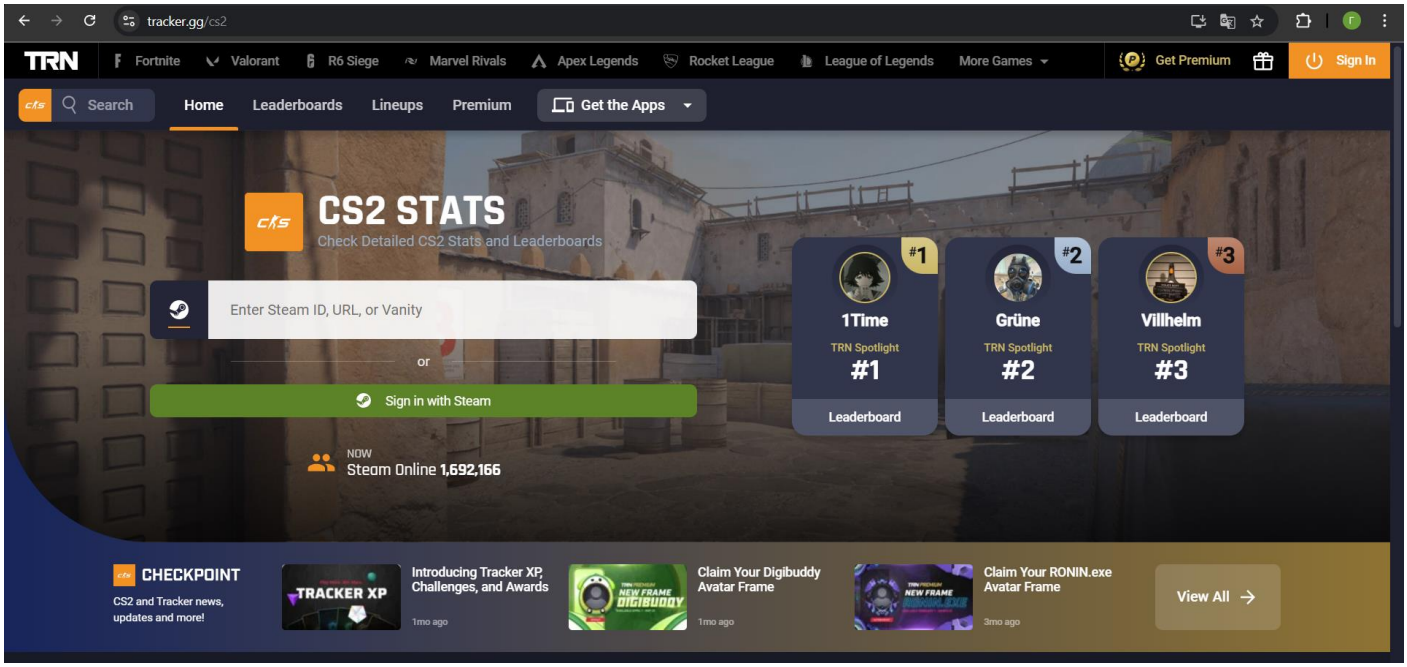


Рисунок 1.2 – Інтерфейс сервісу Tracker Network з детальною статистикою CS2

Переваги:

- Висока стабільність роботи.
- Актуальна інформація з офіційних джерел.
- Підтримка декількох платформ одночасно.

Недоліки:

- Вивід інформації обмежений стандартними шаблонами.
- Відсутність візуалізації (графіки, діаграми).
- Неможливість кастомізації під окрему команду чи сервер.

FaceitStatsBot (GitHub open-source)

Це приклад відкритих рішень, що використовують FACEIT API для виводу основних статистик через Discord-бота. Як правило, реалізація мінімальна: виводиться ELO, рівень, кількість матчів, K/D тощо.

Переваги:

- Простота у розгортанні.
- Повна кастомізація коду.

- Не потребує сторонніх сервісів.

Недоліки:

- Відсутність підтримки.
- Примітивний вигляд повідомлень.
- Немає графіків, аналітики або зображень.

Отже, хоча існуючі рішення для збору ігрової статистики та інтеграції з Discord можуть бути корисними для базового моніторингу показників гравців, без потужних інструментів візуалізації, розширених звітів та гнучкої інтеграції вони не завжди задовольняють потреби професійних користувачів, команд чи тренерів. FaceitHelper, завдяки своїй комплексній архітектурі, можливостям масштабування та підтримці інноваційних функцій, виступає ефективним та перспективним інструментом, що здатен значно підвищити якість аналізу ігрової телеметрії та покращити процес прийняття рішень у кіберспортивній діяльності. Без належної підтримки таких спеціалізованих платформ, багато гравців і команд ризикують втратити важливі можливості для самовдосконалення та стратегічного розвитку.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Проектування бази даних

У системах аналітики та обробки ігрової статистики бази даних є критично важливою складовою, що забезпечує надійне та ефективне зберігання, доступ і управління великими обсягами інформації. У контексті проєкту FaceitHelper, що інтегрує ігрові дані гравців і матчів, потрібна структурована база даних для збереження інформації про гравців, ігрові сесії та детальну статистику.

Бази даних (БД) представляють собою організовані колекції даних, які дозволяють швидко здійснювати операції читання, запису та оновлення інформації. Вони підтримують збереження складних взаємозв'язків між сутностями, такими як гравці, матчі, команди, а також аналітичних метрик і результатів.

Основна мета застосування бази даних у подібних системах полягає в оптимізації продуктивності — швидкому і точному доступі до актуальних даних, а також у забезпеченні цілісності і узгодженості інформації, що дозволяє створювати достовірні звіти та графіки [2].

При виборі типу бази даних для проєкту необхідно враховувати особливості даних і характер запитів. Для структурованої, сильно зв'язаної інформації реляційні бази даних (РБД) є оптимальним вибором. Проте, у випадках, коли необхідно працювати з напівструктурованими або неструктурованими даними, варто розглядати NoSQL-системи.

Ключовими критеріями вибору є продуктивність, масштабованість, надійність і простота інтеграції. Реляційні бази даних, такі як PostgreSQL, демонструють високу стабільність та підтримку складних запитів, що важливо для аналітики ігрових даних у реальному часі. Крім того, PostgreSQL має відкритий код, що знижує витрати на ліцензування [11].

Серед переваг реляційних БД слід виділити:

- Цілісність і узгодженість даних: Встановлення зовнішніх та первинних ключів, унікальних обмежень і правил дозволяє підтримувати коректність і достовірність даних.

- Підтримка транзакцій: Реляційні бази гарантують, що зміни даних у межах транзакції будуть виконані повністю або не будуть застосовані зовсім, що виключає часткові оновлення, які можуть призвести до помилок [6]. Транзакції мають властивості ACID (атомарність, узгодженість, ізольованість, довговічність).
- Можливість складних запитів: Використання мови SQL дозволяє здійснювати гнучкі та ефективні операції вибірки, фільтрації та агрегації даних, необхідні для формування детальних статистичних звітів.
- Масштабованість: Реляційні бази добре масштабуються вертикально, забезпечуючи високу продуктивність на апаратних ресурсах, а PostgreSQL також має розширені можливості для горизонтального масштабування.

Таблиця 2.1 – Приклад структури даних

id	Ім'я	Прізвище	Електронна пошта	Вік
1	Олексій	Іванов	oleksii.ivanov@email.com	25
2	Марія	Петрівна	maria.petrivna@email.com	30
3	Дмитро	Коваленко	dmytro.kovalenko@email.com	22

Якщо дані системи стають більш гнучкими або включають документовані чи графові структури, тоді доцільно використовувати NoSQL бази даних, які не вимагають фіксованої схеми і забезпечують швидкий доступ у масштабованих розподілених середовищах.

Однак у випадку аналітичної системи FaceitHelper використання PostgreSQL забезпечує збалансованість між продуктивністю, надійністю, узгодженістю та зручністю розробки, особливо завдяки підтримці Entity Framework Core, що спрощує моделювання даних і роботу з ними.

Таким чином, обрана архітектура бази даних створює міцний фундамент для подальшого розвитку функціоналу системи, підтримує гнучкість та можливість розширення.

Після того як було обрано тип бази даних, було розпочато роботу над початковою структурою, для початку розпочнемо з полів таблиці Players(рис. 2.1):

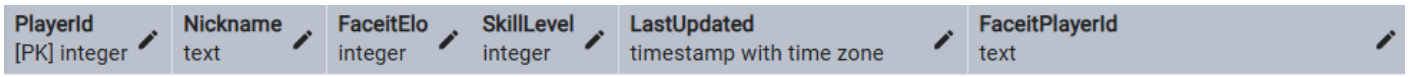


Рисунок 2.1 – Структура таблиці Players

Таблиця **Players** зберігає інформацію про гравців, які відслідковуються системою. Ключовими полями є:

PlayerId (integer, первинний ключ) Унікальний ідентифікатор гравця у базі даних. Використовується для однозначної ідентифікації запису про конкретного гравця.

Nickname (text) Нікнейм або ім'я гравця, яке він використовує у грі або на платформі. Це текстове поле, яке зберігає видимий для інших користувачів псевдонім.

FaceitElo (integer) Рейтинг ELO гравця на платформі Faceit. Це числове значення, яке відображає рівень навичок або майстерності гравця у грі.

SkillLevel (integer) Рівень майстерності гравця, який може бути додатковою метрикою, визначеною Faceit, що деталізує рівень навичок у порівнянні з іншими користувачами.

LastUpdated (timestamp with time zone) Дата і час останнього оновлення інформації про гравця. Містить часову зону, що дозволяє зберігати точний час оновлення.

FaceitPlayerId (text) Унікальний ідентифікатор гравця на платформі Faceit, який використовується для інтеграції з API та зіставлення даних між системами.

Далі опишу структуру таблиць **Matches** (рис. 2.2), **PlayerStats** (рис. 2.3):

Рисунок 2.2 – Структура таблиці Matches

Таблиця **Matches** зберігає основну інформацію про ігрові матчі, які відбуваються на платформі. Вона містить дані, необхідні для ідентифікації кожного матчу, відстеження часу його проведення, а також результату.

MatchId — унікальний ідентифікатор матчу, який дозволяє однозначно ідентифікувати кожен запис у таблиці.

Map — назва ігрової карти, на якій відбувся матч, що дає змогу аналізувати продуктивність на різних локаціях.

StartTime — дата і час початку матчу з урахуванням часової зони, необхідні для хронологічного відстеження подій.

EndTime — дата і час завершення матчу, що дозволяє визначити тривалість ігрової сесії.

Result — текстове поле, яке містить результат матчу, наприклад, перемогу або поразку гравця чи команди.

Рисунок 2.3 – Структура таблиці PlayerStats

Таблиця **PlayerStats** зберігає детальну статистику гравців за кожен матч. Вона пов’язує конкретного гравця з конкретним матчем і фіксує ключові ігрові показники.

StatId — унікальний ідентифікатор запису (первинний ключ), що забезпечує однозначну ідентифікацію кожної статистики.

PlayerId — ідентифікатор гравця, що вказує, до якого гравця належать зафіксовані дані.

MatchId — ідентифікатор матчу, з яким пов’язана статистика гравця.

Kills — кількість вбивств, здійснених гравцем у матчі.

Deaths — кількість смертей гравця у матчі.

Assists — кількість асистів (допомог іншим гравцям у вбивствах) гравця.

Headshots — кількість вбивств головою (headshots), що є показником точності.

HeadshotPercentage — відсоток вбивств головою від загальної кількості вбивств, що дозволяє оцінити точність гравця.

Загальна схема бази даних FaceitHelper, та звязки між таблицями:

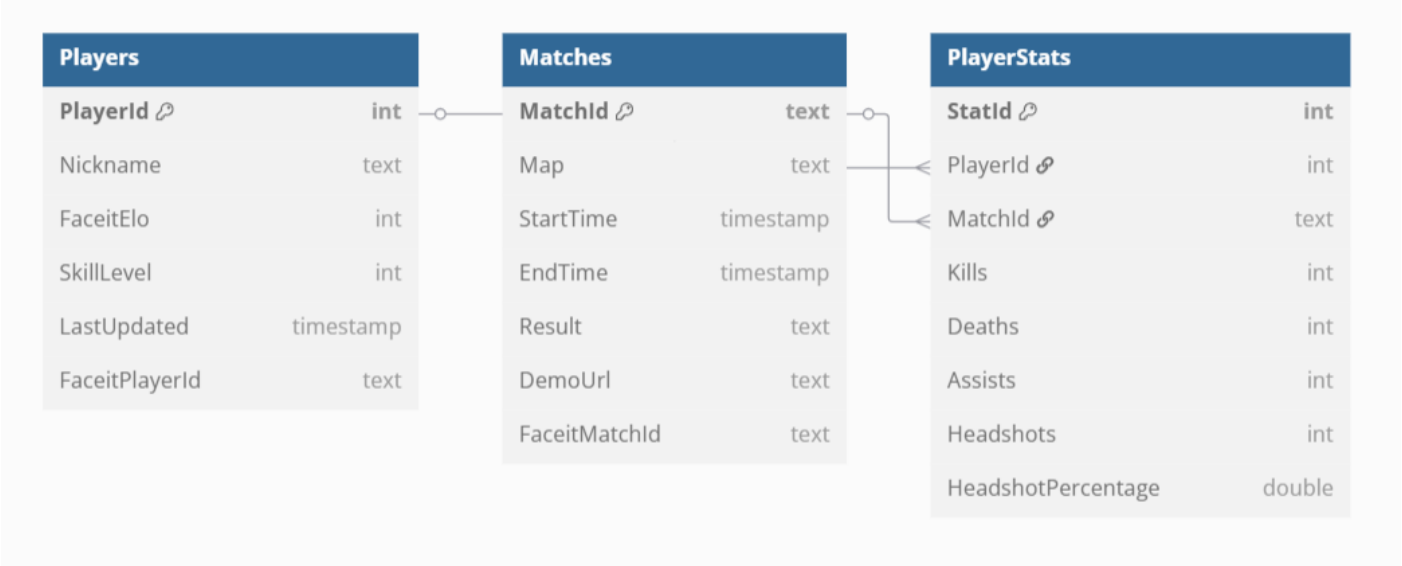


Рисунок 2.4 – Загальна схема зв'язків бази даних

2.2. Побутова об’єктно-орієнтованої моделі

2.2.1. Діаграма класів

Діаграма класів, згідно з Mermaid [7], є фундаментальним елементом об’єктно-орієнтованого моделювання. Вона слугує як для загального концептуального опису структури програми, так і для детального проектування з подальшим перетворенням моделей у програмний код. Крім того, діаграми класів широко застосовуються для моделювання даних. На діаграмі класи відображають ключові компоненти системи, їх взаємодії та ті об’єкти, які потребують реалізації в коді.

Діаграма класів є важливим інструментом, що забезпечує візуалізацію структури програмного проекту, показуючи взаємозв’язки між класами, їхні атрибути та методи. Вона допомагає розробникам краще зрозуміти архітектуру програми, візуалізуючи структуру і зв’язки між компонентами системи, що значно полегшує орієнтацію у кодовій базі. Діаграма також відображає типи взаємозв’язків — асоціації, композиції, агрегації та успадкування, що сприяє глибшому розумінню організації системи і правильному розподілу функціональності між класами.

Крім того, діаграми класів є ефективним засобом комунікації між розробниками та іншими учасниками проєкту, сприяючи обговоренню архітектурних рішень, виявленню потенційних проблем та їх спільному вирішенню. Отже,

побудова діаграм класів є ключовою практикою, яка допомагає структурувати код, покращує розуміння проєкту та підвищує ефективність командної роботи.

Діаграма класів основних сутностей програми FaceitHelper для інтеграції з Discord зображена на рисунку 2.5:

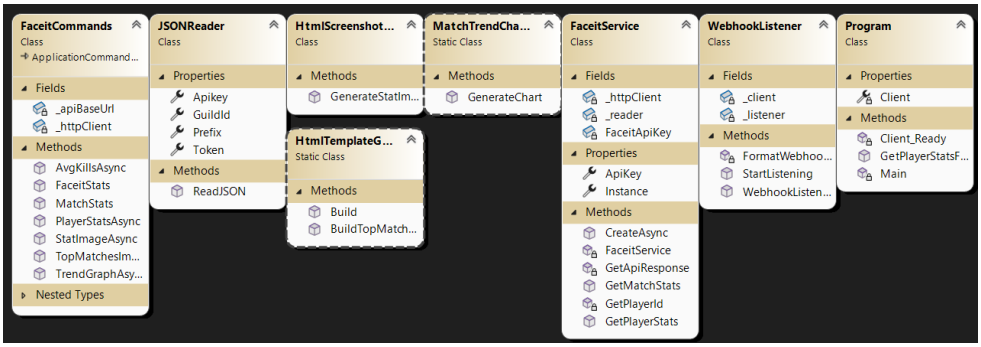


Рисунок 2.5 – Діаграма класів основних компонентів програми FaceitHelper для роботи із Discord

Діаграма класів основних DTO-сутностей програми FaceitHelper, які відповідають за передачу, обробку та візуалізацію статистичних даних у Discord, наведена на рисунку 2.6:

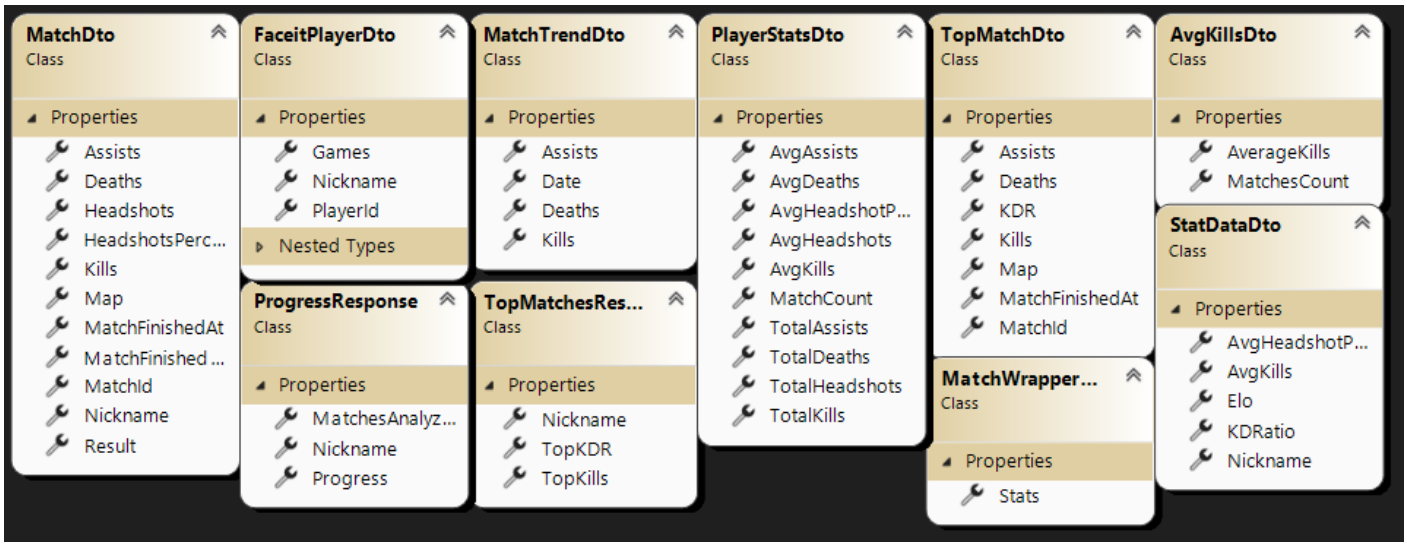


Рисунок 2.6 – Діаграму класів DTO для обробки та передачі даних з власної API,та роботою із Discord

Діаграма класів основних сервісних компонентів програми FaceitHelperAPI, що працює з API та оновленням даних у БД, наведена на рисунку 2.7:

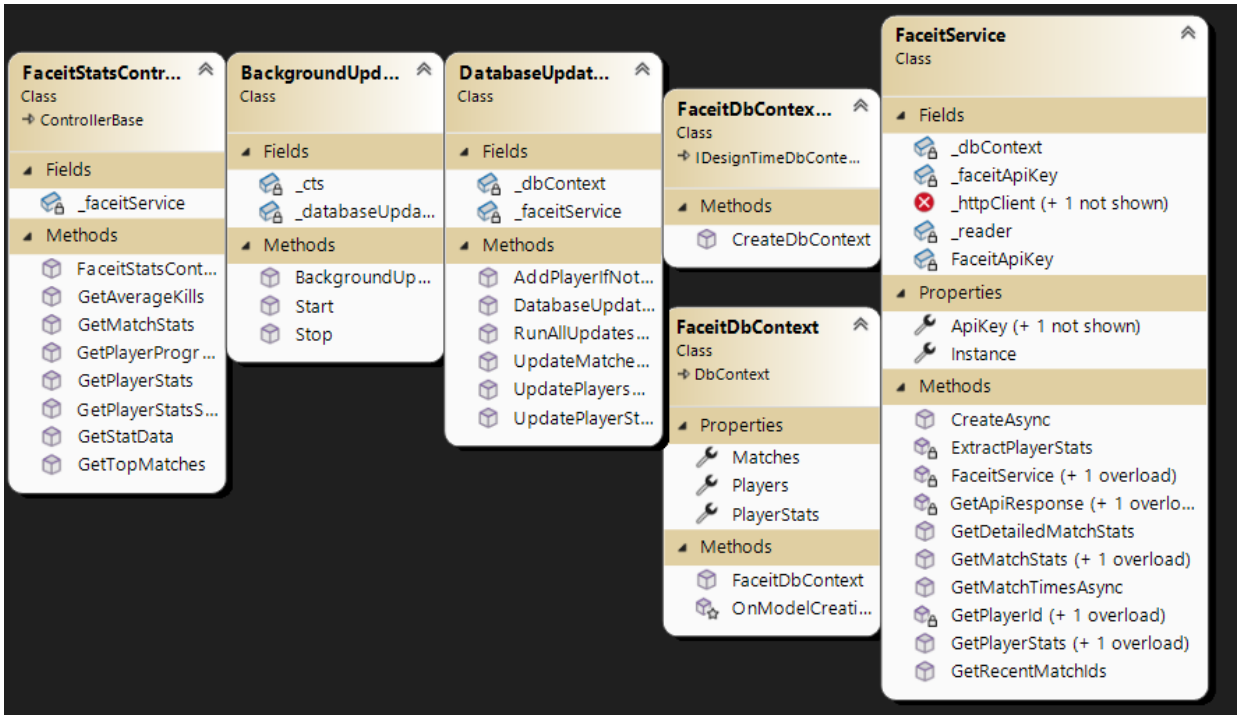


Рисунок 2.7 – Діаграма класів основних сервісних компонентів програми FaceitHelperApi для роботи із БД та FaceitApi

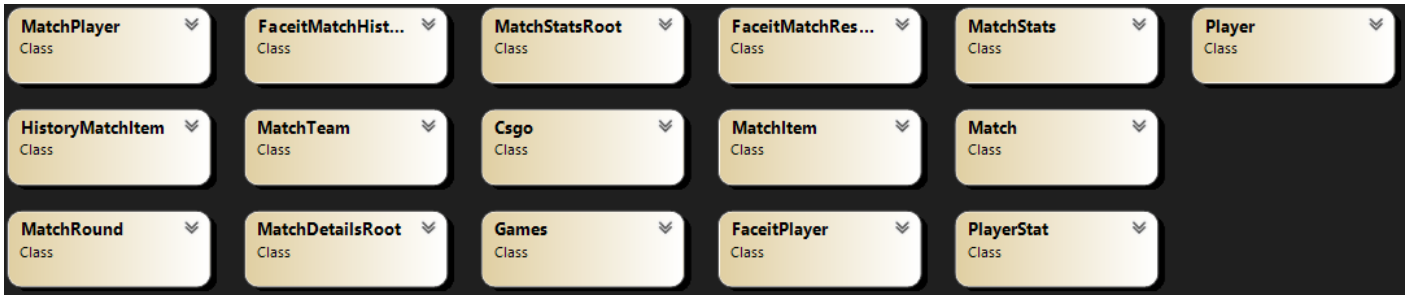


Рисунок 2.8 – Діаграма класів основних DTO і сутностей для обробки та збереження даних з API Faceit та передачі даних програмі яка працює із Discord

2.2.2. Use case діаграма

Відповідно до INASS [8], використання діаграм варіантів використання (use case diagrams) є ефективним засобом опису функціональних вимог програмного забезпечення. Вони допомагають краще зрозуміти, як система повинна функціонувати, моделюючи різноманітні сценарії взаємодії між користувачами та системою, а також визначаючи їхні потреби й очікування.

Діаграми варіантів використання надають розробникам можливість детально аналізувати взаємодію з користувачами та іншими системами, визначати ключові функції і дії, які повинна виконувати система. Вони також допомагають встановити

ролі та взаємини між акторами й системою, а також описати потоки обміну даними між ними.

Головною метою use case діаграм є чітке формулювання функціональних вимог до системи. Це необхідно для того, щоб точно визначити можливості системи та способи взаємодії користувачів із нею, що сприяє виявленню їхніх потреб і очікувань та забезпеченню відповідності реалізації цим вимогам.

Крім того, use case діаграми допомагають уточнити вимоги та пріоритезувати функціональність, забезпечуючи наочне відображення різних сценаріїв використання системи. Це полегшує комунікацію з усіма зацікавленими сторонами, сприяє кращому розумінню вимог і запобігає можливим непорозумінням. В результаті це гарантує, що функціональні можливості системи будуть відповідати потребам користувачів.

Отже, діаграми варіантів використання є важливим інструментом для моделювання і аналізу вимог, що сприяє створенню системи, яка максимально відповідає очікуванням користувачів. Прикладом такого підходу є use case діаграма для системи FaceitHelper — Discord-бота для збору, аналізу та візуалізації ігрової статистики на основі даних платформи FACEIT (рис. 2.9):

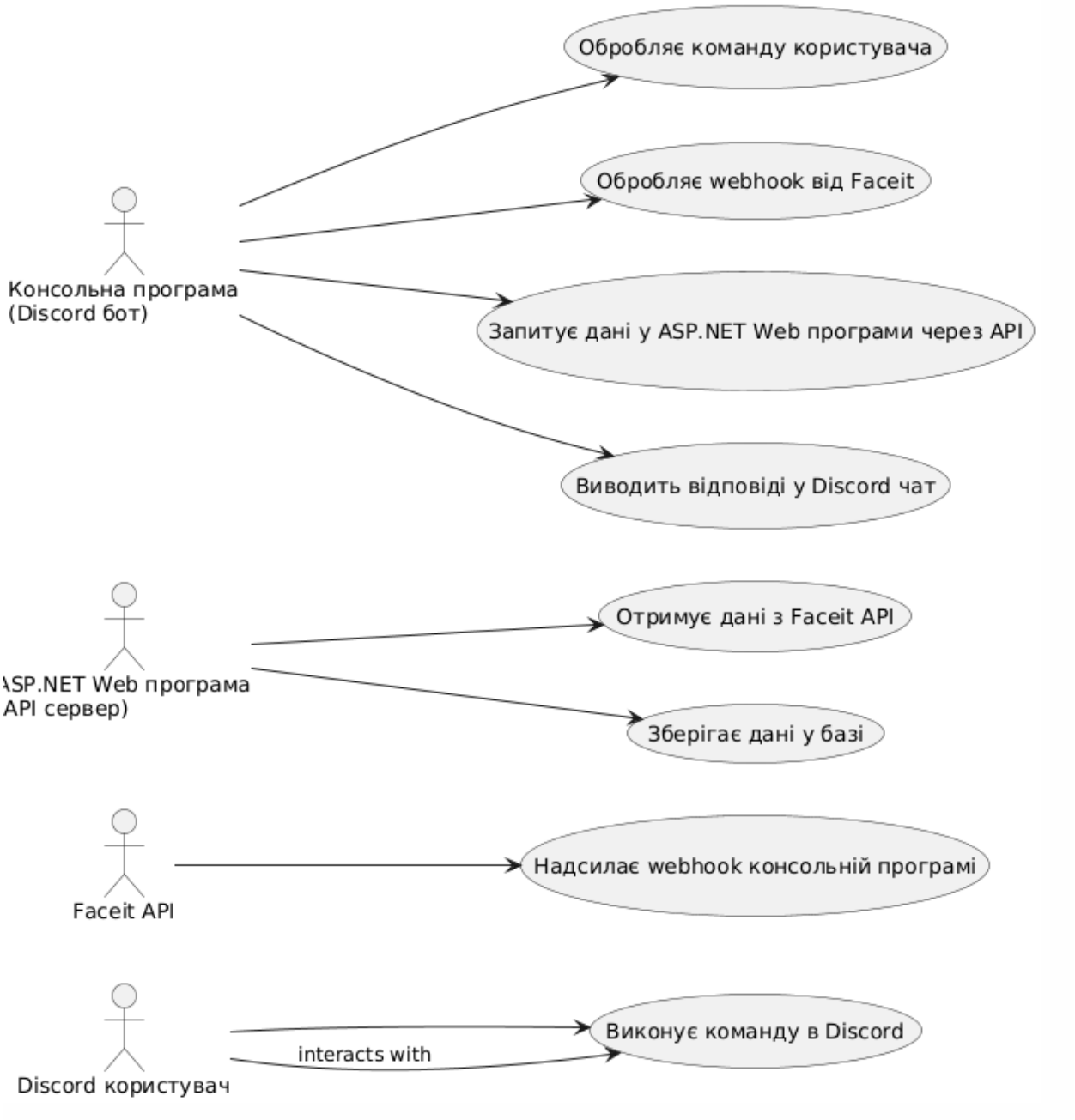


Рисунок 2.9 – Use case діаграма

2.2.3. Архітектура застосунку

Вибір архітектури додатку — це ключовий етап розробки, який визначає структуру системи, спосіб взаємодії компонентів і загальну гнучкість проекту. Архітектура впливає на масштабованість, підтримку, тестування і розгортання системи.

Для мого проекту було прийнято рішення застосувати **архітектуру мікросервісів** [8], що відповідає особливостям взаємодії двох незалежних програм: веб-сервера на ASP.NET [4], який працює з Faceit API [3] і надає власне API для обробки та зберігання даних, і консольної програми — Discord бота, який обробляє команди користувачів та отримує webhook-и від Faceit.

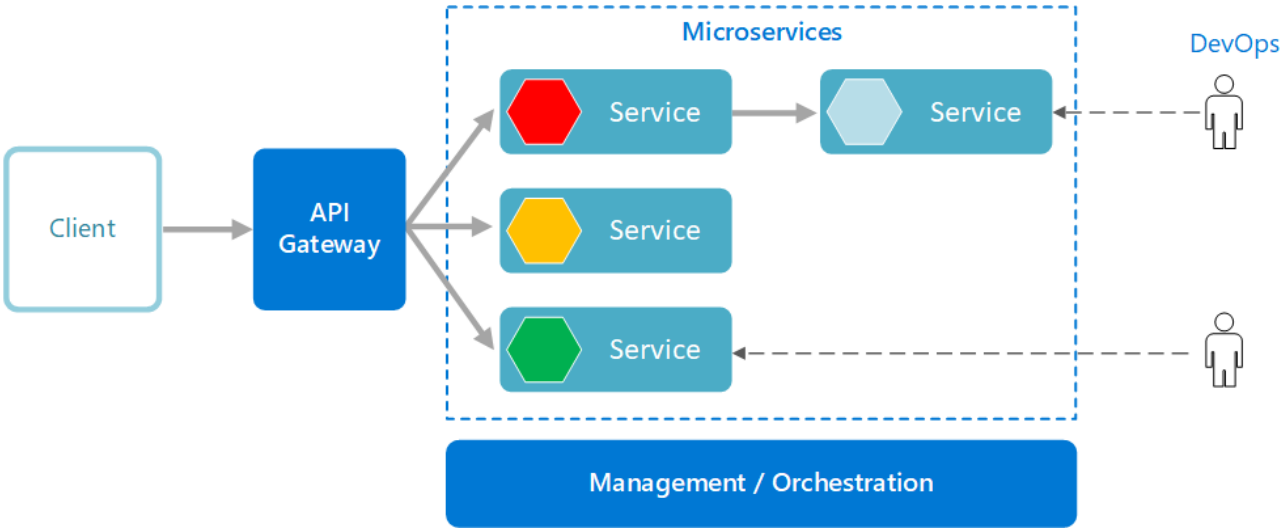


Рисунок 2.10 – Архітектура мікросервісів

Однією з ключових переваг архітектури мікросервісів, як описано у Medium [9], є те, що компонентні служби мають невеликий розмір. Це дозволяє їм бути розробленими невеликими командами з самого початку. Ці команди можуть бути розділені між різними службами, що спрощує масштабування розробки, якщо це необхідно.

Основні переваги мікросервісів:

- Розділення відповідальностей. Веб-програма зосереджена на інтеграції з Faceit API, зберіганні і наданні даних, а консольна програма — на логіці Discord бота та обробці команд.

- Незалежне розгортання і масштабування. Кожна служба може оновлюватися, масштабуватися і підтримуватися окремо, без впливу на іншу.
- Гнучкість у технологіях. Для кожного сервісу можна використовувати оптимальний стек технологій і налаштовувати його окремо.
- Ізоляція помилок. Помилка в одному компоненті (наприклад, Discord боті) не призведе до відмови всього додатку.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

1.1. Засоби розробки

1.1.1. Мова програмування

Під час вибору технологій для реалізації проекту було враховано низку критеріїв, таких як можливість інтеграції з зовнішніми API, продуктивність, масштабованість, швидкість розробки та підтримка кросплатформенності.

У результаті було прийнято рішення використовувати мову програмування **C#** у поєднанні з фреймворком **ASP.NET Core** для створення веб-сервісу, а також консольний застосунок на C# для реалізації Discord бота.

ASP.NET Core у проекті

Як зазначено у джерелі “Pro ASP.NET Core 6” [1], ASP.NET Core — це сучасна, кросплатформенна та високопродуктивна платформа для розробки веб-додатків і API від Microsoft. Вона включає модульну архітектуру, що дозволяє гнучко налаштовувати компоненти і розгортати застосунки на Windows, Linux чи macOS.

У контексті проекту ASP.NET Core забезпечує:

- Створення REST API, що взаємодіє із зовнішнім API Faceit, отримує, обробляє і зберігає дані.
- Безпечний обмін даними між веб-сервісом і Discord ботом.
- Високу продуктивність завдяки оптимізованому середовищу виконання і підтримці асинхронних операцій.
- Масштабованість і підтримку мікросервісної архітектури, що відповідає сучасним вимогам до розробки.

Консольний застосунок на C#

Для обробки команд користувачів Discord та реакції на webhook-и від Faceit обрана консольна програма на C#.

Такий підхід дозволяє:

- Легко інтегруватися з Discord API для отримання та обробки команд.
- Реалізовувати логіку бота поза межами веб-сервісу, що підвищує гнучкість

системи.

- Забезпечувати швидку обробку подій у реальному часі.
- Підтримувати простоту розгортання і оновлення окремих компонентів системи.

Основні переваги обраних технологій:

- Кросплатформність. Обидва компоненти проекту можна запускати на різних операційних системах без істотних змін.
- Висока продуктивність і масштабованість. ASP.NET Core дозволяє обробляти велику кількість запитів ефективно, а мікросервісна архітектура дає змогу масштабувати окремі частини системи.
- Розділення відповідальностей. Веб-сервіс відповідає за інтеграцію з зовнішніми API та зберігання даних, а консольна програма — за обробку команд і взаємодію з Discord.
- Активна підтримка і розвинена екосистема. Використання технологій Microsoft забезпечує доступ до широкого спектру інструментів, бібліотек і спільноти.

Використання C# та ASP.NET Core у поєднанні з консольною програмою для Discord бота є оптимальним рішенням для реалізації масштабованої, безпечної та гнучкої системи інтеграції з Faceit API і Discord. Такий підхід сприяє швидкій розробці, простоті підтримки та подальшому розвитку проекту.

1.1.2. База даних

Система керування базами даних (СКБД) є ключовим компонентом будь-якого програмного продукту, що працює з інформацією. Правильний вибір СКБД істотно впливає на продуктивність, надійність та масштабованість вашої системи. Для реалізації рішення по управлінню та розвитку стартап-команди було обрано PostgreSQL — потужну, відкриту і стабільну систему керування базами даних.

Основні переваги PostgreSQL:

Відкрите програмне забезпечення та кросплатформеність PostgreSQL — це система з відкритим кодом, що підтримується на різних операційних платформах, таких як Windows, Linux і macOS, забезпечуючи гнучкість у виборі середовища розгортання.

Висока продуктивність та можливості масштабування PostgreSQL ефективно обробляє великі обсяги інформації, підтримуючи як вертикальне, так і горизонтальне масштабування, що є важливим для динамічних стартапів.

Розвинуті функції безпеки Система пропонує широкий спектр інструментів для аутентифікації, шифрування, контролю доступу та аудиту, що забезпечує надійний захист даних від несанкціонованого доступу.

Гнучкість у роботі з даними Підтримка складних типів даних, включно з JSON/JSONB, повнотекстовим пошуком та процедурними мовами дозволяє реалізувати складну бізнес-логіку на рівні СКБД.

Інтеграція з ASP.NET Core Завдяки широкому вибору драйверів та ORM-бібліотек (наприклад, Entity Framework Core, Npgsql) PostgreSQL легко інтегрується з екосистемою .NET [11], спрощуючи розробку та обслуговування додатків.

Активна спільнота та підтримка PostgreSQL користується підтримкою великої та активної спільноти, що гарантує доступ до докладної документації, ресурсів та регулярних оновлень.

Попри те, що робота з PostgreSQL вимагає певного рівня спеціалізованих знань, її численні переваги роблять цю СКБД відмінним вибором для проектів на базі C# та ASP.NET Core, які прагнуть до створення масштабованих, безпечних та високопродуктивних веб-рішень. Відкритість і гнучкість PostgreSQL сприяють стабільності роботи та простоті подальшого розвитку вашого продукту.

1.2. Вимоги до технічного та програмного забезпечення

Для ефективної роботи системи та коректної розробки проекту необхідно врахувати такі вимоги:

Вимоги до середовища використання:

- Discord: Основний інтерфейс взаємодії користувачів із системою — через Discord. Користувачі виконують команди у чаті Discord, тому веб-браузер для користувача не потрібен.
- Підключення до інтернету: Стабільне інтернет-з’єднання необхідне для взаємодії Discord бота із серверами та зовнішніми API (Faceit).

Вимоги до середовища розробки:

- Операційна система: Підтримуються Windows, macOS, Linux.

Інструменти розробки:

- .NET SDK для розробки ASP.NET Core Web API та консольної програми — Discord бота.
- IDE: Visual Studio, Visual Studio Code або JetBrains Rider. PostgreSQL — система керування базами даних.
- Інтернет-з’єднання для завантаження необхідних бібліотек і оновлень.

Функціональні можливості системи:

- Обробка команд у Discord: Консольна програма приймає та обробляє команди від користувачів через Discord.
- Отримання даних із зовнішнього API Faceit: Веб-сервіс на ASP.NET Core отримує статистичні дані з Faceit API та зберігає їх у базі даних.
- Взаємодія між веб-сервісом і Discord ботом: Бот звертається до API веб-сервера для отримання актуальної інформації, яку потім відображає у чаті Discord.
- Обробка webhook: Консольна програма отримує webhook-и від Faceit про створення та матчів і відповідно реагує на ці події.

1.3. Опис програмної реалізації

1.3.1. Загальна архітектура проекту

Проект реалізовано у вигляді розподіленої системи, що складається з двох основних компонентів — веб-сервісу та консольної програми, які взаємодіють між собою та забезпечують комплексний функціонал для роботи з ігровою статистикою платформи FACEIT у середовищі Discord.

Компоненти системи:

- Веб-сервіс на базі ASP.NET Core

Цей компонент відповідає за безпосередню взаємодію з відкритим API платформи FACEIT. Він здійснює отримання детальної статистичної інформації про гравців, матчі та події, а також зберігає ці дані у реляційній базі даних PostgreSQL. Веб-сервіс реалізує REST API, через яке зовнішні клієнти, зокрема Discord-бот, можуть отримувати актуальні дані. Також веб-сервіс відповідає за обробку і збереження історії подій, що забезпечує швидкий і надійний доступ до інформації.

- Консольний Discord-бот

Discord-бот реалізовано як окремий консольний додаток, що забезпечує інтерактивну взаємодію з користувачами у месенджері Discord. Він приймає команди від користувачів, надсилає запити до веб-сервісу, отримує актуальні статистичні дані та формує зрозумілі відповіді у вигляді тексту та візуалізацій. Крім того, бот отримує webhook-повідомлення від FACEIT про створення та завершення матчів і оперативно реагує на них, повідомляючи користувачів про важливі події.

Взаємодія між компонентами

Архітектура побудована за принципом мікросервісів, що дозволяє розділити логіку між двома незалежними програмними модулями. Веб-сервіс і бот взаємодіють через стандартизований REST API, що забезпечує гнучкість, масштабованість та можливість подальшого розвитку системи. Веб-сервіс регулярно оновлює статистичні дані із зовнішнього API FACEIT, зберігає їх у базі даних і забезпечує їх доступність для бота. Бот, у свою чергу, відповідає за користувацький інтерфейс у Discord, обробляє команди та події в режимі реального часу.

Переваги обраної архітектури

- Модульність і розділення відповідальностей: Різні компоненти проекту відповідають за окремі функції, що спрощує розробку, тестування і підтримку.
- Масштабованість: Кожен компонент можна розширювати та масштабувати незалежно від іншого, що дозволяє ефективно управляти навантаженням.
- Гнучкість: Можливість оновлювати або замінювати компоненти без впливу на всю систему.
- Надійність: У разі збою одного з компонентів інші можуть продовжувати працювати, забезпечуючи безперервність сервісу.

Таким чином, архітектура проекту забезпечує надійну, масштабовану та гнучку платформу для інтеграції ігрової статистики FACEIT у середовище Discord з високою якістю обробки даних і комфортом для користувачів.

1.3.2. Використані технології та бібліотеки

Для реалізації проекту було обрано надійні та перевірені технології, які відповідають функціональним вимогам і забезпечують стабільну роботу системи.

- Мова програмування C# — використовується як для розробки веб-сервісу, так і для консольного Discord-бота.
- .NET Framework 4.7 — платформа для розробки Discord-бота, яка забезпечує сумісність із бібліотекою DSharpPlus і стабільну роботу у середовищі Windows.
- .NET Core / .NET 6+ — застосовується для розробки веб-сервісу ASP.NET Core, що надає REST API для взаємодії з Discord-ботом.
- ASP.NET Core — фреймворк для побудови веб-API з високою продуктивністю і підтримкою кросплатформенності.
- PostgreSQL — реляційна база даних, яка використовується для надійного зберігання статистичних даних.
- Entity Framework Core — ORM, що спрощує роботу з базою даних у веб-сервісі .
- DSharpPlus — бібліотека для створення Discord-бота [1], сумісна з .NET

Framework 4.7, що забезпечує доступ до Discord API.

- PuppeteerSharp — використовується для генерації графічних інфографік [12] на основі HTML/CSS через headless Chromium.
- Newtonsoft.Json — бібліотека для роботи з JSON, яка застосовується для обробки API-відповідей і конфігурацій.
- JSON-конфігураційні файли — для зберігання параметрів доступу до зовнішніх сервісів (Discord API, Faceit API) та налаштувань роботи бота.

Цей стек технологій дозволяє побудувати гнучку, стабільну і масштабовану систему, яка відповідає вимогам високої продуктивності і забезпечує комфортну взаємодію користувачів із сервісом.

1.3.3. Реалізація веб-сервісу (ASP.NET Core API)

Веб-сервіс реалізовано з використанням платформи ASP.NET Core [9], що дозволяє створити високопродуктивний та масштабований REST API для взаємодії з Discord-ботом і зовнішнім API Faceit.

Основні функції веб-сервісу:

- Отримання статистичних даних: Веб-сервіс здійснює запити до відкритого API платформи FACEIT, отримує інформацію про гравців, матчі, рейтинги та інші ігрові метрики.
- Збереження даних: Отримані дані зберігаються у базі даних PostgreSQL з використанням Entity Framework Core, що забезпечує зручну роботу з реляційними даними.
- Надання API: Веб-сервіс надає REST API для доступу до статистики та інформації, що дозволяє Discord-боту швидко отримувати необхідні дані.
- Безпека: Використовуються стандартні механізми захисту API, включаючи конфігураційні файли для збереження ключів доступу і аутентифікації.
- Асинхронність: Для забезпечення високої продуктивності всі операції з API та базою даних виконуються асинхронно.

Приклад коду контролера для отримання статистики гравця:

```
[ApiController]
[Route("api/[controller]")]
public class FaceitController : ControllerBase
{
    private readonly IFaceitService _faceitService;
    public FaceitController(IFaceitService faceitService)
    {
        _faceitService = faceitService;
    }
    // GET api/faceit/player/{playerId}
    [HttpGet("player/{playerId}")]
    public async Task<IActionResult> GetPlayerStats(string playerId)
    {
        var stats = await _faceitService.GetPlayerStatsAsync(playerId);
        if (stats == null)
            return NotFound();
        return Ok(stats);
    }
}
```

Цей контролер приймає HTTP GET-запит з ідентифікатором гравця, звертається до сервісу для отримання статистики та повертає результат у форматі JSON.

Приклад сервісного класу для роботи з API FACEIT:

```
public class FaceitService : IFaceitService
{
    private readonly HttpClient _httpClient;
    public FaceitService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }
    public async Task<PlayerStats> GetPlayerStatsAsync(string playerId)
```

```
    {
        var response = await _httpClient.GetAsync($"https://api.faceit.com/players/{playerId}/stats");
        if (!response.IsSuccessStatusCode)
            return null;
        var content = await response.Content.ReadAsStringAsync();
        var stats = JsonConvert.DeserializeObject<PlayerStats>(content);
        return stats;
    }
}
```

Цей сервісний клас відповідає за запит до API FACEIT, отримання даних і десеріалізацію JSON-відповіді у об’єкт моделі.

Модель даних для збереження статистики гравця:

```
public class PlayerStats
{
    public string PlayerId { get; set; }
    public int Elo { get; set; }
    public int MatchesPlayed { get; set; }
    public double KillDeathRatio { get; set; }
    // Інші релевантні поля статистики
}
```

Модель містить основні поля, які відображають ключові показники гравця, необхідні для подальшого аналізу та відображення.

Особливості реалізації

- Використання асинхронних методів дозволяє не блокувати потік виконання і підвищує масштабованість системи.
- Логування і обробка помилок реалізовані на рівні middleware та сервісів для забезпечення стабільності.
- REST API забезпечує зручний і стандартизований інтерфейс для споживачів даних, таких як Discord-бот.
- Веб-сервіс конфігурується через зовнішні файли, що забезпечує гнучкість при розгортанні.

Зараз опишу декілька запитів для API:

На рисунку 3.1 представлено список запитів до API. У кожному з них фігурує параметр nickname — ім’я користувача на платформі Faceit. Цей параметр визначає, для якого саме гравця виконується запит. Також у деяких запитах присутній параметр matchCount, який вказує кількість матчів, що враховуються під час обчислення статистики.

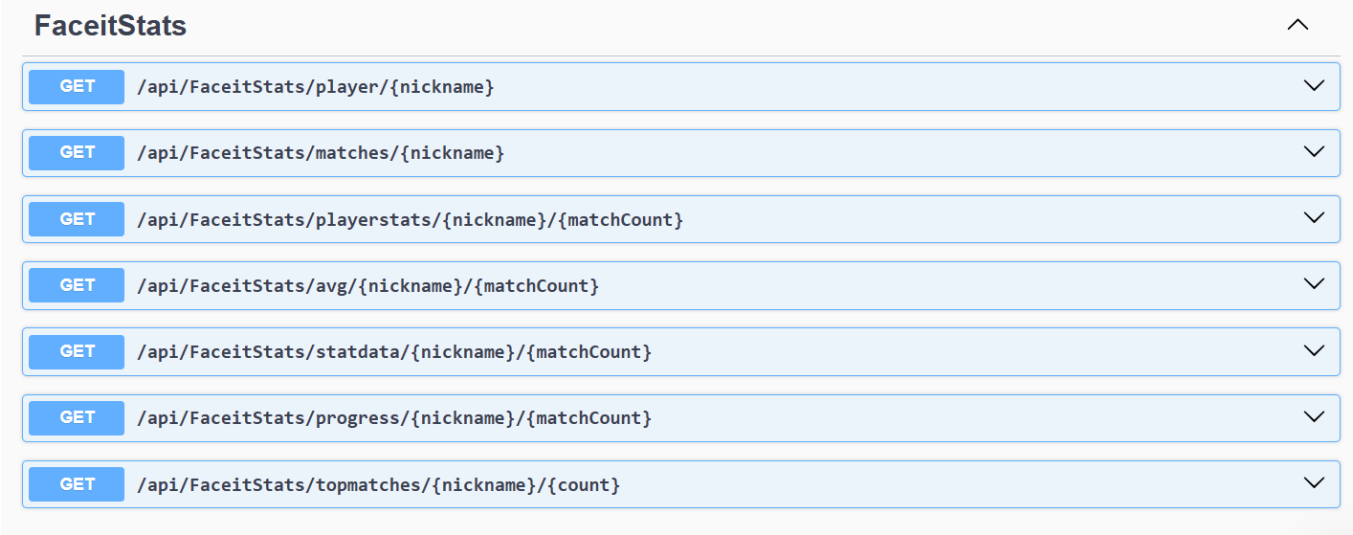


Рисунок 3.1 – Список запитів до API

Тепер давайте розглянемо декілька прикладів запитів що потрібно їм передати і що ми отримуємо, на рисунку 3.2 зображено за його ім’ям (нікнеймом) на платформі Faceit. Для виконання запиту необхідно обов’язково вказати параметр nickname. У разі коректного запиту сервер повертає відповідь у форматі JSON (application/json), яка містить такі поля:

- **nickname** (string): нікнейм гравця;
- **matchesPlayed** (integer): кількість зіграних матчів;
- **rank** (string): ранг або рівень гравця;
- **rating** (number/float): рейтинг гравця.

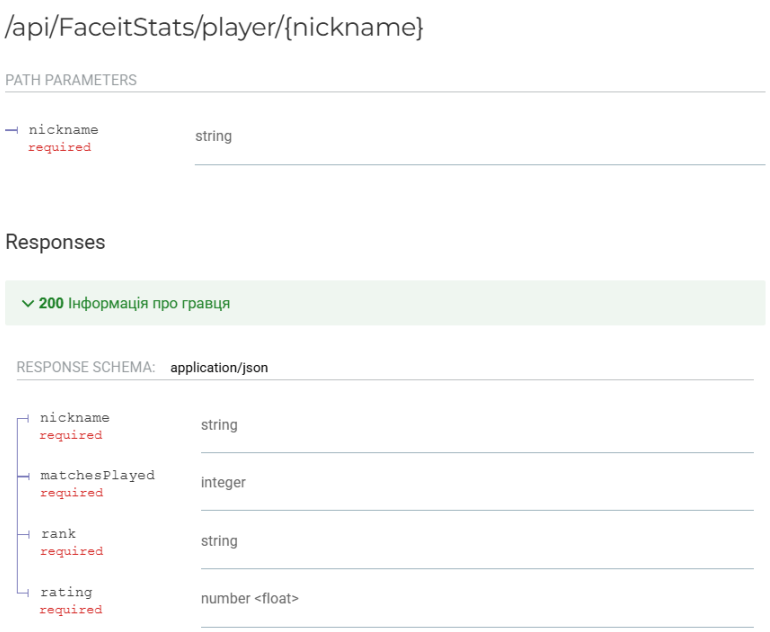


Рисунок 3.2 – Інтерфейс API для отримання статистики гравця

Тепер розглянемо приклад запиту до API, зображеного на рисунку 3.3. В цьому запиті потрібно отримати прогрес гравця у грі за певну кількість матчів. Для його успішного виконання обов’язково необхідно вказати параметр nickname, що є ім’ям користувача на платформі Faceit.

Крім того, у запиті використовується параметр matchCount, який визначає кількість матчів, за якими буде обчислюватись прогрес. Значення цього параметра можна задати явно, або залишити за замовчуванням (10).

У випадку коректного запиту сервер повертає відповідь у форматі JSON (application/json), що містить такі поля:

- **nickname** (string): нікнейм гравця;
- **progress** (масив об'єктів): містить інформацію про прогрес гравця у окремих матчах або за певний період.

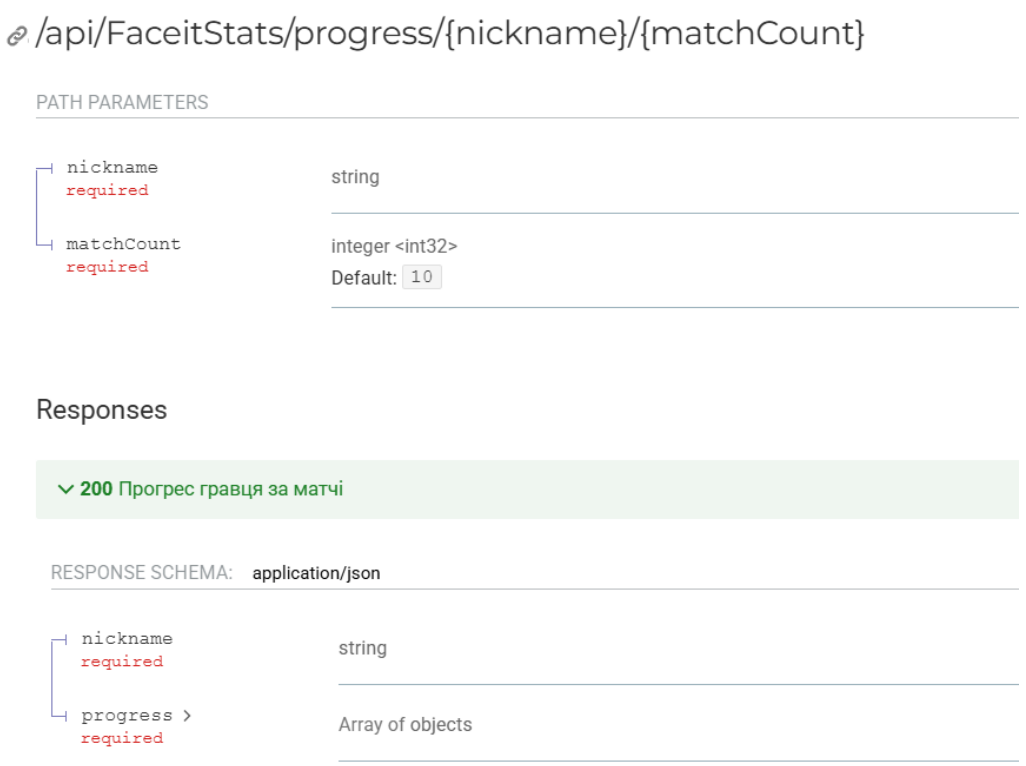


Рисунок 3.3 – Інтерфейс API для отримання прогрему користувача

1.3.4 Обробка webhook

У проекті реалізовано клас WebhookListener, який відповідає за прийом і обробку webhook-повідомлень від платформи FACEIT. Клас створює локальний HTTP-сервер, що прослуховує запити за адресою http://localhost:8080/ і асинхронно приймає HTTP POST-запити з інформацією про події [9].

Після отримання webhook у форматі JSON дані десеріалізуються, після чого формується текстове повідомлення, що відображає основну інформацію про подію, таку як тип події, ідентифікатор матчу, учасники та час.

Це повідомлення надсилається у перший текстовий канал першої гільдії Discord за допомогою клієнта Discord [1].

Після успішного прийому та обробки webhook відправляється HTTP-відповідь із підтвердженням. Основний метод обробки webhook:

Основний метод обробки webhook:

```
        public async Task StartListening()
        {
            _listener.Start();
            while (true)
            {
                var context = await _listener.GetContextAsync();
                string body;
                using (var reader = new StreamReader(context.Request.InputStream, Encoding.UTF8))
                    body = await reader.ReadToEndAsync();
                // Збереження JSON у файл (опціонально)
                File.WriteAllText($"webhook_{DateTime.Now:yyyyMMdd_HHmmss}.json", body);
                dynamic data = JsonConvert.DeserializeObject(body);
                string message = FormatWebhookMessage(data);
                var channel = _client.Guilds.Values.FirstOrDefault()?.Channels.Values.FirstOrDefault(c => c.Type == ChannelType.Text);
                if (channel != null)
                    await channel.SendMessageAsync(message);

                byte[] buffer = Encoding.UTF8.GetBytes("Webhook received successfully!");
                context.Response.ContentLength64 = buffer.Length;
                context.Response.StatusCode = 200;
                await context.Response.OutputStream.WriteAsync(buffer, 0, buffer.Length);
                context.Response.OutputStream.Close();
            }
        }
```

Форматування повідомлень про події: Повідомлення адаптуються залежно від типу події webhook, що дозволяє користувачам отримувати інформативні сповіщення в Discord.

```
try
{
    var team1 = data?.payload?.teams?[0]?.roster?[0]?.nickname;
    var team2 = data?.payload?.teams?[1]?.roster?[0]?.nickname;
    string players = $"{team1} vs {team2}";
    switch ((string)eventType)
    {
        case "match_object_created":
            return $"🆕 Новий матч створено!\n🎮 Match ID: `{matchId}`\n🌍 Регіон: {region}\n📦 Тип: {entityName}\n🕒 Час створення: {createdAt}";
        case "match_status_ready":
            return $"✅ Матч готовий до початку!\n🎮 Match ID: `{matchId}`\n👥 Гравці: {players}\n🌍 Регіон: {region}\n🚀 Переходьте до гри!";
        case "match_status_finished":
            return $"🏁 Матч завершено!\n🎮 Match ID: `{matchId}`\n🕒 Початок: {startedAt}\n🕒 Кінець: {finishedAt}\n👥 Гравці: {players}\n📸 Очікуйте демо!";
        default:
            return $"📥 Подія: `{eventType}` для матчу `{matchId}`";
    }
}
catch
{
    return $"📥 Подія: `{eventType}` (⚠ помилка при парсингу деталей)";
}
```

1.3.5 Обробка користувацьких команд і формування відповіді

У проекті реалізовано механізм обробки команд користувачів у Discord за допомогою бібліотеки DSharpPlus [1]. Консольний Discord-бот приймає slash-команди, які дозволяють користувачам отримувати різноманітну статистику з платформи FACEIT [3].

Основні функції:

- Обробка базових команд, таких як /faceitstats, /matchstats, /playerstats тощо.
- Запити до REST API веб-сервісу для отримання актуальних даних [9].
- Формування текстових відповідей із ключовою статистикою.
- Генерація графічних інфографік за допомогою PuppeteerSharp, які

надсилаються як зображення у чат Discord.

- Асинхронна обробка команд для забезпечення швидкої відповіді.

Приклад обробки команди

```
[SlashCommand("faceitstats", "Отримати базову статистику гравця FACEIT")]
public async Task FaceitStatsCommand(InteractionContext ctx, [Option("playerId", "Ідентифікатор гравця FACEIT")] string playerId)
{
    await ctx.DeferAsync();
    var stats = await _faceitService.GetPlayerStatsAsync(playerId);
    if (stats == null)
    {
        await ctx.EditResponseAsync("Статистика не знайдена.");
        return;
    }
    string responseMessage = $"Рівень гравця: {stats.SkillLevel}\nELO: {stats.Elo}\nМатчів зіграно: {stats.MatchesPlayed}";
    await ctx.EditResponseAsync(responseMessage);
}
```

Цей механізм дозволяє користувачам ефективно отримувати потрібну інформацію безпосередньо в чаті Discord, підвищуючи зручність та інтерактивність системи.

1.3.6 Візуалізація статистики (генерація інфографіки)

Для покращення сприйняття статистичних даних у проекті реалізована генерація графічних інфографік у форматі PNG на основі HTML-шаблонів. Технології та реалізація

- Використовується бібліотека PuppeteerSharp для керування headless-браузером Chromium, який рендерить HTML+CSS у зображення.
- HTML-шаблони містять стильне оформлення для візуального представлення основних статистичних показників.
- Згенеровані PNG-файли відправляються у Discord у відповідь на відповідні команди користувачів [1].

Приклад коду генерації зображення

```
public class HtmlScreenshotService
{
    public async Task<Stream> GenerateStatImageFromHtmlAsync(string htmlContent)
    {
        var launchOptions = new LaunchOptions
        {
            Headless = true,
            ExecutablePath = @"E:\unik\diploma_db\ungoogled-chromium-136.0.7103.97-1_Win64\chrome.exe",
            Args = new[] { "--no-sandbox" }
        };
        using var browser = await Puppeteer.LaunchAsync(launchOptions);
        using var page = await browser.NewPageAsync();
        await page.SetContentAsync(htmlContent);
        await page.SetViewportAsync(new ViewPortOptions { Width = 800, Height = 600 });
        var container = await page.QuerySelectorAsync(".container");
        if (container == null) throw new Exception("HTML не містить елементу .container");
        var bytes = await container.ScreenshotDataAsync();
        return new MemoryStream(bytes);
    }
}
```

Приклад HTML-шаблону для статистики

```
public static class HtmlTemplateGenerator
{
    public static string Build(string nickname, int elo, double avgKills, double kd, double hs)
    {
        return $@"
<html>
<head>
  <!-- стилі пропущено -->
</head>
<body>
  <div class='container'>
    <div class='nickname'>{nickname}</div>
    <div class='stat'>🔥 ELO: <b>{elo}</b></div>
    <div class='stat'>🎯 AVG kills: <b>{avgKills:F1}</b></div>
    <div class='stat'>⚔ K/D: <b>{kd:F2}</b></div>
    <div class='stat'>🔫 HS%: <b>{hs:F1}%</b></div>
  </div>
</body>
</html>";
    }
}
```

На рисунку 3.4 наведено візуалізовану статистику у вигляді фото:

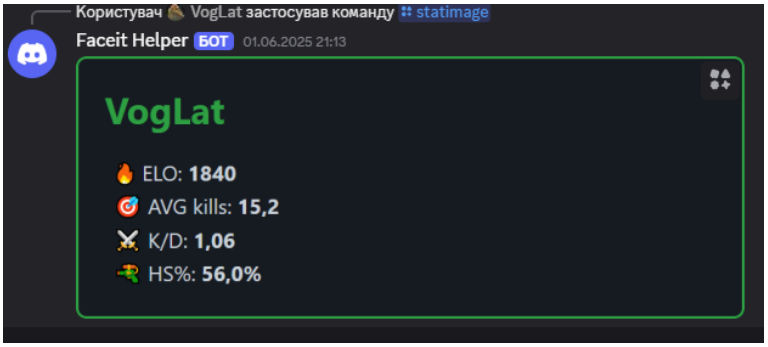


Рисунок 3.4 – Візуалізована статистика гравця

На рисунку 3.5 наведено статистику у вигляді графіків на яких наведено основні статистики гравця у матчах:

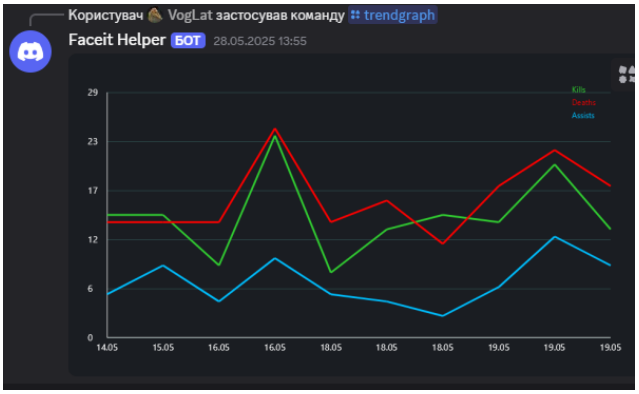


Рисунок 3.5 – Візуалізована статистика гравця у вигляді графіків

1.3.7 Робота з базою даних (PostgreSQL, Entity Framework Core)

У проекті для збереження статистичних даних використовується реляційна база даних **PostgreSQL**, інтегрована з веб-сервісом через ORM-бібліотеку **Entity Framework Core** [2,4]. Це дозволяє працювати з базою даних через об'єктно-орієнтовані моделі, спрощуючи доступ до даних і підтримку схеми.

Контекст бази даних.

Основний клас контексту бази даних — FaceitDbContext — визначає таблиці і зв’язки між ними:

```
public class FaceitDbContext : DbContext
{
   public FaceitDbContext(DbContextOptions<FaceitDbContext> options) : base(options) { }
   public DbSet<Player> Players { get; set; }
   public DbSet<Match> Matches { get; set; }
   public DbSet<PlayerStat> PlayerStats { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
           modelBuilder.Entity<PlayerStat>()
              .HasKey(ps => ps.StatId);
           modelBuilder.Entity<PlayerStat>()
              .HasOne(ps => ps.Player)
              .WithMany(p => p.PlayerStats)
              .HasForeignKey(ps => ps.PlayerId);
           modelBuilder.Entity<PlayerStat>()
              .HasOne(ps => ps.Match)
              .WithMany(m => m.PlayerStats)
              .HasForeignKey(ps => ps.MatchId);
        }
     }
```

Опис моделей

- Player — гравець, містить унікальний ідентифікатор, нікнейм і зв’язок із статистикою.
- Match — матч, з інформацією про час, карту та інші деталі.
- PlayerStat — детальна статистика гравця в конкретному матчі, пов’язана з відповідними гравцем та матчем.

Особливості реалізації

- Використовується патерн Code First, що дозволяє описувати структуру бази

даних через класи і автоматично створювати відповідні таблиці [4].

- Зв’язки між таблицями (один-до-багатьох) реалізовані за допомогою Fluent API у методі OnModelCreating.
- Підтримується цілісність даних і оптимізована робота з запитами завдяки налаштуванням ключів і зовнішніх ключів.

1.3.8 Тестування та перевірка функціональності

Для забезпечення стабільності та коректної роботи системи проведено комплексне тестування основних функціональних модулів, включаючи обробку команд користувачів у Discord, прийом webhook подій від FACEIT та збереження даних у базу.

Тестування команд Discord

1. Перевірка коректності отримання і обробки slash-команд.

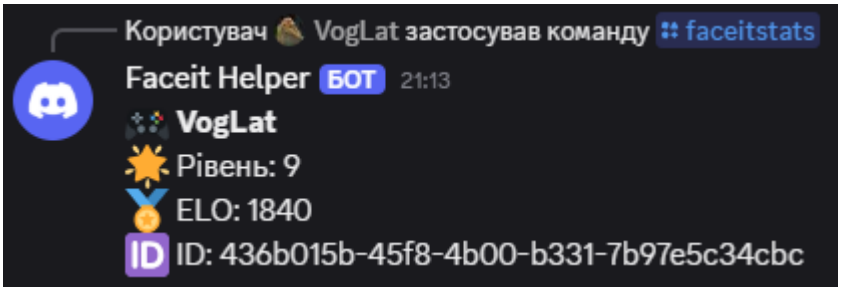


Рисунок 3.6 – Відповідь Discord-бота на команду faceitstats з основною статистикою гравця

2. Валідація параметрів команд, перевірка відповідей бота.
3. Тестування як текстових відповідей, так і генерації графічної статистики.

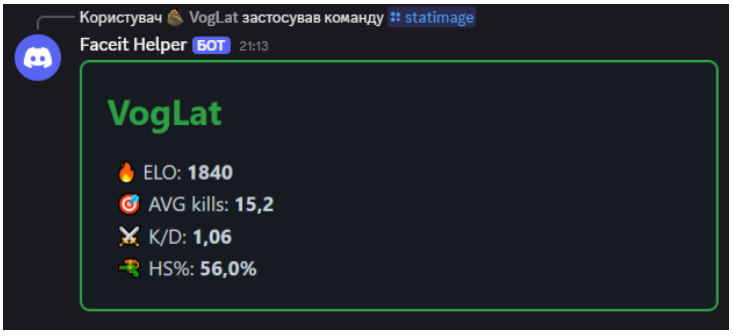


Рисунок 3.7 – Відповідь Discord-бота на команду statimage з основною статистикою гравця

4. Тестування граничних випадків, наприклад, запити з некоректними або відсутніми даними.

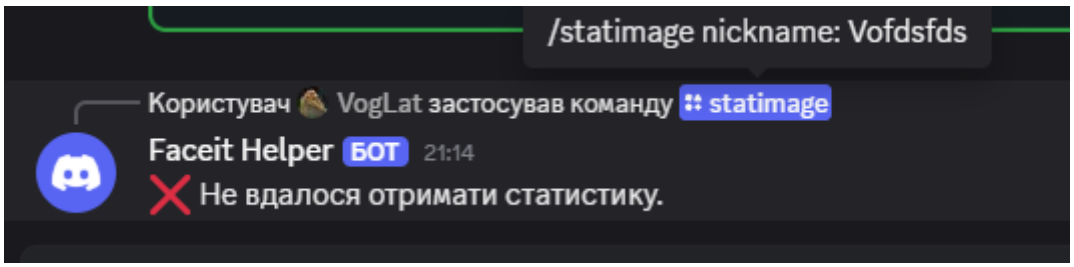


Рисунок 3.8 – Відповідь Discord-бота на команду statimage із повідомленням про помилку отримання статистики

Тестування webhook

1. Імітація надходження webhook подій (створення матчу, завершення матчу тощо) за допомогою інструментів Postman або власних тестових скриптів.

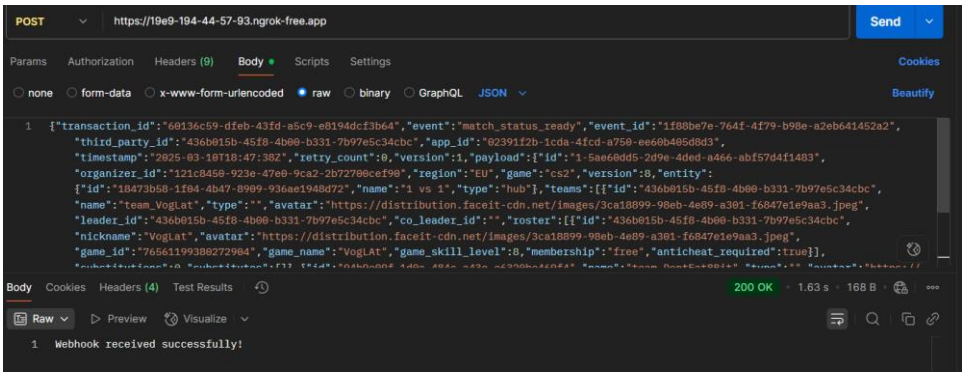


Рисунок 3.9 – Імітація отримання події від FACEIT для тестування обробки webhook.

2. Перевірка коректності парсингу вхідних даних.
3. Перевірка логування і збереження оригінальних JSON у файли для аудиту.

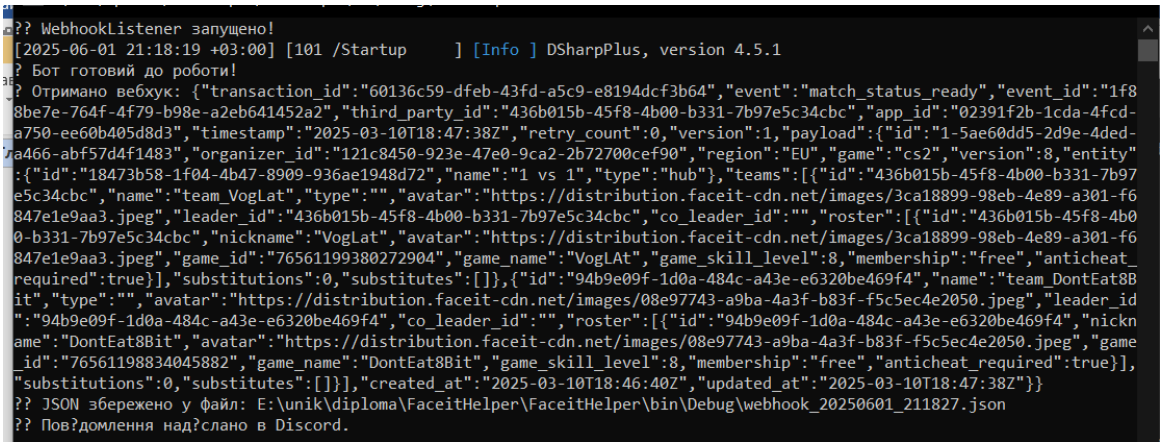


Рисунок 3.10 – Лог роботи WebhookListener у консолі

4. Переконання, що повідомлення про події успішно надсилаються у Discord.

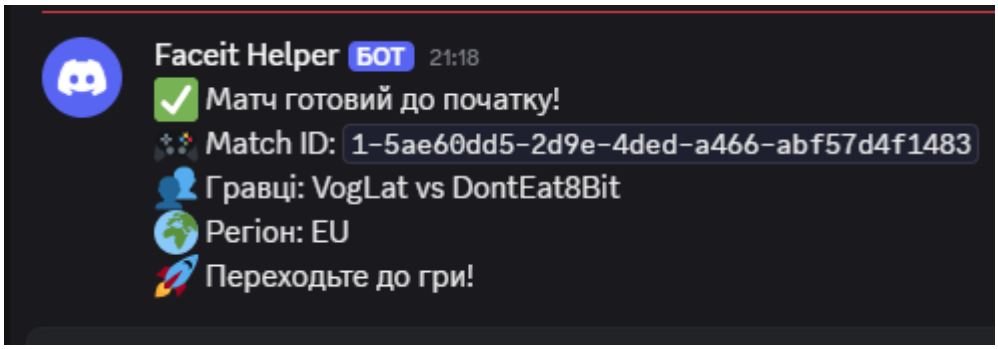


Рисунок 3.11 – Повідомлення Discord-бота про готовність матчу, надіслане у текстовий канал

Тестування роботи з базою даних

1. Перевірка коректності запису отриманих статистичних даних у таблиці PostgreSQL.

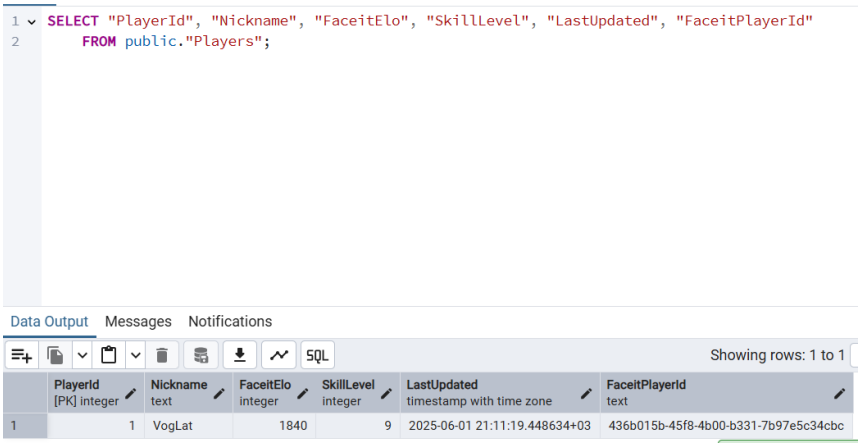


Рисунок 3.12 – Скріншот таблиці Players у PostgreSQL (pgAdmin)

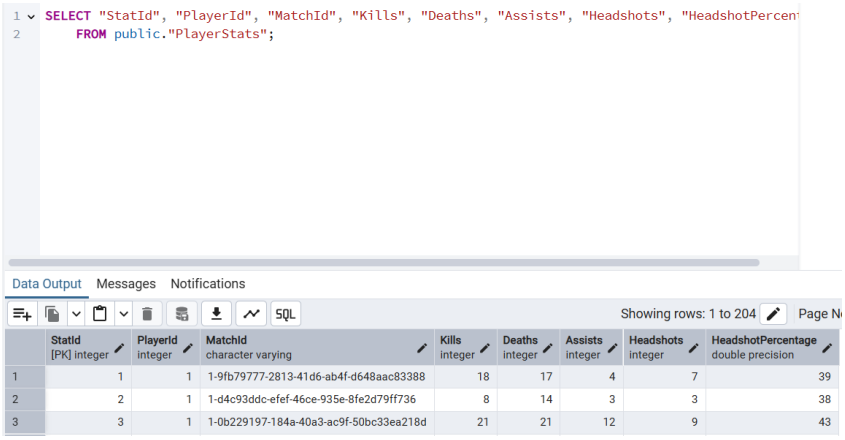


Рисунок 3.13 – Скріншот таблиці PlayerStats у PostgreSQL (pgAdmin)

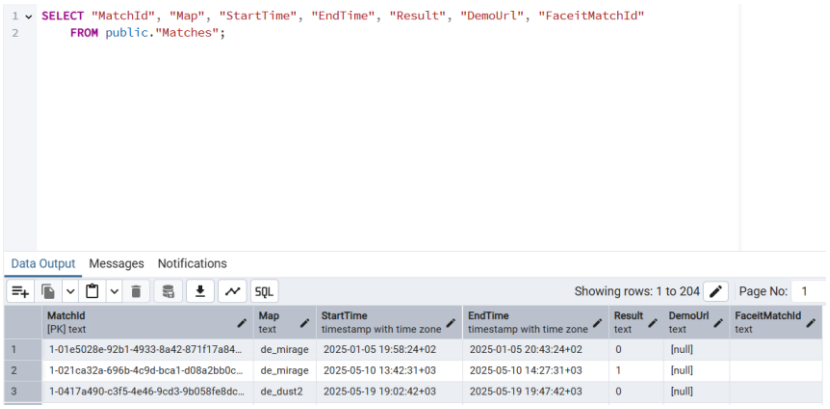


Рисунок 3.14 – Скріншот таблиціMatches у PostgreSQL (pgAdmin)

ВИСНОВКИ

Під час розробки проекту, що складається з ASP.NET Core веб-сервісу та консольного Discord-бота, було проведено детальний аналіз вимог та особливостей інтеграції із зовнішньою платформою FACEIT. Реалізація системи базується на сучасних технологіях і архітектурних підходах, що забезпечують її надійність, масштабованість та зручність використання.

Обрана архітектура з розділенням на веб-сервіс і мікросервіс у вигляді Discord-бота дозволяє ефективно розподілити функціональність, підвищити гнучкість та спростити подальший розвиток системи. Веб-сервіс відповідає за збір та збереження статистичних даних з платформи FACEIT, а бот забезпечує інтерактивну взаємодію з користувачами у середовищі Discord.

Для зберігання даних застосовано PostgreSQL у поєднанні з Entity Framework Core, що дозволяє працювати з базою у зручному об’єктно-орієнтованому стилі. Використання асинхронних операцій, обробки webhook і генерації інфографіки сприяє високій продуктивності та оперативності системи.

У процесі розробки особливу увагу було приділено безпеці, обробці помилок, а також логуванню подій, що забезпечує стабільність роботи і полегшує підтримку проекту. Тестування основних функцій підтвердило коректність роботи як веб-сервісу, так і бота.

Отже, розроблені програмні компоненти формують ефективне рішення для інтеграції ігрової статистики FACEIT у Discord, яке може бути успішно застосоване для підтримки ігрових спільнот та підвищення взаємодії користувачів. Система готова до подальшого розширення та адаптації під нові вимоги, що гарантує її актуальність у довгостроковій перспективі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Discord Developer Portal [Електронний ресурс]. – Режим доступу: https://discord.com/developers/docs

2. Elmasri R., Navathe S. Fundamentals of Database Systems. – 7th ed. – Pearson, 2016. – 1200 с.

3. Faceit API Documentation [Електронний ресурс]. – Режим доступу: https://developers.faceit.com/

4. Freeman A. Pro ASP.NET Core 6. – Apress, 2022. – 1080 с.

5. Gnatyuk S. M., Romaniuk V. Ye. Prohramuvannia movoiu C# : navch. posib. – Lviv: LNU im. Ivana Franka, 2020. – 280 с.

6. Haerder T., Reuter A. Principles of Transaction-Oriented Database Recovery. // ACM Computing Surveys. – 1983. – Vol. 15(4). – С. 287–317.

7. Lerman J., Miller R. Programming Entity Framework: Code First. – O'Reilly Media, 2020. – 240 с.

8. Medium. The What, Why, and How of a Microservices Architecture [Електронний ресурс]. – Режим доступу: https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9

9. Microsoft Learn. ASP.NET Core fundamentals [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core

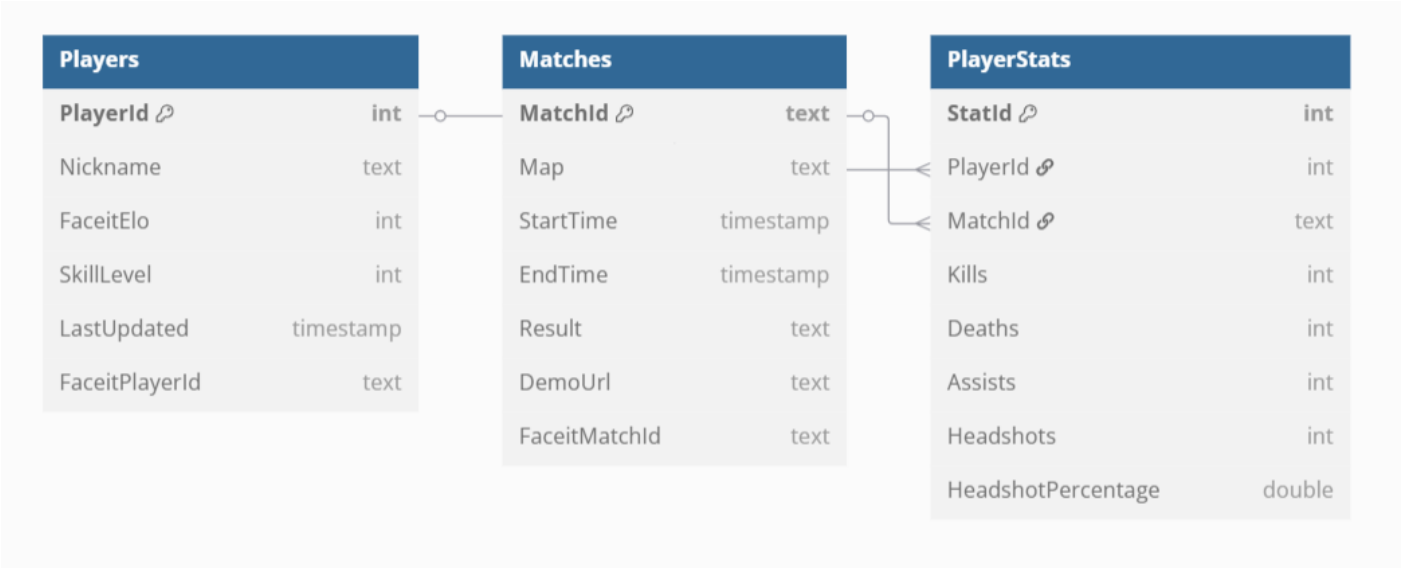
10. Nickoloff J., Kuenzli S. Docker in Action. – Manning Publications, 2019. – 350 с.

11. Stonebraker M., Kemnitz G. The POSTGRES Next-Generation Database Management System // Communications of the ACM. – 1991. – Vol. 34(10). – С. 78–92.

12. Tyshchenko A. O. Osnovy kompiuternoi hrafiky ta vizualizatsii danykh. – Kyiv: Nauk. svit, 2021. – 212 с.

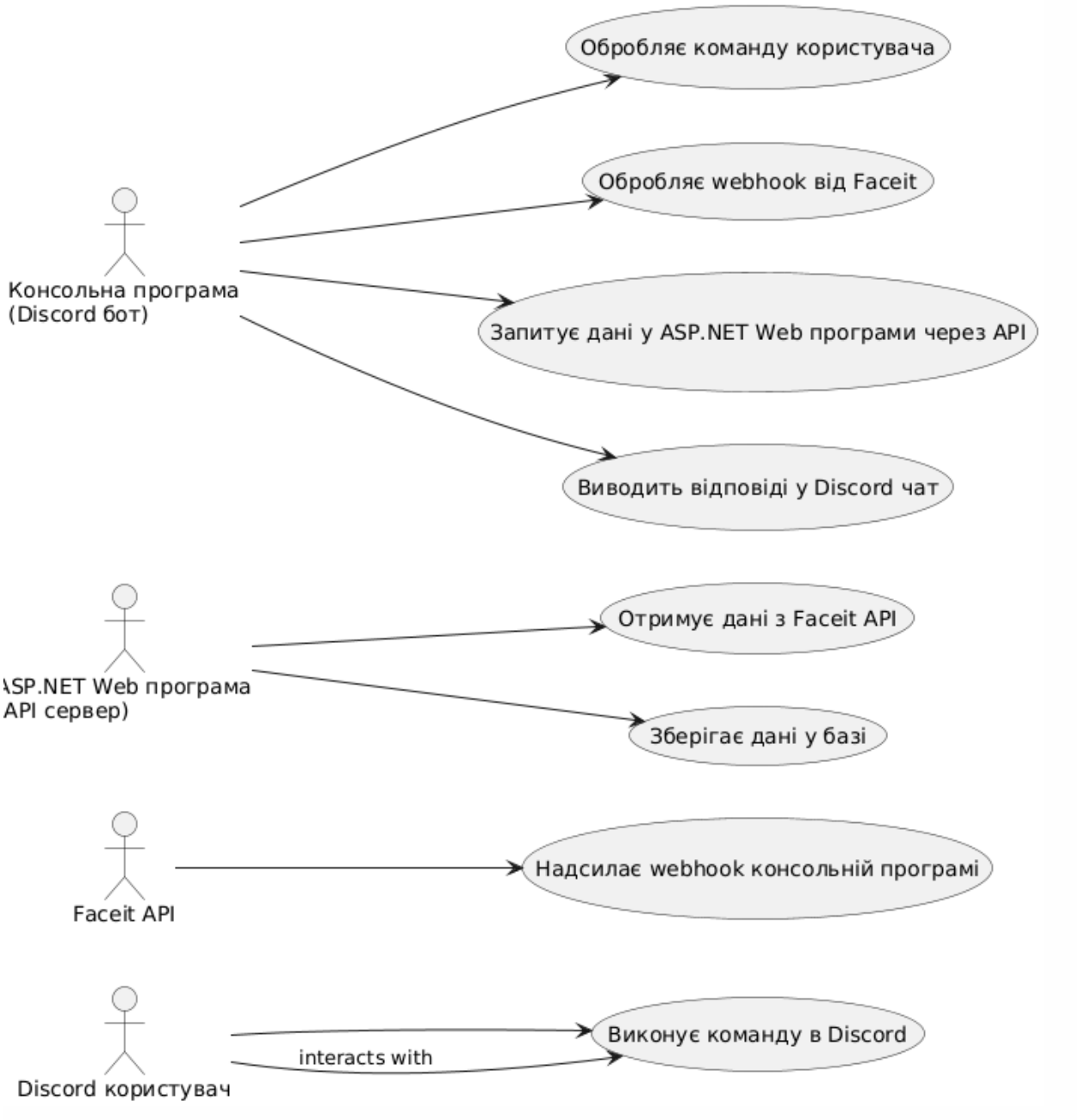
ДОДАТОК А

Entity–relationship діаграма *бази даних FaceitHelper:*



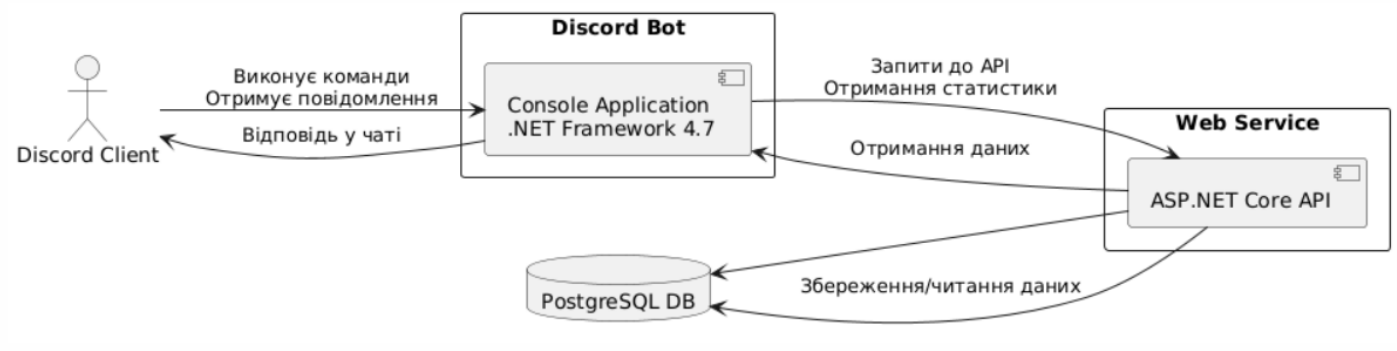
ДОДАТОК Б

Use case діаграма:



ДОДАТОК В

Архітектура FaceitHelper:



ДОДАТОК Г

Swagger файл:


```
{
  "openapi": "3.0.1",
  "info": {
    "title": "FaceitHelperApi",
    "version": "1.0"
  },
  "paths": {
    "/api/FaceitStats/player/{nickname}": {
      "get": {
        "tags": [
          "FaceitStats"
        ],
        "parameters": [
          {
            "name": "nickname",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK"
          }
        }
      }
    },
    "/api/FaceitStats/matches/{nickname}": {
      "get": {
        "tags": [
          "FaceitStats"
        ],
        "parameters": [
          {
            "name": "nickname",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK"
          }
        }
      }
    },
    "/api/FaceitStats/playerstats/{nickname}/{matchCount}": {
      "get": {
        "tags": [
          "FaceitStats"
        ],
        "parameters": [
          {
            "name": "nickname",
            "in": "path",
            "required": true,
            "schema": {
```

```
            "type": "string"
          }
        },
        {
          "name": "matchCount",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "format": "int32",
            "default": 5
          }
        }
      ],
      "responses": {
        "200": {
          "description": "OK"
        }
      }
    }
  },
  "/api/FaceitStats/avg/{nickname}/{matchCount}": {
    "get": {
      "tags": [
        "FaceitStats"
      ],
      "parameters": [
        {
          "name": "nickname",
          "in": "path",
          "required": true,
          "schema": {
            "type": "string"
          }
        },
        {
          "name": "matchCount",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "format": "int32",
            "default": 20
          }
        }
      ],
      "responses": {
        "200": {
          "description": "OK"
        }
      }
    }
  },
  "/api/FaceitStats/statdata/{nickname}/{matchCount}": {
    "get": {
      "tags": [
        "FaceitStats"
      ],
      "parameters": [
        {
          "name": "nickname",
          "in": "path",
          "required": true,
          "schema": {
            "type": "string"
```

```
          }
        },
        {
          "name": "matchCount",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "format": "int32",
            "default": 5
          }
        }
      ],
      "responses": {
        "200": {
          "description": "OK"
        }
      }
    }
  },
  "/api/FaceitStats/progress/{nickname}/{matchCount}": {
    "get": {
      "tags": [
        "FaceitStats"
      ],
      "parameters": [
        {
          "name": "nickname",
          "in": "path",
          "required": true,
          "schema": {
            "type": "string"
          }
        },
        {
          "name": "matchCount",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "format": "int32",
            "default": 10
          }
        }
      ],
      "responses": {
        "200": {
          "description": "OK"
        }
      }
    }
  },
  "/api/FaceitStats/topmatches/{nickname}/{count}": {
    "get": {
      "tags": [
        "FaceitStats"
      ],
      "parameters": [
        {
          "name": "nickname",
          "in": "path",
          "required": true,
          "schema": {
            "type": "string"
          }
```

```
        },
        {
          "name": "count",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "format": "int32",
            "default": 100
          }
        }
      ],
      "responses": {
        "200": {
          "description": "OK"
        }
      }
    }
  }
 },
 "components": { }
}
```

ДОДАТОК Д

Реалізація усіх команд Discord бота:

```
using DSharpPlus;
using DSharpPlus.Entities;
using DSharpPlus.SlashCommands;
using DSharpPlus.SlashCommands.Attributes;
using FaceitHelper.Dtos;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

namespace FaceitHelper.Commands
{
    public class FaceitCommands : ApplicationCommandModule
    {
        private readonly HttpClient _httpClient = new HttpClient();
        private readonly string _apiBaseUrl = "https://localhost:7046/api/faceitstats";

        [SlashCommand("faceitstats", "Отримати загальну статистику гравця Faceit")]
        public async Task FaceitStats(InteractionContext ctx, [Option("nickname", "Нікнейм гравця")] string nickname)
        {
            await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);
            var url = $"{_apiBaseUrl}/player/{nickname}";
            var response = await _httpClient.GetAsync(url);

            if (!response.IsSuccessStatusCode)
            {
                await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Не вдалося отримати дані
для `{nickname}`"));
                return;
            }

            var json = await response.Content.ReadAsStringAsync();
            var playerData = JsonConvert.DeserializeObject<FaceitPlayerDto>(json);

            string msg = $"🎮 **{playerData.Nickname}**\n🌟 Рівень: {playerData.Games.Csgo.SkillLevel}\n📈 ELO:
{playerData.Games.Csgo.FaceitElo}\n🆔 ID: {playerData.PlayerId}";
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent(msg));
        }

        [SlashCommand("matchstats", "Останні матчі Faceit гравця")]
        public async Task MatchStats(InteractionContext ctx, [Option("nickname", "Нікнейм гравця")] string nickname)
        {
            await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);
            var url = $"{_apiBaseUrl}/matches/{nickname}";
            var response = await _httpClient.GetAsync(url);

            if (!response.IsSuccessStatusCode)
            {
                await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Матчі для `{nickname}` не
знайдено."));
                return;
            }
```

```
        var json = await response.Content.ReadAsStringAsync();
        var wrappedMatches = JsonConvert.DeserializeObject<MatchWrapperDto[]>(json);

        int i = 0;
        foreach (var matchWrapper in wrappedMatches.Take(3))
        {
            var match = matchWrapper.Stats;
            string msg = $"📊 **{nickname} — матч {i + 1}**\n" +
                         $"🏆 Вбивства: {match.Kills}\n" +
                         $"💀 Смертей: {match.Deaths}\n" +
                         $"🤝 Асистів: {match.Assists}\n" +
                         $"🔫 Хедшотів: {match.Headshots} ({match.HeadshotsPercentage}%)\n" +
                         $"🗺️ Карта: {match.Map}\n" +
                         $"🎯 Результат: {(match.Result == "1" ? "Перемога" : "Поразка")}";

            await ctx.FollowUpAsync(new DiscordFollowupMessageBuilder().WithContent(msg));
            i++;
        }
    }

    [SlashCommand("playerstats", "Показує загальну і середню статистику гравця за останні матчі")]
    public async Task PlayerStatsAsync(InteractionContext ctx, [Option("nickname", "Нікнейм гравця на FACEIT")]
string nickname, [Option("matches", "Кількість останніх матчів")] long matchCount = 5)
    {
        await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);
        var url = $"{_apiBaseUrl}/playerstats/{nickname}/{matchCount}";
        var response = await _httpClient.GetAsync(url);

        if (!response.IsSuccessStatusCode)
        {
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Не знайдено матчів для
гравця `{nickname}`."));
            return;
        }

        var json = await response.Content.ReadAsStringAsync();
        var stats = JsonConvert.DeserializeObject<PlayerStatsDto>(json);

        string msg = $"📈 **Статистика гравця `{nickname}` за {stats.MatchCount} матч(і):**\n" +
                     $"🏆 Загальні вбивства: {stats.TotalKills} | 💀 Смертей: {stats.TotalDeaths} | 🤝 Асистів:
{stats.TotalAssists}\n" +
                     $"🔫 Хедшоти: {stats.TotalHeadshots} | 🎯 Середній HS%: {stats.AvgHeadshotPercentage:F2}%\n\n"
+
                     $"📊 **Середні значення за матч:**\n" +
                     $"• Kills: {stats.AvgKills:F1}\n" +
                     $"• Deaths: {stats.AvgDeaths:F1}\n" +
                     $"• Assists: {stats.AvgAssists:F1}\n" +
                     $"• Headshots: {stats.AvgHeadshots:F1}";

        await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent(msg));
    }

        [SlashCommand("avg", "Показує середню кількість вбивств на матч для гравця")]
        public async Task AvgKillsAsync(InteractionContext ctx, [Option("nickname", "Нікнейм гравця на FACEIT")]
    string nickname, [Option("matches", "Кількість останніх матчів")] long matchCount = 20)
        {
            await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);
            var url = $"{_apiBaseUrl}/avg/{nickname}/{matchCount}";
            var response = await _httpClient.GetAsync(url);

            if (!response.IsSuccessStatusCode)
            {
                await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Не знайдено матчів для
    гравця `{nickname}`."));
                return;
            }
```

```
            var json = await response.Content.ReadAsStringAsync();
            var avgData = JsonConvert.DeserializeObject<AvgKillsDto>(json);

            string msg = $"📊 **Середня кількість вбивств гравця `{nickname}` за {avgData.MatchesCount}
матч(і):**\n" +
                         $"🏆 {avgData.AverageKills:F2} вбивств/матч";

            await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent(msg));
        }

        [SlashCommand("statimage", "Генерує зображення зі статистикою гравця")]
        public async Task StatImageAsync(InteractionContext ctx, [Option("nickname", "Нікнейм гравця")] string
nickname, [Option("matches", "Кількість матчів")] long matchCount = 5)
        {
            await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);

            try
            {
                var url = $"{_apiBaseUrl}/statdata/{nickname}/{matchCount}";
                var response = await _httpClient.GetAsync(url);

                if (!response.IsSuccessStatusCode)
                {
                    await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent("❌ Не вдалося отримати
статистику."));
                    return;
                }

                var json = await response.Content.ReadAsStringAsync();
                var stats = JsonConvert.DeserializeObject<StatDataDto>(json);
                string html = HtmlTemplateGenerator.Build(stats.Nickname, stats.Elo, stats.AvgKills, stats.KDRatio,
stats.AvgHeadshotPercent);
                var renderer = new HtmlScreenshotService();

                using (var cts = new CancellationTokenSource(TimeSpan.FromSeconds(15)))
                {
                    var imageStreamTask = renderer.GenerateStatImageFromHtmlAsync(html);
                    var completedTask = await Task.WhenAny(imageStreamTask, Task.Delay(-1, cts.Token));
                    if (completedTask == imageStreamTask)
                    {
                        var imageStream = await imageStreamTask;
                        await ctx.EditResponseAsync(new DiscordWebhookBuilder().AddFile("stats.png", imageStream));
                    }
                    else
                    {
                        await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent("❌ Генерація зображення
зайняла забагато часу і була скасована."));
                    }
                }
            }
            catch (Exception ex)
            {
                await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Сталася помилка:
{ex.Message}"));
            }
        }

        [SlashCommand("trendgraph", "Графік статистики гравця по останніх матчах")]
        public async Task TrendGraphAsync(InteractionContext ctx, [Option("nickname", "Нікнейм гравця на
FACEIT")] string nickname, [Option("matches", "Кількість останніх матчів")] long matchCount = 10)
        {
            await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);
            var url = $"{_apiBaseUrl}/progress/{nickname}/{matchCount}";
            var response = await _httpClient.GetAsync(url);

            if (!response.IsSuccessStatusCode)
            {
                await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Не вдалося отримати дані
про матчі для `{nickname}`."));
```

```
return;
        }

        var json = await response.Content.ReadAsStringAsync();
        var progress = JsonConvert.DeserializeObject<ProgressResponse>(json);

        if (progress?.Progress == null || progress.Progress.Count == 0)
        {
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent("⚠ Дані про матчі відсутні."));
            return;
        }

        using (var imageStream = MatchTrendChartGenerator.GenerateChart(progress.Progress))
        {
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().AddFile("trend.png", imageStream));
        }
    }

    [SlashCommand("topmatchesimage", "Генерує зображення з ТОП матчами гравця")]
    public async Task TopMatchesImageAsync(InteractionContext ctx, [Option("nickname", "Нікнейм гравця")]
string nickname, [Option("matches", "Кількість матчів для аналізу")] long matchCount = 20)
    {
        await ctx.CreateResponseAsync(InteractionResponseType.DeferredChannelMessageWithSource);
        var url = $"{_apiBaseUrl}/topmatches/{nickname}/{matchCount}";
        var response = await _httpClient.GetAsync(url);

        if (!response.IsSuccessStatusCode)
        {
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Не вдалося отримати ТО
матчі для `{nickname}`."));
            return;
        }

        var json = await response.Content.ReadAsStringAsync();
        var data = JsonConvert.DeserializeObject<TopMatchesResponse>(json);
        var html = HtmlTemplateGenerator.BuildTopMatchesHtml(data.Nickname, data.TopKills, data.TopKDR);
        var renderer = new HtmlScreenshotService();

        try
        {
            var stream = await renderer.GenerateStatImageFromHtmlAsync(html);
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().AddFile("top_matches.png", stream));
        }
        catch (Exception ex)
        {
            await ctx.EditResponseAsync(new DiscordWebhookBuilder().WithContent($"❌ Помилка при генерації
зображення: {ex.Message}"));
        }
    }

    private class ProgressResponse
    {
        public string Nickname { get; set; }
        public int MatchesAnalyzed { get; set; }
        public List<MatchTrendDto> Progress { get; set; }
    }
  }
}
```