БАКАЛАВРСЬКА РОБОТА
БР.АКП-38.00.00.000 ПЗ
група АКП-23-1К
Вадим Гаврилів

2025

Міністерство освіти і науки України
Івано-Франківський національний технічний університет нафти і газу
Факультет автоматизації та енергетики
Кафедра автоматизації та комп'ютерно-інтегрованих технологій

Гаврилів Вадим Тарасович
(прізвище, ім'я, по батькові)

УДК 681.004.932
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розроблення автоматичного віртуального консультанта для месенджера
(назва роботи)
Discord

Автоматизація та комп'ютерно-інтегровані технології
(назва освітньої програми)

174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка
(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Нормоконтроль
доцент О. В. Кучмистенко
(посада) (підпис) (дата) (ініціали та прізвище)

Рецензент
доцент І. І. Чигур
(посада) (підпис) (дата) (ініціали та прізвище)

Здобувач освітнього ступеня
АКП-23-1К В. Т. Гаврилів
(шифр групи) (підпис) (дата) (ініціали та прізвище)

Науковий керівник
доцент Р. Б. Скрип'юк
(посада) (підпис) (дата) (ініціали та прізвище)

Допущено до захисту
Завідуючий кафедри
доцент А. І. Лагойда
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

(повне найменування закладу вищої освіти)

Факультет автоматизації та енергетики

Кафедра автоматизації та комп'ютерно-інтегрованих технологій

Освітній рівень перший (бакалаврський)

Спеціальність 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка

(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри АКІТ
А.І. Лагойда.
«___» _____ 20__ року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Гавриліву Вадиму Тарасовичу

(прізвище, ім'я, по батькові)

1. Тема роботи **Розроблення автоматичного віртуального консультанта для месенджера Discord.**

керівник роботи Скрип'юк Ростислав Богданович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від « 07 » травня 20 25 року № 52/8

2. Строк подання студентом роботи 13.06.2025

3. Вихідні дані до роботи: матераіали переддипломої практики, технічна література, інтернет-ресурс.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. 1. Аналіз існуючих discord-ботів і вибір платформи для Розробки. 2. Створення бази даних sqlite в середовищі vscode. 3. Програмна реалізація автоматичного віртуального консультанта для месенджера disocrd. 4. Експерементальне дослідження розробленого автоматичного віртуального консультанта для месенджера disocrd Загальні висновки. Перелік на джерела.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) 1. БР.АКП-38.00.00.001 – Функціональне середовище БД SQLite Browser. 2. БР.АКП-38.00.00.002 – Робоче вікно середовища SQLite Browser з відкритою базою даних . 3. БР.АКП-38.00.00.003 - Вікно створення на налаштування таблиць SQLite Browser. 4.БР.АКП-38.00.00.004 – Структура файлів інтерпретатора Python. 5.БР.АКП-38.00.00.005 – Вигляд класу Databace в середовищі. 6. БР.АКП-38.00.00.06 – Вигляд інтерфейсу бази даних в середовищі.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 06.11.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Термін виконання етапів роботи	Примітка
1	Аналіз існуючих discord-ботів і вибір платформи для розробки	13.05.2025	
2	Створення бази даних sqlite в середовищі vscode	23.05.2025	
3	Програмна реалізація автоматичного віртуального консультанта для месенджера disocrd	28.05.2025	
4	Експерементальне дослідження розробленого автоматичного віртуального консультанта для месенджера disocrd	07.06.2025	
5	Оформлення бакалаврської роботи	12.06.2025	

Студент _____ В. Т. Гаврилів
(підпис) (ініціали та прізвище)

Керівник роботи _____ Р. Б. Скрип’юк
(підпис) (ініціали та прізвище)

АНОТАЦІЯ

Бакалаврська робота містить: 52 сторінки, 39 рисунків, 1 таблиця, 8 посилань на літературу.

Тема: «Розроблення автоматичного віртуального консультанта для месенджера Discord».

Об'єкт дослідження: процес інформаційної взаємодії користувача з автоматизованими консультативними системами в середовищі месенджера Discord.

Предмет дослідження: методи та алгоритми розроблення віртуального консультанта на базі штучного інтелекту для інтеграції з платформою Discord.

Мета роботи полягає у створенні інтелектуального програмного агента, здатного взаємодіяти з користувачами у режимі реального часу, надаючи консультації, відповіді на часті запитання та інформаційну підтримку.

Результати роботи: у роботі розглянуто процес розроблення автоматичного віртуального консультанта для месенджера Discord. Віртуальний консультант базується на використанні штучного інтелекту, зокрема методів обробки природної мови (NLP), та інтегрується з Discord API. Також проаналізовано архітектуру системи, методи обробки запитів користувачів, алгоритми ухвалення рішень, а також засоби забезпечення ефективного та безпечного функціонування бота в чат-середовищі.

Результатом є функціональний прототип, що демонструє здатність до адаптації, контекстного аналізу та самонавчання, що дозволяє підвищити якість обслуговування спільнот Discord.

Ключові слова: продуктивність, двигун, аналіз, регулювання, критерій стійкості, моделювання, об’єкт керування, структурна схема, функція передачі.

ANNOTATION

Bachelor's thesis contains: 52 pages, 39 figures, 1 tables, 8 sources.

Topic: "Study of the dynamics of the regulation of the drilling process with electric drills under conditions of uncertainty".

Object of research: is the technological process of deepening oil and gas wells with electric drills.

Subject of research: are methods of analysis and automatic control of the process of deepening wells with electric drills.

Purpose of work: is to study the dynamics of the regulation of the drilling process of oil and gas wells with electric drills under conditions of uncertainty factors.

Within the framework of the study, a comprehensive analysis of the technological characteristics and energy indicators of electric drills was carried out, in particular, the process of wear of the cone bit and its effect on drilling efficiency was studied. The main patterns of the drilling process were analyzed. Among the optimality criteria, the criterion of maximum power on the bit and minimization of the cost of drilling the well was highlighted. The goals of process management are formulated and a set of measures for their achievement is determined, including a list of specific targeted actions, the implementation of which ensures the achievement of the set goal. The necessary resource base is also determined and a system for monitoring and controlling the implementation of measures is developed.

Mathematical models of the relationship between energy indicators and the axial load on the bit are established. An improved approach to the use of the power of a submerged electric motor in the drilling process, in particular by increasing the accuracy of the power stabilization system on its shaft.

Keywords: performance, engine, analysis, regulation, stability criterion, modeling, control object, structural diagram, transfer function.

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

БД	База даних
ОС	Операційна система
ПЗ	Програмне забезпечення
ПК	Персональний комп’ютер
ПП	Програмний продукт
FTP	File Transfer Protocol
GNU	General Public License.
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol

ВСТУП

Сучасні цифрові комунікації дедалі більше переходять у сферу інтерактивних онлайн-платформ, серед яких особливе місце займають месенджери, зокрема Discord. Цей сервіс набув широкого застосування не лише як засіб спілкування, але й як інструмент організації роботи спільнот, проведення онлайн-заходів, підтримки користувачів тощо. У зв’язку з цим зростає потреба в автоматизованих засобах комунікації, здатних забезпечити оперативне та якісне інформування учасників платформи [1÷8].

Одним із рішень є впровадження віртуальних консультантів — програмних агентів, що використовують методи штучного інтелекту для обробки звернень користувачів, генерації відповідей і виконання простих дій у рамках заданого функціоналу. Розроблення такого консультанта для Discord дозволяє автоматизувати рутинні процеси, знизити навантаження на модераторів і підвищити загальний рівень взаємодії в спільнотах.

Актуальність теми зумовлена необхідністю створення інтелектуальних рішень для гнучкої підтримки користувачів у реальному часі, а також широкими можливостями інтеграції таких рішень у популярні платформи. Метою цієї роботи є проєктування та реалізація автоматичного віртуального консультанта, який зможе ефективно функціонувати в середовищі Discord та відповідати вимогам до сучасних інформаційних систем.

1. АНАЛІЗ ІСНУЮЧИХ DISCORD-БОТІВ І ВИБІР ПЛАТФОРМИ ДЛЯ РОЗРОБКИ

Discord є однією з найпопулярніших платформ для голосового та текстового спілкування, що використовується не лише геймерами, а й освітніми, технічними, науковими та комерційними спільнотами. Однією з ключових переваг Discord є підтримка ботів — програмних агентів, які автоматизують виконання певних дій у межах серверів. Завдяки гнучкому Discord API, розробники мають можливість створювати кастомізовані боти з широким спектром функцій [2, 3].

На сьогодні існує велика кількість готових рішень, які умовно можна поділити на кілька категорій:

– модераційні боти (наприклад, Dyno, MEE6): забезпечують автоматичний контроль за поведінкою учасників, ведення журналів подій, видачу попереджень, блокування тощо;

– інформаційні боти (Carl-bot, Tatsumaki): надають користувачам допоміжну інформацію, дозволяють створювати команди за шаблоном, інтегрувати бази знань, повідомлення про події;

– навчальні або спеціалізовані боти: реалізують вузькоспеціалізовані функції, зокрема надання консультацій, підказок, інтеграцію з зовнішніми сервісами (Google, GitHub, Notion тощо).

Незважаючи на велику кількість ботів, існує потреба в розробці інтелектуальних віртуальних консультантів, які можуть забезпечувати персоніфіковану підтримку, вести діалог на основі природної мови та навчатися з досвіду спілкування.

1.1 Огляд технологій для розробки Discord-бота

Найпоширенішими мовами програмування для створення Discord-ботів є [2]:

– Python (з бібліотекою discord.py або її форками): зручна синтаксична структура, велика кількість готових рішень, сумісність з NLP-бібліотеками (transformers, nltk, spaCy).

– JavaScript / Node.js (з бібліотекою discord.js): висока продуктивність, підтримка асинхронних викликів, інтеграція з іншими веб-сервісами.

– Go, Java, C#: менш популярні, але також використовуються в більш специфічних проектах.

У цьому проєкті для реалізації обрано мову Python разом із бібліотекою discord.py (або її форком py-cord), що дозволяє реалізувати як базові, так і розширені функції, включно з інтеграцією моделей штучного інтелекту.

Мову програмування Python я обрав не випадково. Це одна з найзручніших і водночас потужних мов, яка ідеально підходить для розробки Discord-ботів. Її головна перевага — це простий, зрозумілий синтаксис, який не перевантажує зайвими дужками, складною структурою чи шаблонними конструкціями. Саме через це Python часто обирають як новачки, так і досвідчені розробники [7].

Python з’явився наприкінці 80-х років, що дозволило писати менше коду, але при цьому залишатиметься максимально виразною. Назва Python не має нічого спільного з плазунами — вона походить від улюбленого комедійного шоу автора: *Monty Python's Flying Circus*.

Сьогодні Python підтримує кілька стилів програмування: процедурне, об’єктно-орієнтоване, функціональне. А це значить, що розробник може обрати той підхід, який найбільше відповідає потребам конкретного проєкту. Для написання ботів, наприклад, зручно поєднувати класи з асинхронними функціями, що Python дозволяє без проблем.

Ще одна сильна сторона мови — величезна екосистема готових бібліотек. Потрібен інтерфейс до бази даних? Будь ласка. Хочеш реалізувати нейромережу або телеграм-бота? Усі інструменти вже давно створені. Саме завдяки бібліотекам discord.py, disnake та багатьом іншим Python став одним із основних виборів для створення чат-ботів [7].

Також варто зазначити, що Python працює практично скрізь: на Windows, Linux, macOS. Це дозволяє писати код один раз, а потім запускати його на будь-якому пристрої без значних змін. До того ж, оновлення мови постійно вдосконалюють синтаксис, додають корисні інструменти для обробки помилок, керування пам'яттю й асинхронного програмування.

Загалом, Python — це мова, яка не обмежує. Вона надає свободу творити, дає змогу швидко протестувати ідеї та реалізувати навіть складні проєкти з мінімальними витратами часу на базову інфраструктуру. Саме тому вона ідеально підходить для реалізації мого універсального Discord-бота.

1.2 Створення та еволюція Python

Мова програмування Python пройшла довгий шлях — від простого хобі-розроблення до одного з найвпливовіших інструментів у сучасному програмуванні. Все почалося ще в 1989 році, коли Гвідо ван Россум вирішив створити щось простіше і зручніше, ніж існуючі тоді мови. Його метою було дати розробникам інструмент, який не перевантажений зайвою складністю, але дозволяє вирішувати серйозні завдання [7].

Перший офіційний реліз Python з'явився на початку 1990-х. Уже тоді мова виділялась простим синтаксисом, який був зрозумілий навіть новачкам, і великим набором вбудованих можливостей. Завдяки цьому Python швидко почали використовувати у сфері освіти, а згодом — у промислових і наукових проєктах.

Протягом наступних десятиліть мова продовжувала розвиватися. Вихід Python 2 забезпечив стабільну основу для тисяч проєктів, однак із часом потреба в оновленні стала очевидною. Тому в 2008 році з'явився Python 3, який привніс чимало змін — нову систему роботи з текстом, удосконалення в обробці помилок, асинхронність та багато іншого. Підтримка другої версії остаточно завершилася у 2020 році, і тепер весь фокус зосереджено на Python 3 [7].

Із кожною новою версією Python стає потужнішим. Наприклад, у версії 3.10, яка була актуальною на момент створення цього проєкту, з'явилися конструкції match-case, покращене підказування типів та зручні інструменти для

асинхронного коду. Всі ці можливості безпосередньо впливають на зручність і швидкість розробки ботів та інших застосунків.

Ще один важливий аспект — це підтримка з боку спільноти. Python активно використовують такі гіганти, як Google, Dropbox, NASA та багато інших. А це значить, що під цю мову створюються нові фреймворки, бібліотеки й рішення майже для будь-якої сфери: від автоматизації до штучного інтелекту.

На сьогодні Python — це вже не просто мова, а цілий екосвіт. Розробник із базовими знаннями може за лічені дні зібрати працюючий прототип, тоді як досвідчений програміст — побудувати масштабну систему з мікросервісами, базами даних і веб-інтерфейсом.

Саме завдяки такій еволюції Python став незамінним у розробці мого Discord-бота — від зручності у старті до гнучкості при масштабуванні функціоналу [7].

1.3 Переваги мови програмування Python

Як тільки я почав працювати над проєктом Discord-бота, вибір мови програмування був очевидним — Python. Причина проста: ця мова надає саме той баланс між простотою та потужністю, якого так не вистачає в багатьох інших технологіях. Але спробую пояснити більш конкретно, чому саме Python [7].

Найперше — читабельність коду. У Python практично немає «візуального шуму» — зайвих дужок, фігурних конструкцій чи заплутаних оголошень. Код виглядає майже як звичайні речення англійською мовою. Це дозволяє швидше його писати, легше підтримувати й ефективно працювати навіть у команді, де кожен читає і змінює чужий код.

Друге — універсальність. Python підтримує різні парадигми програмування: хочеш — пиши функціонально, хочеш — об’єктно. Це дуже зручно, коли треба одночасно працювати з базами даних, обробкою команд користувача і логікою бота — усе це гармонійно вміщується в одному коді.

Ще один важливий аргумент — велика кількість бібліотек. Якщо потрібно реалізувати щось нестандартне (наприклад, інтеграцію з Discord, штучний

інтелект, обробку зображень чи підключення до API), майже напевно вже існує готове рішення або принаймні стартовий пакет, який значно економить час. Для мого проєкту, наприклад, бібліотеки disnake, sqlite3, dotenv і psutil стали ключовими.

Не менш важливо і те, що Python легко запускається на різних операційних системах — Windows, Linux, macOS. А це означає, що не треба щоразу адаптовувати код під конкретне середовище. Написав один раз — і воно працює [7].

Нарешті, Python має надзвичайно активну спільноту. Якщо з'являється помилка або виникає питання, зазвичай достатньо кількох хвилин пошуку, щоб знайти відповідь. Форумів, документації та відкритих прикладів — безліч.

Усе це робить Python ідеальним вибором для проєктів, де важлива швидкість реалізації, зрозумілий код і можливість легко розширювати функціонал. Саме такою і є моя система — бот, який повинен бути зрозумілим не лише комп'ютеру, а й людині.

1.4 Аргументування вибору платформи Visual Studio Code

На етапі вибору середовища розробки я перепробував кілька різних редакторів, але врешті-решт зупинився на Visual Studio Code — і жодного разу про це не пошкодував. Причин для цього вибору було кілька, і всі вони практичні.

Перш за все, VS Code — безкоштовний. Це справді важливо, особливо для студентських або індивідуальних проєктів, де бюджет обмежений. Незважаючи на те, що редактор не потребує ліцензії, він нічим не поступається дорогим середовищам розробки [4, 5].

По-друге, VS Code має вбудовану підтримку Python. З перших хвилин після встановлення редактор пропонує підсвічування синтаксису, автодоповнення, інтеграцію з терміналом, відлагодження і навіть підказки помилок у коді. Все працює «з коробки» або встановлюється за кілька кліків.

Ще одна перевага — гнучкість і розширюваність. Якщо чогось не вистачає — просто встановлюєш потрібне розширення. Наприклад, у моєму проєкті я

використовував додатки для роботи з Git, інтеграцію з SQLite, середовище для тестування коду, а також Discord Rich Presence — розширення, яке показує статус розробника прямо у Discord [3].

Також варто відзначити високу продуктивність VS Code. Навіть на середньому за характеристиками комп'ютері редактор працює стабільно, без затримок. І це при тому, що водночас можуть бути відкриті десятки файлів, кілька терміналів і кілька активних розширень.

Для мого проєкту — розробки бота для Discord — середовище виявилося особливо зручним. Я мав змогу працювати з кодом, базами даних, документацією та тестовими консолями, не виходячи з однієї програми. І це суттєво прискорило розробку.

Загалом, Visual Studio Code — це поєднання простоти, функціональності та широких можливостей для налаштування. Саме тому я обрав саме його як основний інструмент для створення універсального Discord-бота [3,6].

Вимсновок до розділу

Таким чином, у результаті аналізу сучасних Discord-ботів і можливих платформ для розробки було визначено доцільність використання Python, бібліотеки discord.py та інтеграції з OpenAI API для побудови інтелектуального віртуального консультанта.

2 СТВОРЕННЯ БАЗИ ДАНИХ SQLITE В СЕРЕДОВИЩІ VSCODE

2.1 Опис роботи програмного забезпечення Visual Studio Code

Під час створення Discord-бота мені було важливо мати зручне середовище, яке не тільки підтримує Python, але й дозволяє швидко писати, тестувати та відлагоджувати код. Саме тому Visual Studio Code (VS Code) став для мене основним робочим інструментом [5, 6].

Цей редактор має все необхідне для комфортної роботи з ботами. Зокрема, підтримка Python реалізована через офіційне розширення, яке встановлюється за кілька секунд. Після цього доступні функції автодоповнення, підсвічування синтаксису, перевірка помилок у реальному часі — і це справді полегшує життя.

Однією з найбільш корисних функцій стала інтеграція з терміналом. Я міг запускати бот прямо в редакторі, не перемикаючись на зовнішні консолі. Це зручно, особливо коли хочеться швидко протестувати нову функцію або перевірити, чи підключена база даних.

VS Code також підтримує систему контролю версій Git, що стало ще одним плюсом. Я мав змогу фіксувати зміни, відстежувати історію правок і за потреби повертатися до попередніх версій файлів. Це особливо важливо, коли проєкт швидко розростається, і виникає потреба зберігати стабільні етапи розробки.

Ще один момент — можливість встановлення розширень для роботи з базами даних. У моєму випадку це було особливо актуально, бо бот працює з SQLite. Завдяки додатку SQLite Viewer я мав змогу переглядати вміст бази даних, перевіряти таблиці, переглядати записи — і все це безпосередньо з редактора.

Крім того, існує розширення Discord Rich Presence, яке дозволяє показувати в Discord, над чим ти саме працюєш у VS Code. Це виглядає дрібницею, але створює атмосферу активної розробки, особливо коли в команді кілька людей і кожен бачить, чим займається інший.

Загалом, використання VS Code дало мені змогу зосередитись на розробці бота, а не на боротьбі з інструментами. Його простота, швидкість і підтримка розширень зробили розробку набагато приємнішою та ефективнішою.

2.2 Вибір середовища проєктування бази даних

Розробка програми починається з декількох важливих етапів:

- вибір відповідної бази даних та середовища;
- вибір функціонального сервера.

Для реалізації дипломного проєкту було обрано середовище проєктування баз даних SQLite Browser–інструмент, що інтегрує проєктування, моделювання, створення й експлуатацію БД в єдиному безкоштовному середовищі для системи баз даних SQLite, головне вікно якого зображено на рис. 2.1.

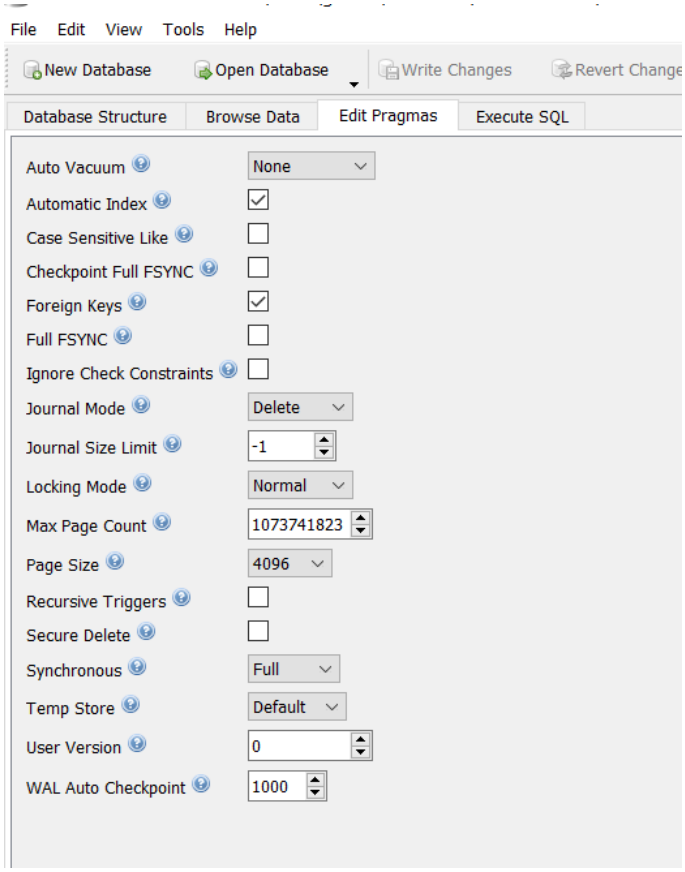


Рисунок 2.1 – Функціональне середовище БД SQLite Browser [6]

Вигляд робочого вікна SQLite Browser зображено на рис. 2.2.

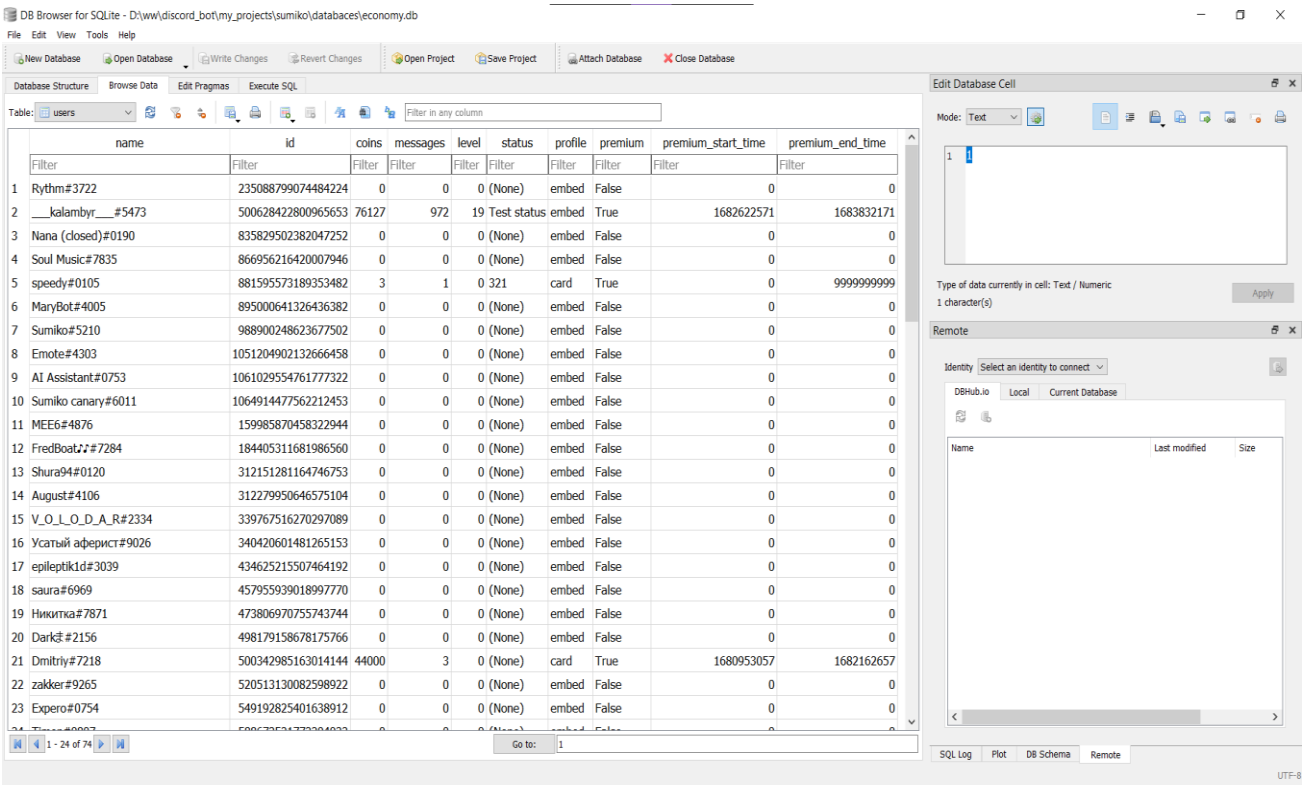


Рисунок 2.2 – Вигляд робочого вікна середовища SQLite Browser з відкритою базою даних

Правильний вибір та хороше налаштування з підключенням гарантує безперебійну роботу з базою даних, та безпомилкову компіляцію коду. А це в свою чергу забезпечить надійність та безпеку даних внесених до БД. Тому, перш за все, потрібно подбати про правильне налаштування конфігурації БД. Вигляд вікна створення та налаштування таблиць зображено на рис. 2.3 [6].

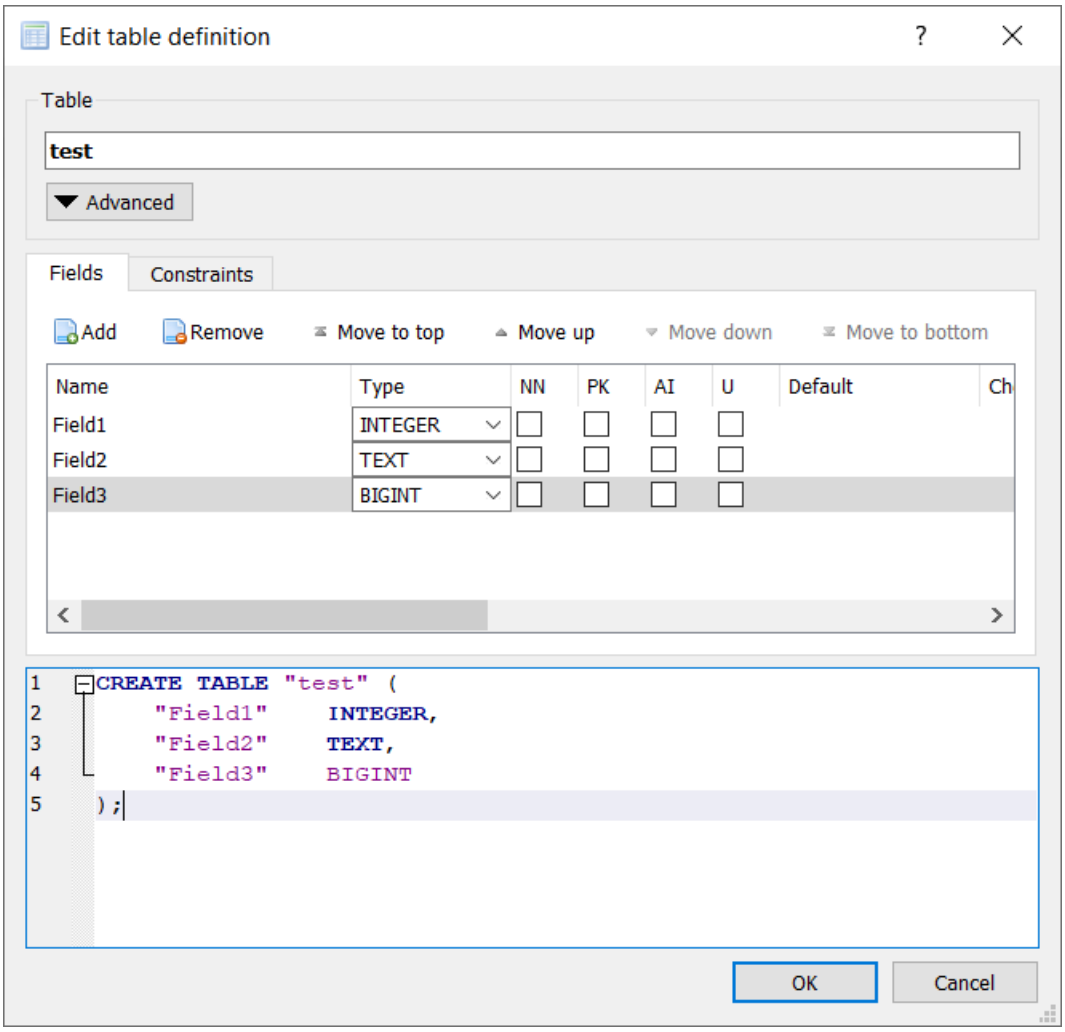


Рисунок 2.3 – Вікно створення на налаштування таблиць SQLite Browser

Слід зауважити, що налаштуванню піддаються не тільки сама БД а й конектори. Це драйвери за допомогою яких відбувається безпосереднє підключення до серверів SQL баз даних. І їх редагування саме в середовищі SQLite Browser, робить це середовище ще більш зручним у використанні [6,8].

Вигляд вікна встановлення SQLite Browser 2.4.

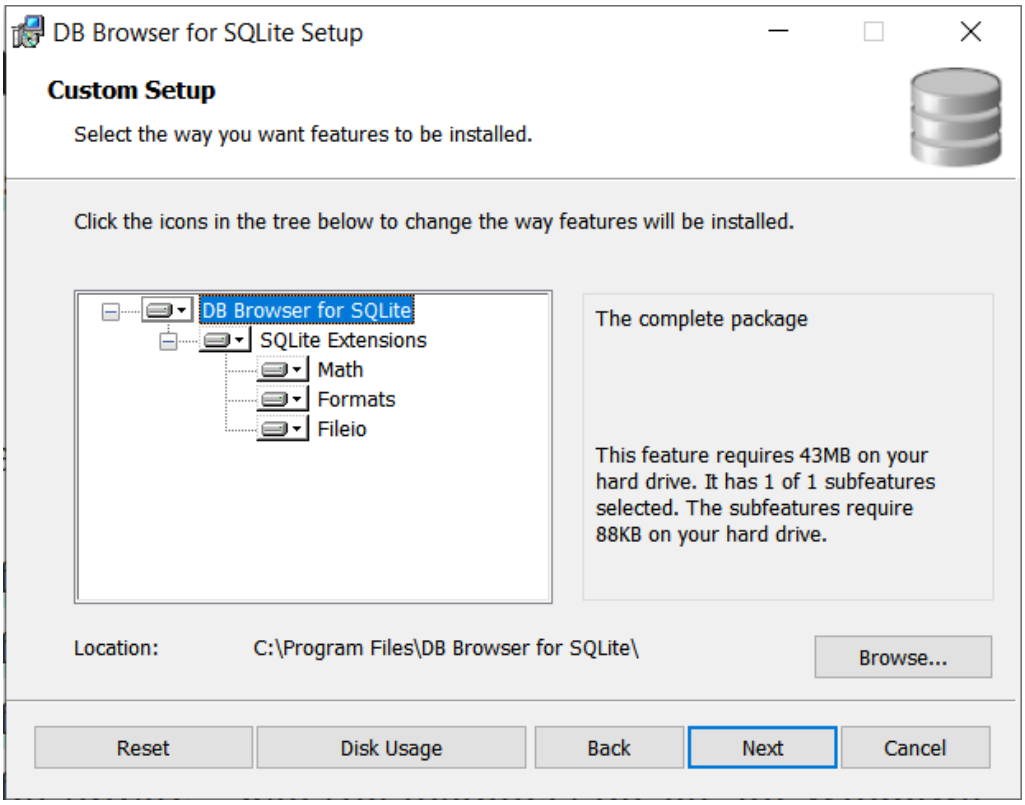


Рисунок 2.4 – Вікно середовища розробки SQLite Browser

Все налаштування відбувається доволі елементарно. Це одна з багатьох причин використання саме SQLite Browser. Легке створення таблиць, легке написання коду для стовпців та їх налаштувань, і наступне підключення до самого сервера – найбільша заслуга VS Code.

2.3 Характеристики і вимоги для встановлення Visual Studio Code

Visual Studio Code (VS Code) — це безкоштовне кросплатформне середовище розробки з відкритим кодом, яке підтримує велику кількість мов програмування та розширень. Завдяки своїй легкості, зручному інтерфейсу та інтеграції з інструментами розробника, VS Code є одним із найпопулярніших інструментів для створення Discord-ботів на Python [6,8].

Основні вимоги до характеристик ПК для встановлення Visual Studio Code наведено в табл. 2.1.

Таблиця 2.1 – Основні вимоги до характеристик ПК

Операційна система	Windows	OS X	Linux
Версія ОС	Microsoft Windows 10/8/7/Vista/2003/XP (включаючи 64-bit)	Mac OS X 10.5 або вище, аж до MacOS 10.12 (Sierra)	GNOME або KDE стільниця
Оперативна пам'ять	1 GB ОЗП мінімум, 2 GB ОЗП рекомендується		
Простір на диску	300 MB на твердому диску + принаймні 1 GB для кешування		
Роздільна здатність	1024×768 мінімальна роздільна здатність		

Вигляд вікна вибору версії зображено на рис. 2.5.

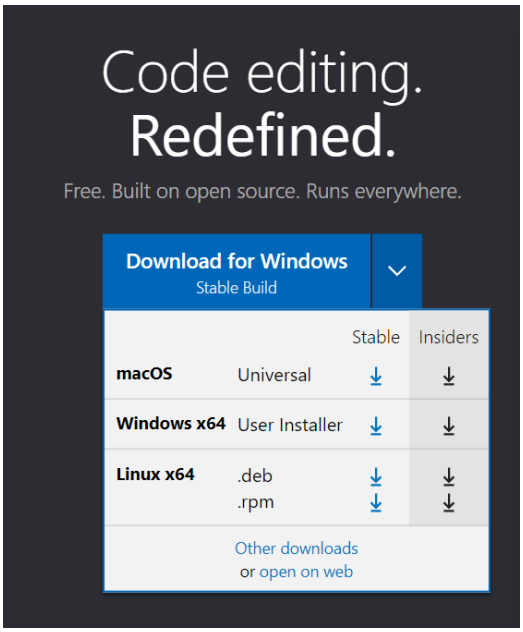


Рисунок 2.5 – Вікно вибору версії продукту

2.4.1 Завантаження та встановлення

Перейдіть на офіційний сайт Visual Studio Code: https://code.visualstudio.com/

Оберіть версію для вашої операційної системи (Windows, macOS або Linux). Завантажте інсталяційний файл та запустіть його. У вікні інсталятора рекомендується:

– обрати опцію "Add to PATH" — це дозволить запускати VS Code з командного рядка;

– позначити "Register Code as an editor for supported file types".

За потреби, дозволити створення ярлика на робочому столі.

2.4.2 Інсталяція розширень для Python

Після встановлення Visual Studio Code, необхідно налаштувати його для роботи з Python: відкрийте програму VS Code; перейдіть до розділу Extensions (або натисніть Ctrl+Shift+X). У пошуку введіть Python і встановіть розширення від Microsoft. За бажанням також рекомендується встановити:

– Pylance — для покращеного автодоповнення та аналізу коду.

– Jupyter — якщо планується працювати з інтерактивними ноутбуками.

– Black або autopep8 — для автоматичного форматування коду.

– Prettier — універсальний форматер.

– .env — для роботи з конфігураційними змінними середовища.

2.4.3 Налаштування інтерпретатора Python

Якщо Python ще не встановлено, завантажте його з офіційного сайту: https://www.python.org/downloads/

У VS Code натисніть Ctrl+Shift+P і виберіть команду Python: Select Interpreter. Оберіть потрібну версію Python, встановлену у вашій системі.

Для коректної роботи з Discord-ботом важливо, щоб усі залежності встановлювалися у віртуальне середовище. Для цього доцільно створити окреме середовище через: python -m venv venv та активувати його у VS Code.

Рекомендується організувати структуру файлів інтерпретатора Python наступним чином (рис. 2.6):

```bash
/discord-bot/
│
├── main.                # основний файл бота
├── requirements.txt     # список залежностей
├── .env                 # файл конфігураційних змінних
├── /modules/            # додаткові модулі
└── README.md
```

Рисунок 2. 6 – Структура файлів інтерпретатора Python [7]

Щоб спростити запуск бота, можна налаштувати launch.json у папці .vscode (рис. 2.7):

```json
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Запустити бота",
      "type": "python",
      "request": "launch",
      "program": "${workspaceFolder}/main.py",
      "console": "integratedTerminal"
    }
  ]
```

Рисунок 2. 6 – Налаштування launch.json

Таким чином, правильне налаштування Visual Studio Code значно спрощує процес розробки, тестування та масштабування Discord-бота. Інтеграція з Python, зручність роботи з розширеннями та можливість використання віртуальних

середовищ робить VS Code оптимальним вибором для реалізації віртуального консультанта.

Висновок до розділу

Для створення системи було обрано мову програмування Python разом із бібліотекою discord.py та інтеграцією з OpenAI API, що дозволяє реалізувати обробку природної мови, генерацію змістовних відповідей і ведення контекстного діалогу з користувачами.

Також налаштовано зручне середовище розробки на базі Visual Studio Code, що забезпечує ефективну роботу з кодом, віртуальними середовищами та системами контролю версій.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ АВТОМАТИЧНОГО ВІРТУАЛЬНОГО КОНСУЛЬТАНТА ДЛЯ МЕСЕНДЖЕРА DISOCRD

3.1 Розробка модулів та класів для роботи з Discord API

Першим кроком у створенні мого Discord-бота стало проєктування основної структури проєкту. Я розбив функціональність бота на окремі логічні блоки – модулі та класи, щоб спростити підтримку й розширення коду в майбутньому. Це дозволило працювати з проєктом гнучко: кожна частина відповідає за свою конкретну ділянку, і всі частини взаємодіють між собою через чітко визначені інтерфейси (рис 3.1) [2,3,5].

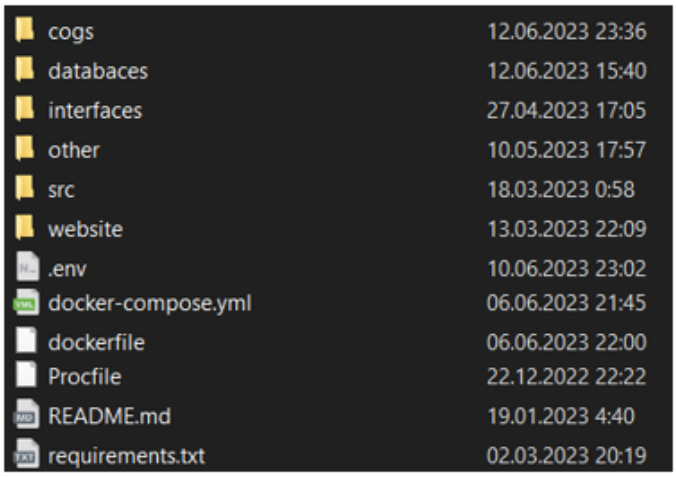


Рисунок 3.1 – Структура проєкту

Також у проєкті присутні папки cogs, databaces, interfaces і src, де зберігаються окремі модулі.

Модуль databace.py відповідає за роботу з базою даних. У ньому реалізовано запис нових користувачів, зчитування існуючих, перевірку, чи вже є певний ID у базі, і додавання нових записів у разі потреби. Уся ця логіка автоматизована, тому бот не дублює дані й не створює зайвих рядків.

Код запису нових користувачів наведений на рис. 3.2.

```python
# add new member
@commands.Cog.listener()
async def on_member_join(self, member: disnake.Member):
    if cursor.execute(f"SELECT id FROM users WHERE id = {member.id}").fetchone() is None:
        cursor.execute(f"INSERT INTO users VALUES ('{member}', {member.id}, 0, 0, 0, '(None)', 'embed', 'False', 0, 0)")
    if cursor.execute(f"SELECT id FROM items WHERE id = {member.id}").fetchone() is None:
        cursor.execute(f"INSERT INTO items VALUES ('{member}', {member.id}, 'None', 0, 0)")
    else:
        pass
    connection.commit()
```

Рисунок 3.3 – Перевірка та запис нових користувачів в БД

Окремо варто згадати модуль events.py. Саме тут прописані дії бота на такі події, як приєднання до сервера. Наприклад, коли бот потрапляє на нову гільдію, він автоматично надсилає її власнику привітальне повідомлення з подякою та посиланнями на сайт проєкту та канал підтримки [8].

Код обробки приєднань наведений на рис. 3.4.

```python
# join to guild
@commands.Cog.listener()
async def on_guild_join(self, guild: disnake.Guild):
    view = View()
    button_website = Button(label='Website', url=settings['WEBSITE_URL'], style=disnake.ButtonStyle.url)
    button_support = Button(label='Support server', url=settings['GUILD_URL'], style=disnake.ButtonStyle.url)

    view.add_item(button_website)
    view.add_item(button_support)
    try:
        await guild.owner.send('**Thanks for the invitation!**\nAt the moment, I can be configured using comma
    except Exception as e:
        print(f'[ERROR] {__name__}: {e}')

    embed = disnake.Embed(title=':white_check_mark: Join :white_check_mark:', description=f'Total guilds: **{le
    embed.add_field(name='• Name', value=f'```{guild.name}```', inline=False)
    embed.add_field(name='• ID', value=f'```{guild.id}```', inline=True)
    embed.add_field(name='• Members', value=f'```Total: {len(guild.members)} members```', inline=True)
    await self.client.get_channel(settings['LOGS']).send(embed=embed)
```

Рисунок 3.4 – Обробка приєдняння бота

Загалом, кожен модуль у проєкті виконує окрему функцію, і всі разом вони утворюють логічну та стабільну систему. Такий підхід дозволяє легко додавати нові можливості, не порушуючи вже наявну структуру, а також прискорює процес розробки та налагодження бота.

Код інших модулів представлений в додатку А.

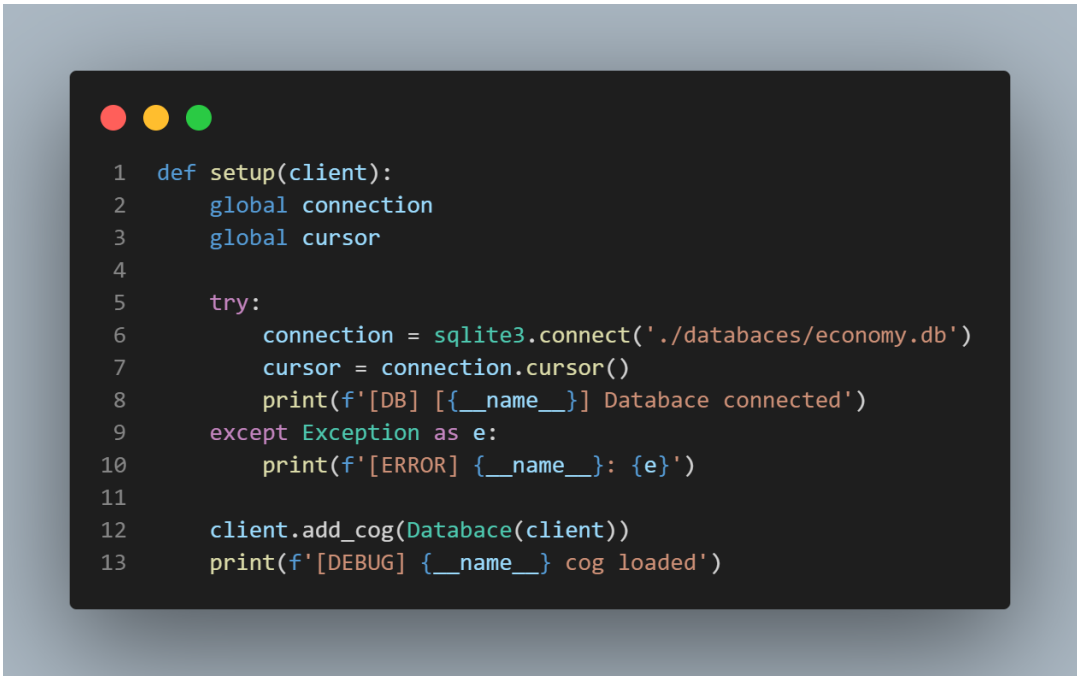
3.2 Опис класів взаємодії з БД та їх методів

Щоб зберігати й обробляти дані користувачів, я створив окремі класи, які забезпечують зручну взаємодію між Discord-ботом і базою даних. Це дозволило уникнути дублювання коду та спростити логіку роботи з інформацією.

Основним елементом тут став клас Database, який відповідає за доступ до таблиць, їх створення, оновлення записів і обробку подій. Він використовує об’єкт DbInterface, що виконує роль “посередника” між Python і SQLite. Саме через нього відбувається увесь обмін даними [8].

```python
import disnake
from disnake.ext import commands

import sqlite3
from config import settings
from interfaces.db_interface import DbInterface


class Database(commands.Cog):

    def __init__(self, client):
        self.client = client
        self.db = DbInterface()


    # create tables
    @commands.Cog.listener()
    async def on_ready(self):
        cursor.execute("""CREATE TABLE IF NOT EXISTS users (
            name TEXT,
            id INT,
            coins BIGINT,
            messages INT,
            level INT,
            status TEXT,
            profile TEXT,
            premium TEXT,
            premium_start_time INT,
            premium_end_time INT
        )""")
        cursor.execute("""CREATE TABLE IF NOT EXISTS items (
            name TEXT,
            id INT,
            item TEXT,
            reward_from INT,
            reward_to INT
        )""")
        connection.commit()

        for guild in self.client.guilds:
            for member in guild.members:
                if cursor.execute(f"SELECT id FROM users WHERE id = {member.id}").fetchone() is None:
                    if member.id == settings['OWNER_ID']:
                        cursor.execute(f"INSERT INTO users VALUES ('{member}', {member.id}, 0, 0, 0, '(None)', 'embed', 'True', 0, 9999999999)")
                    else:
                        cursor.execute(f"INSERT INTO users VALUES ('{member}', {member.id}, 0, 0, 0, '(None)', 'embed', 'False', 0, 0)")

                if cursor.execute(f"SELECT id FROM items WHERE id = {member.id}").fetchone() is None:
                    cursor.execute(f"INSERT INTO items VALUES ('{member}', {member.id}, 'None', 0, 0)")
                else:
                    pass
        connection.commit()
```

Рисунок 3.6 – Вигляд класу Databace в середовищі



```python
def setup(client):
    global connection
    global cursor

    try:
        connection = sqlite3.connect('./databaces/economy.db')
        cursor = connection.cursor()
        print(f'[DB] [{__name__}] Database connected')
    except Exception as e:
        print(f'[ERROR] {__name__}: {e}')

    client.add_cog(Database(client))
    print(f'[DEBUG] {__name__} cog loaded')
```

Рисунок 3.7 – Вигляд коду для завантаження класу Databace

Структура класу Database [2, 8]

- __init__ — конструктор класу. Він приймає параметр client і ініціалізує підключення до бази даних. Через DbInterface() створюється об’єкт для роботи з SQL-запитами.
- on_ready — метод-слухач, який спрацьовує при запуску бота. Тут автоматично створюються дві таблиці, якщо вони ще не існують:

  –"users" — для зберігання ID користувача, його рівня, статусу, кількості повідомлень, балансу та іншого;

  – "items" — для фіксації предметів, які може мати користувач, та діапазонів нагород.
- on_guild_join – ще один метод-слухач, що спрацьовує, коли бот приєднується до нового сервера. Він перебирає всіх учасників гільдії й перевіряє, чи вже є їхні дані в базі. Якщо ні — створює нові записи з базовими параметрами. Таким чином, база одразу синхронізується з новим середовищем.
- on_message — цей метод обробляє кожне повідомлення, яке надсилається на сервері. Якщо повідомлення не від бота, то кількість повідомлень

користувача збільшується на 1. Це робиться через SQL-запит, який викликається методом data_edit.

- setup — метод, який підключає клас Database до клієнта Discord. Він також ініціалізує з'єднання з базою через SQLite та забезпечує завантаження класу як Cog.

Усі запити до бази реалізовані через об’єкти connection і cursor. Завдяки чітко розділеній логіці, кожен метод відповідає за свою частину: або створення, або оновлення, або перевірку.

Що важливо — весь цей механізм працює автоматично, без потреби втручатися вручну при кожному новому повідомленні чи приєднанні користувача. Це дозволяє зосередитися на розвитку функціоналу, а не на технічних деталях збереження даних.

Такий підхід дав змогу створити гнучку систему для зберігання інформації, яка легко масштабується й адаптується під будь-яку кількість серверів і користувачів.

3.3 Розробка алгоритмів та методів для реалізації функцій бота

На цьому етапі я почав реалізовувати функції, які забезпечують "живу" роботу Discord-бота — обробку команд, управління модулями, обробку помилок та інші інтерактивні можливості. Для цього була створена низка методів і класів, кожен з яких відповідає за окрему частину логіки.

Щоб керувати модулями (cogs) під час запуску або оновлення бота, я реалізував клас Loader, у якому розмістив методи:

- **unload_cogs()** — проходить по всіх .py-файлах у папці ./cogs і вивантажує відповідні модулі з пам’яті за допомогою client.unload_extension().
- **load_cogs()** — виконує протилежну дію — завантажує всі наявні модулі через client.load_extension().

Ці методи допомагають керувати ботом без перезапуску всієї програми. Якщо потрібно внести зміни — достатньо просто перевантажити ког.

```python
class Loader:

    # unload cogs
    def unload_cogs(self):
        try:
            for filename in os.listdir('./cogs'):
                if filename.endswith('.py'):
                    client.unload_extension(f'cogs.{filename[:-3]}')

            print('[UNLOAD] Cogs unloaded')
        except Exception as e:
            print(f'[ERROR] {__name__}: {e}')


    # load cogs
    def load_cogs(self):
        try:
            for filename in os.listdir('cogs'):
                if filename.endswith('.py'):
                    client.load_extension(f'cogs.{filename[:-3]}')
            print('[LOAD] Cogs loaded')
        except Exception as e:
            print(f'[ERROR] {__name__}: {e}')
```

Рисунок 3.8 – Вигляд класу Loader в середовищі

Команда /reload

Команду /reload я реалізував окремо – вона дозволяє адміністраторам вручну оновлювати всі модулі, не перезапускаючи бота. Команда перевіряє, чи користувач має права адміністратора, і лише після цього виконує перевантаження когів. Це зручно для тестування або швидких змін без простоїв.

Метод reload_cogs

Цей метод працює у парі з Loader. Він викликає unload_cogs(), а потім — load_cogs() для перезавантаження всіх модулів. Після цього обчислюється час виконання команди і відправляється підтвердження у вигляді вбудованого повідомлення (Embed), що все пройшло успішно.

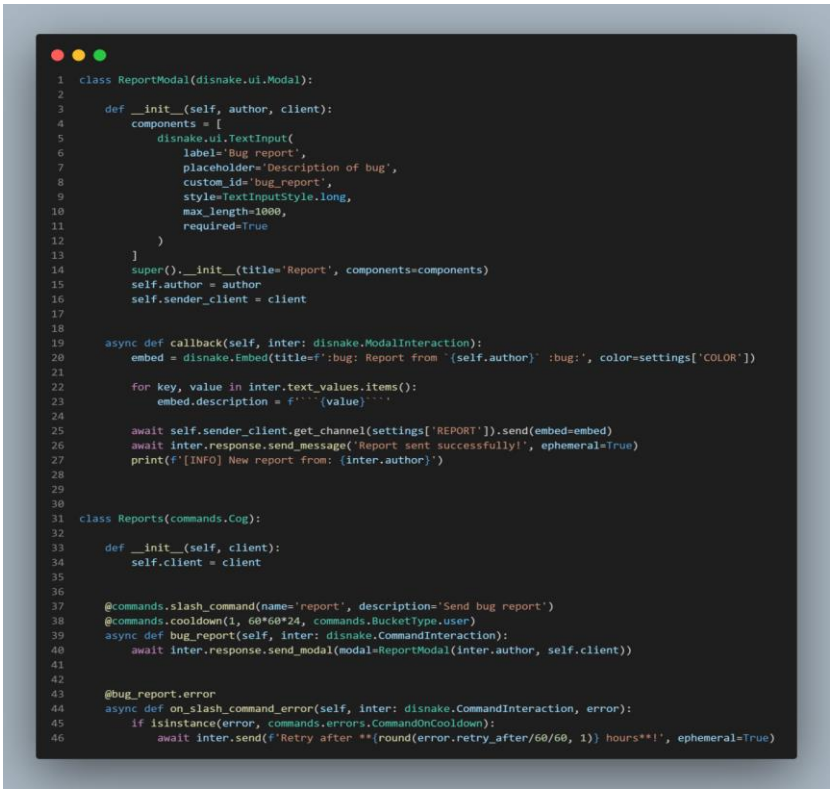
```python
# reload cogs
@client.command(name='reload')
async def reload_cogs(inter: disnake.CommandInteraction):
    if inter.author.id == settings['OWNER_ID']:
        try:
            start_time = round(time.time(), 4)

            for filename in os.listdir('./cogs'):
                if filename.endswith('.py'):
                    client.reload_extension(f'cogs.{filename[:-3]}')

            end_time = round(time.time(), 4)
            await inter.reply(embed=disnake.Embed(title='Cogs successfuly reloaded!', color=settings['COLOR']).set_footer(text=f'Response time: {round((end_time - start_time) * 1000)} ms'))
            print('[RELOAD] Cogs successfuly reloaded')
        except Exception as e:
            await inter.reply(embed=disnake.Embed(title='Reload error', description=f'```py\n{e}\n```', color=disnake.Color.red()))
            print(f'[ERROR] {__name__}: {e}')
    else:
        await inter.reply(embed=disnake.Embed(title=":x: Error :x:", description="You are not an admin!", color=disnake.Color.red()), ephemeral=True)

```

Рисунок 3.9 – Вигляд методу reload_cogs в середовищі

Класи ReportModal та Reports [3, 8].

Для збору звітів про помилки від користувачів я створив модальні вікна, які запускаються через команду /report: ReportModal — відкриває вікно з полем для введення опису проблеми; Reports — це ког, що містить команду /report, яка викликає модальне вікно.

Завдяки цьому користувачі можуть повідомляти про баги прямо в інтерфейсі Discord, а я миттєво отримую звіти в окремий канал.



```python
class ReportModal(disnake.ui.Modal):

    def __init__(self, author, client):
        components = [
            disnake.ui.TextInput(
                label='Bug report',
                placeholder='Description of bug',
                custom_id='bug_report',
                style=TextInputStyle.long,
                max_length=1000,
                required=True
            )
        ]
        super().__init__(title='Report', components=components)
        self.author = author
        self.sender_client = client


    async def callback(self, inter: disnake.ModalInteraction):
        embed = disnake.Embed(title=f':bug: Report from `{self.author}` :bug:', color=settings['COLOR'])

        for key, value in inter.text_values.items():
            embed.description = f'```{value}```'

        await self.sender_client.get_channel(settings['REPORT']).send(embed=embed)
        await inter.response.send_message('Report sent successfully!', ephemeral=True)
        print(f'[INFO] New report from: {inter.author}')



class Reports(commands.Cog):

    def __init__(self, client):
        self.client = client


    @commands.slash_command(name='report', description='Send bug report')
    @commands.cooldown(1, 60*60*24, commands.BucketType.user)
    async def bug_report(self, inter: disnake.CommandInteraction):
        await inter.response.send_modal(modal=ReportModal(inter.author, self.client))


    @bug_report.error
    async def on_slash_command_error(self, inter: disnake.CommandInteraction, error):
        if isinstance(error, commands.errors.CommandOnCooldown):
            await inter.send(f'Retry after **{round(error.retry_after/60/60, 1)} hours**!', ephemeral=True)
```

Рисунок 3.10 – Класи ReportModal і Reports

Метод application_info

Ця команда відображає системну інформацію про бота: кількість серверів, користувачів, версію, пінг, використання ресурсів, час роботи тощо. Разом із цією інформацією користувач бачить також кнопку “Status” — вона веде на сторінку моніторингу стану проєкту.

Метод формує повідомлення у форматі Embed і надсилає його з кнопкою. Це виглядає красиво, інформативно й зручно для користувачів, які хочуть знати, що відбувається з ботом.

Усі описані алгоритми мають одну спільну мету: зробити бота максимально гнучким, масштабованим і зручним в управлінні. Завдяки такій структурі його легко підтримувати, розширювати й оновлювати — без зайвих перезапусків або втрати даних.

```python
@commands.slash_command(name='info', description='Bot information')
async def application_info(self, inter: disnake.CommandInteraction):
    start_time = time.time()

    view = View()
    button_status = Button(label='Status', url=settings['STATUSPAGE_URL'], style=disnake.ButtonStyle.url)
    view.add_item(button_status)

    # uptime
    current_time = time.time()
    difference = int(round(current_time - self.start_time))
    correct_uptime = str(datetime.timedelta(seconds=difference))

    embed = disnake.Embed(title='Information:', color=settings['COLOR'])
    embed.set_author(name=self.client.user, icon_url=self.client.user.avatar, url=settings['GUILD_URL'])
    embed.add_field(name='• Guilds', value=f'```Total: {len(self.client.guilds)} guilds```', inline=True)
    embed.add_field(name='• Users', value=f'```Total: {len(self.client.users)} users```', inline=True)
    embed.add_field(name='• Commands', value=f'```Total: {len(self.client.all_slash_commands.keys())} commands```', inline=True)
    embed.add_field(name='• Ping', value=f'```{round(self.client.latency * 1000)} ms```', inline=True)
    embed.add_field(name='• CPU and RAM usage', value=f'```CPU: {psutil.cpu_percent()}%\nRAM (GB): {round(psutil.virtual_memory().used / pow(1024, 3), 1)}/{round(psutil.virtual_memory().total / pow(1024, 3), 1)}```', inline=True)
    embed.add_field(name='• Bot version', value=f'```{settings["VERSION"]}```', inline=True)
    embed.add_field(name=' ', value=f'[Invite]({settings["INVITE_LINK"]}) • [Support Server]({settings["GUILD_URL"]}) • [Website]({settings["WEBSITE_URL"]}) • [Source Code](https://www.youtube.com/watch?v=dQw4w9WgXcQ)')
    embed.set_footer(text=f'Response time: {round((time.time() - start_time) * 1000)} ms | Uptime: {correct_uptime}')
    await inter.response.send_message(embed=embed, view=view)

```

Рисунок 3.11 – Вигляд методу application_info в середовищі

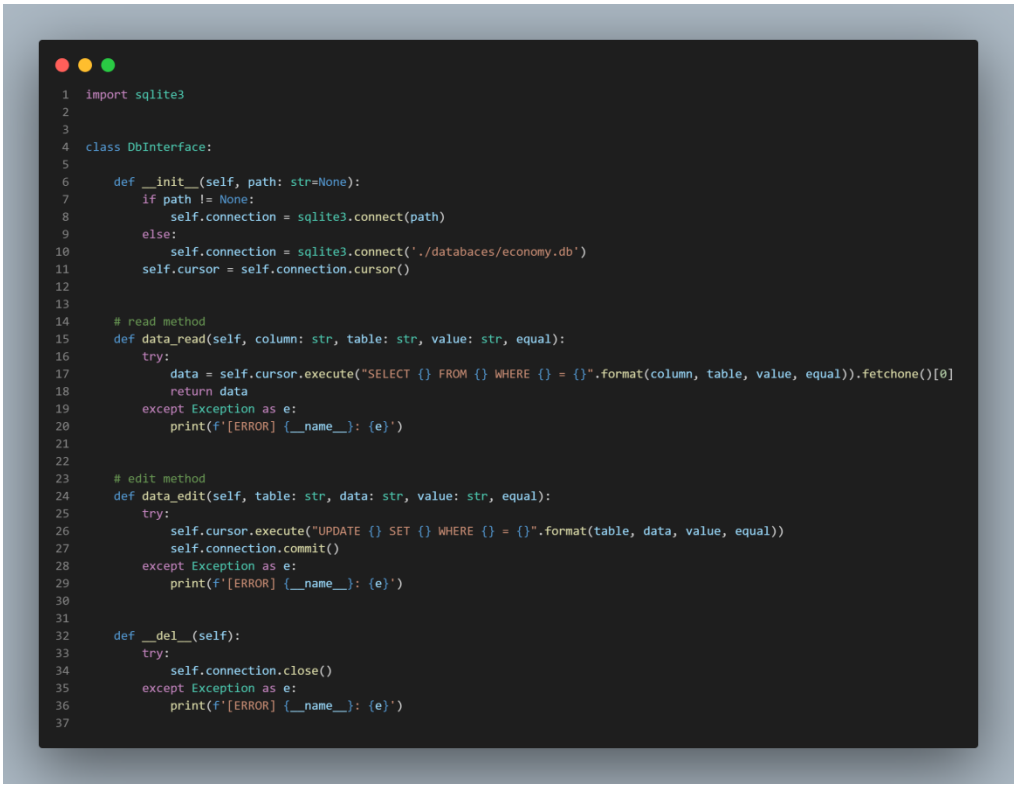
Для взаємодії оссновного коду з базою даних – використовується інтерфейс бази даних – DbInterface (рис 3.12).

У конструкторі класу виконується підключення до бази даних. Якщо переданий шлях до бази даних не є None, то використовується цей шлях для підключення. В іншому випадку, використовується шлях ./databaces/economy.db. Також створюється курсор для виконання SQL-запитів.

Метод data_read виконує запит до бази даних для отримання значення певного стовпця column з таблиці table, де певне поле value має відповідати певному значенню equal. Результат запиту повертається як результат методу.

Метод data_edit виконує запит до бази даних для редагування даних в таблиці table. Він оновлює значення певного стовпця data у рядку, де певне поле value має відповідати певному значенню equal. Зміни в базі даних фіксуються за допомогою self.connection.commit.

Метод __del__ викликається при знищенні об'єкта класу і виконує закриття з'єднання з базою даних.



```python
import sqlite3


class DbInterface:

    def __init__(self, path: str=None):
        if path != None:
            self.connection = sqlite3.connect(path)
        else:
            self.connection = sqlite3.connect('./databaces/economy.db')
        self.cursor = self.connection.cursor()


    # read method
    def data_read(self, column: str, table: str, value: str, equal):
        try:
            data = self.cursor.execute("SELECT {} FROM {} WHERE {} = {}".format(column, table, value, equal)).fetchone()[0]
            return data
        except Exception as e:
            print(f'[ERROR] {__name__}: {e}')


    # edit method
    def data_edit(self, table: str, data: str, value: str, equal):
        try:
            self.cursor.execute("UPDATE {} SET {} WHERE {} = {}".format(table, data, value, equal))
            self.connection.commit()
        except Exception as e:
            print(f'[ERROR] {__name__}: {e}')


    def __del__(self):
        try:
            self.connection.close()
        except Exception as e:
            print(f'[ERROR] {__name__}: {e}')

```

Рисунок 3.12 – Вигляд інтерфейсу бази даних в середовищі

3.4 Програмний код головного файлу для запуску бота

Головний файл є центральною частиною програмної структури Discord-бота, який ініціалізує з’єднання з Discord API, обробляє вхідні повідомлення користувачів, а також реалізує логіку відповідей віртуального консультанта.

Нижче наведено приклад базового коду головного файлу, який забезпечує запуск і первинну функціональність бота [8]:

```
import disnake
from disnake.ext import commands, tasks
import os
import time
from config import settings, guilds
from async_eval import eval
from dotenv import load_dotenv
load_dotenv()
start_time = time.time()
token = [os.getenv('SUMIKO_TOKEN'), os.getenv('CANARY_TOKEN')]
# client = commands.Bot(command_prefix=commands.when_mentioned, intents=disnake.Intents.all(), test_guilds=guilds)  # For Sumiko
client = commands.Bot(command_prefix='.', intents=disnake.Intents.all())  # For Canary
@tasks.loop(seconds=10)
async def changer():
    await client.change_presence(status=disnake.Status.idle, activity=disnake.Activity(type=disnake.ActivityType.competing, name=f'ping: {round(client.latency * 1000)} ms | service mode'))

@client.event
async def on_ready():
    print(f'{client.user.name} ready!\nPing: {round(client.latency * 1000)} ms | Guilds: {len(client.guilds)} | Users: {len(client.users)}')
    client.remove_command('help')
    await client.change_presence(status=disnake.Status.online, activity=disnake.Activity(type=disnake.ActivityType.listening, name='you ❤'))
# eval
@client.command(name='e')
async def eval_command(inter: disnake.CommandInteraction, *, code):
```

```
    if inter.author.id == settings['OWNER_ID']:
        start_time = round(time.time(), 4)
        if 'token' in code:
            await inter.reply(embed=disnake.Embed(title=':x: Error :x:', description="You can't eval this!", color=disnake.Color.red()))
            return
        try:
            result = eval(code)
            end_time = round(time.time(), 4)
            await inter.reply(embed=disnake.Embed(description=f'```py\n{result}\n```', color=settings['COLOR']).set_footer(text=f'Response time: {round((end_time - start_time) * 1000)} ms'))
        except Exception as e:
            await inter.reply(embed=disnake.Embed(title=f'Eval: "{code}" error', description=f'```py\n{e}\n```', color=disnake.Color.red()))
    else:
        await inter.reply(embed=disnake.Embed(title=':x: Error :x:', description='You are not an admin!', color=disnake.Color.red()), ephemeral=True)
# reload cogs
@client.command(name='reload')
async def reload_cogs(inter: disnake.CommandInteraction):
    if inter.author.id == settings['OWNER_ID']:
        try:
            start_time = round(time.time(), 4)

            for filename in os.listdir('./cogs'):
                if filename.endswith('.py'):
                    client.reload_extension(f'cogs.{filename[:-3]}')
        end_time = round(time.time(), 4)
```

```
                await inter.reply(embed=disnake.Embed(title='Cogs successfuly reloaded!', color=settings['COLOR']).set_footer(text=f'Response time: {round((end_time - start_time) * 1000)} ms'))
                print('[RELOAD] Cogs successfuly reloaded')
            except Exception as e:
                await inter.reply(embed=disnake.Embed(title='Reload error', description=f```py\n{e}\n```', color=disnake.Color.red()))
                print(f'[ERROR] {__name__}: {e}')
        else:
            await inter.reply(embed=disnake.Embed(title=":x: Error :x:", description="You are not an admin!", color=disnake.Color.red()), ephemeral=True)
# service mode
@client.command(name='smode')
async def service_mode(inter: disnake.CommandInteraction, mode: str):
    if mode == 'start':
        changer.start()
        Loader().unload_cogs()
        await inter.reply(f'Service mode {mode}')
        print(f'[DEBUG] Service mode {mode}')
        return
    if mode == 'stop':
        changer.stop()
        Loader().load_cogs()
        await client.change_presence(status=disnake.Status.online, activity=disnake.Activity(type=disnake.ActivityType.listening, name='you ❤'))
        await inter.reply(f'Service mode {mode}')
        print(f'[DEBUG] Service mode {mode}')
        return
class Loader:
    # unload cogs
```

```
        def unload_cogs(self):
                try:
                        for filename in os.listdir('./cogs'):
                                if filename.endswith('.py'):
                                        client.unload_extension(f'cogs.{filename[:-
3]}')
                        print('[UNLOAD] Cogs unloaded')
                except Exception as e:
                        print(f'[ERROR] {__name__}: {e}')
        # load cogs
        def load_cogs(self):
                try:
                        for filename in os.listdir('cogs'):
                                if filename.endswith('.py'):
                                        client.load_extension(f'cogs.{filename[:-3]}')
                        print('[LOAD] Cogs loaded')
                except Exception as e:
                        print(f'[ERROR] {__name__}: {e}')
if __name__ == '__main__':
    Loader().load_cogs()
    try:
        client.run(token[0])
    except Exception as e:
        print(f'[ERROR] {__name__}: {e}')
```

Аналізуючи частину програмного коду головної сторінки, можна зробити висновок, що програма створена на Python з використанням платформі Visual Studio Code та створена на базі бібліотеки disanke.

Висновок до розділу

У цьому розділі було реалізовано програмну частину віртуального консультанта для месенджера Discord. Проведено інтеграцію з Discord API за допомогою мови програмування Python та бібліотеки discord.py, що забезпечило стабільне підключення бота до серверу та обробку текстових повідомлень користувачів.

Представлений код забезпечує базову функціональність Discord-бота, здатного приймати команди користувача, обробляти відповіді та надсилати їх у чат. Така архітектура є розширюваною та дозволяє легко інтегрувати додаткові функції, логіку, бази даних або інтерфейс адміністрування для покращення роботи віртуального консультанта.

4 ЕКСПЕРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РОЗРОБЛЕНОГО АВТОМАТИЧНОГО ВІРТУАЛЬНОГО КОНСУЛЬТАНТА ДЛЯ МЕСЕНДЖЕРА DISOCRD

Автоматичний віртуальний консультант (чат-бот) для месенджера Discord - це програма, створена для надання користувачам можливості отримувати консультацію та інформацію за допомогою віртуального помічника у месенджері Discord. З метою оцінки ефективності розробленого віртуального консультанта було проведено експериментальне дослідження в тестовому середовищі месенджера Discord. Основною задачею дослідження стало перевірення працездатності, швидкодії, якості відповідей та стабільності роботи бота в умовах реального діалогу з користувачами.

Для експериментального тестування було створено окремий Discord-сервер з кількома каналами. До сервера було додано віртуального консультанта у вигляді Discord-бота, розгорнутого на хостинговій платформі з постійним доступом до Інтернету. У тестуванні брали участь користувачі, які моделювали різні сценарії взаємодії, що охоплювали як стандартні, так і нестандартні запити.

Було визначено такі основні критерії дослідження: час відповіді на запит користувача; якість та релевантність відповідей; стійкість роботи бота при багаторазових викликах; обробка помилкових або некоректних запитів; контекстна послідовність у діалозі.

4.1 Загальна структура програми

На фінальному етапі розробки особливо важливо було структурувати проєкт так, щоб бот залишався зрозумілим, легко розширювався та не перетворювався на “хаос із файлів”. Я дотримувався принципу модульності — розділив функціональність на окремі компоненти, кожен із яких відповідає за свою ділянку.

Загалом структура програми виглядає так [4, 8]:

- Головний файл (main.py) — це стартова точка. Саме він ініціалізує підключення до Discord, завантажує модулі, задає поведінку бота при запуску та обробляє глобальні команди.
- Модулі команд — у цій частині зібрані всі слеш-команди, які користувач може викликати. Наприклад: /balance, /profile, /give, /anime, /weather тощо. Команди винесено в окремі файли, кожен з яких виконує лише свою функцію.
- Папка **cogs** — це логічне "ядро" бота. У кожному когові зберігаються функції або обробники подій: хтось приєднався до сервера, хтось надіслав повідомлення, хтось отримав новий рівень. Кожен файл тут — окрема мікросистема, яку можна оновлювати, вмикати чи вимикати незалежно від інших.
- База даних (SQLite) — вся важлива інформація (ID користувачів, статистика, преміум-статуси, історія команд) зберігається локально у SQLite. Такий підхід простий і ефективний: не потрібно додаткових серверів, а читання і запис відбуваються миттєво.
- Інтерфейс бази даних (DbInterface) — це окремий клас, через який усі модулі взаємодіють із БД. Завдяки цьому зміни в логіці збереження не потребують редагування кожного окремого модуля — достатньо змінити метод в одному місці.
- Додаткові компоненти — логування, обробка помилок, генерація зображень, модальні вікна, Discord-статуси, інтеграції з API, наприклад, для отримання погоди чи інформації про аніме.

Ця структура дозволяє:

- легко масштабувати проєкт — новий функціонал додається у вигляді нового модуля, не зачіпаючи існуючі;
- оновлювати код без перезапуску бота, просто через перезавантаження когів;
- відокремити логіку, дані й інтерфейс, що значно спрощує налагодження та тестування.

У підсумку бот вийшов не лише функціональним, а й технічно зручним для розробника. Це справжній каркас, на який легко “нарощувати м'язи” — додавати нові команди, реакції, бази, API тощо.

4.2 Опис команд бота

Після приєднання бота до нової гільдії, він відправляє 2 повідомлення в різні чати. Перше повідомлення про приєднання бота до нової гідьдії відправляється в приватний чат з логами бота. Друге повідомлення відправляється в приватний чат користувача, який приєднав бота до гільдії. В ньому йдеться про подяку за приєднання до гільдії та рекомендації перейти на сайт бота і його гільдію підтримки, де в свою чергу можуть надати допомогу при появі питань. Вигляд цих повідомлень зображено на рисунках 4.1 та 4.2 відповідно.

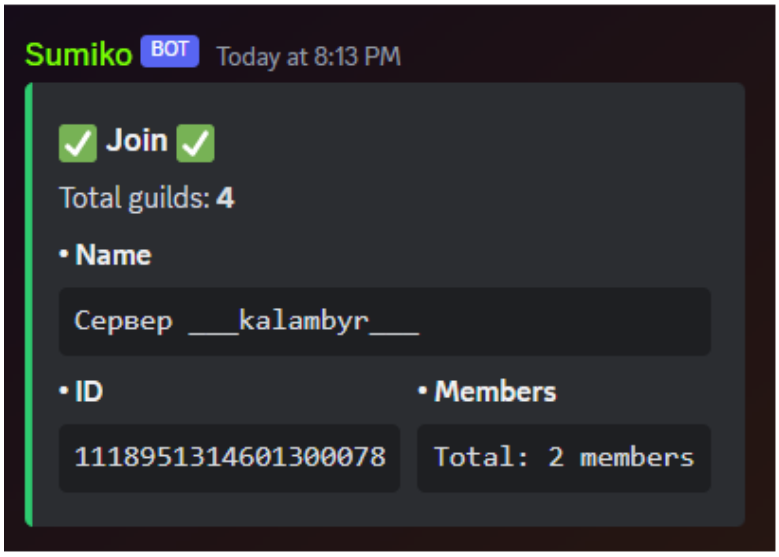


Рисунок 4.1 – Вигляд повідомлення про приєднання до гільдії

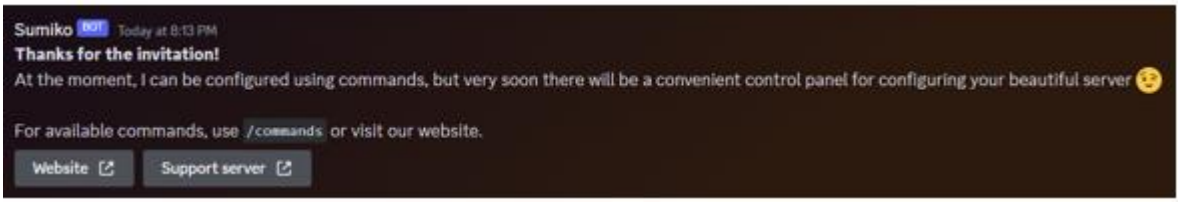


Рисунок 4.2 – Вигляд повідомлення подяки за приєднання

На рис. 4.3, використовуючи команду /commands можна побачити всі доступні команди бота. На момент написання ДП кількість цих команд дорівнює 23.

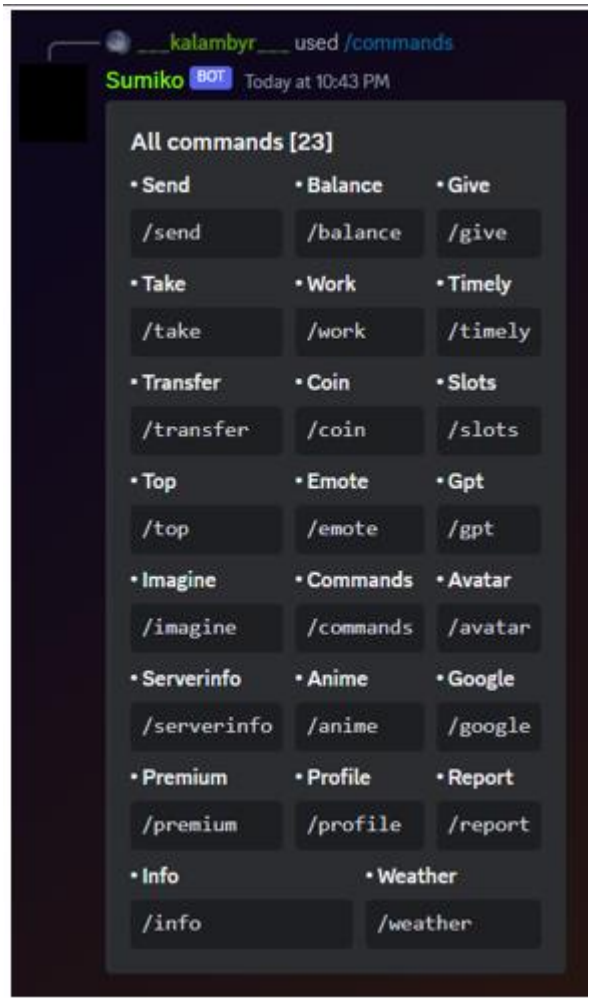


Рисунок 4.3 – Список всіх доступних команд бота

– /balance – відображає поточний баланс користувача. Вся інформація береться з бази даних. Якщо користувач викликає цю команду вперше, бот створює для нього відповідний запис автоматично. Приклад використання цієї команди зображений на рис. 4.4.

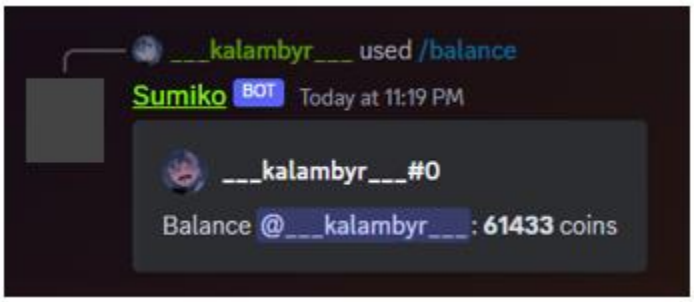


Рисунок 4.4 – Приклад використання команди /balance

– /give – Дозволяє передати частину балансу іншому користувачу. При цьому бот перевіряє, чи в того, хто передає, достатньо коштів, і автоматично оновлює базу. Приклад використання цієї команди зображений на рисунку 4.5.

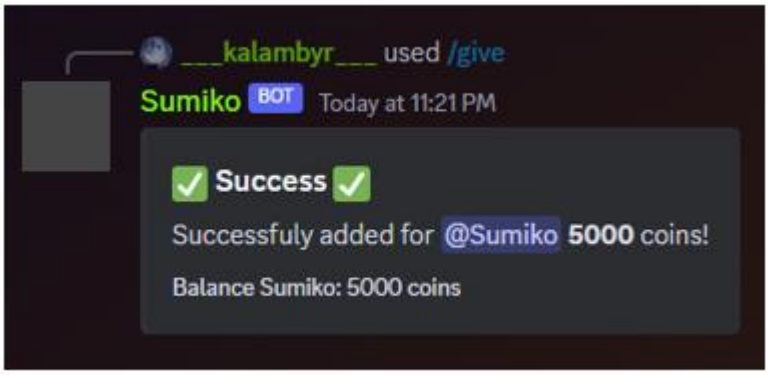


Рисунок 4.5 – Приклад використання команди /give

– /take – команда /take дозволяє забрати певну суму грошей з рахунку користувача. Це може бути корисно для списання коштів за певні послуги або товари, які надаються в проєкті. Приклад використання цієї команди зображений на рис.. 4.6.

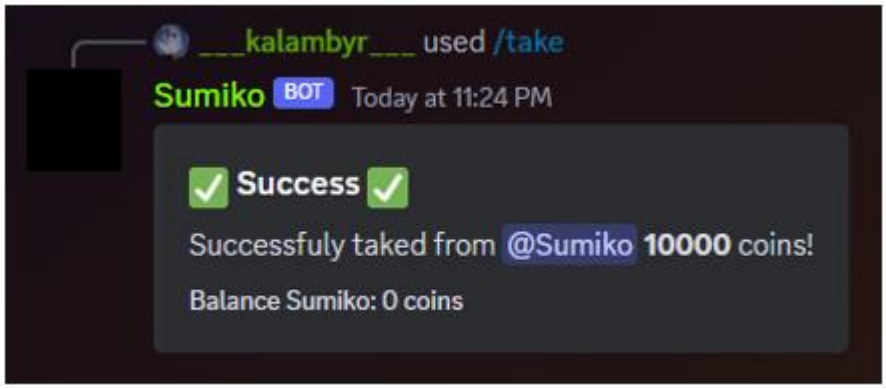


Рисунок 4.6 – Приклад використання команди /take

– /work – команда /work надає можливість користувачеві виконувати роботу для отримання грошової винагороди. Це може бути щоденна або регулярна робота, яку користувачі можуть виконувати, щоб заробляти гроші у проєкті. В команді також прописаний ліміт на її використання (можна виконати одну команду на 20 хвилин). Приклад використання цієї команди та її ліміту зображений на рис. 4.7.

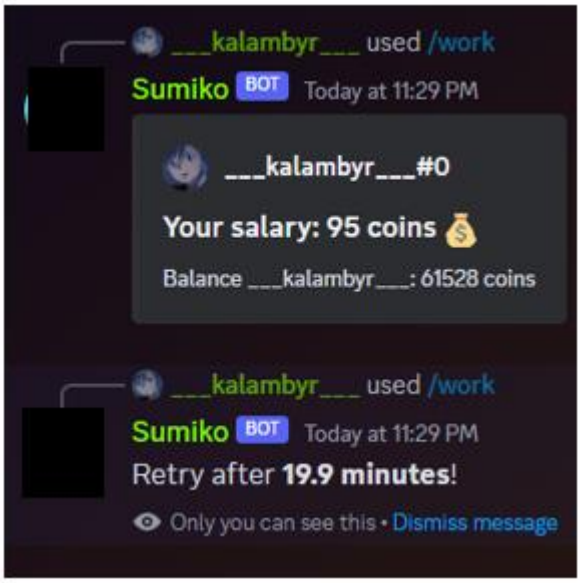


Рисунок 4.7 – Приклад використання команди /work та її ліміт

– /timely – команда /timely надає щотижневий бонус грошей користувачеві. Це може бути спосіб стимулювання активності користувачів та надання їм додаткових ресурсів для використання у проєкті. Приклад використання цієї команди та її ліміту зображений на рис. 4.8.

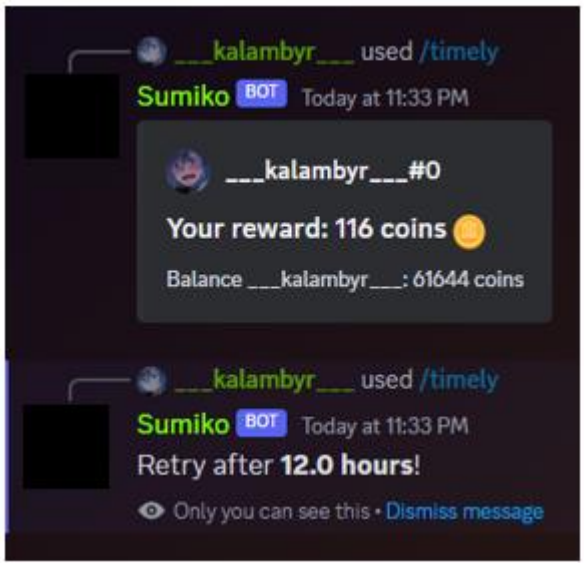


Рисунок 4.8 – Приклад використання команди /timely та її ліміт

– /transfer – команда /transfer дозволяє передати певну кількість грошей на рахунок іншого користувача. Вона подібна до команди /give, але може бути

корисною, коли передача коштів відбувається між різними рахунками користувачів. Приклад використання цієї команди зображений на рис. 4.9.

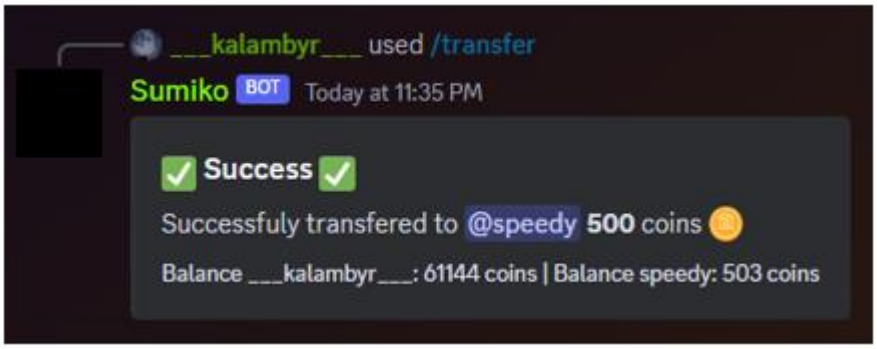


Рисунок 4.9 – Приклад використання команди /transfer

– /coin – команда /coin запускає гру у монетку. Це може бути розважальною функцією, де користувачі можуть випробувати свою удачу та отримати виграш. Приклад використання цієї команди зображений на рис. 4.10.

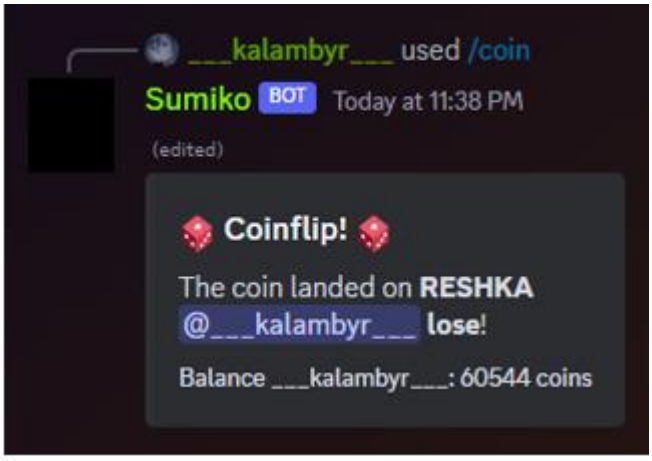


Рисунок 4.10 – Приклад використання команди /coin

– /slots – команда /slots запускає гру в азартні автомати з можливістю виграти грошовий приз. Це може бути функціоналом для розваги, який дозволяє користувачам спробувати свою удачу та отримати виграш у вигляді грошей. Приклад використання цієї команди зображений на рисунку 4.11.

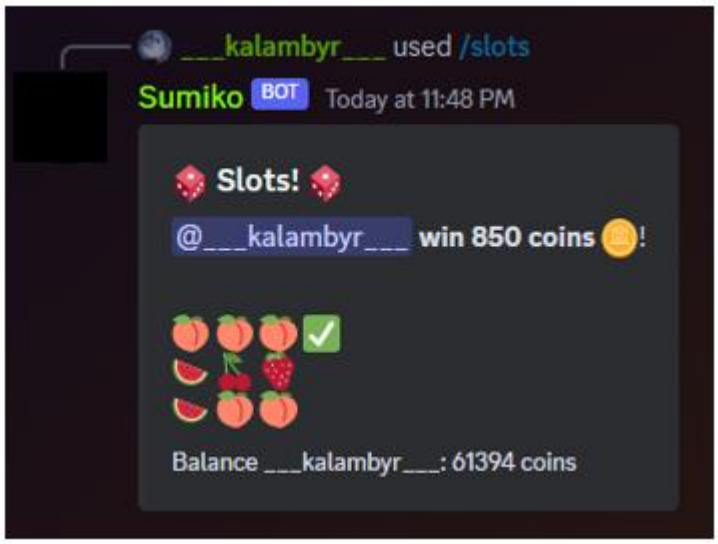


Рисунок 4.11 – Приклад використання команди /slots

– /top – команда /top показує список найкращих користувачів за рівнем, балансом або кількістю повідомлень. Це може стимулювати конкуренцію серед учасників проєкту та показувати успішних користувачів. Приклади використання цієї команди в трьох варіантах зображені на рис. 4.12, 4.13 та 4.14 відповідно.

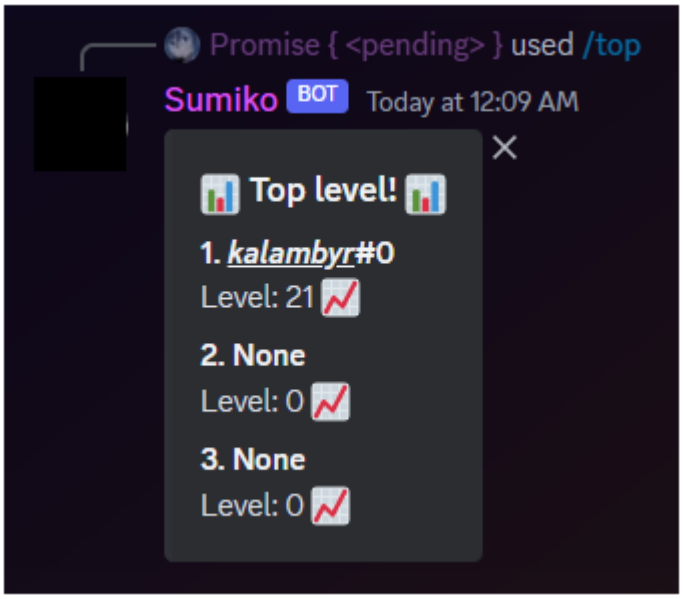


Рисунок 4.12 – Приклад використання команди /top з сортуванням по рівню

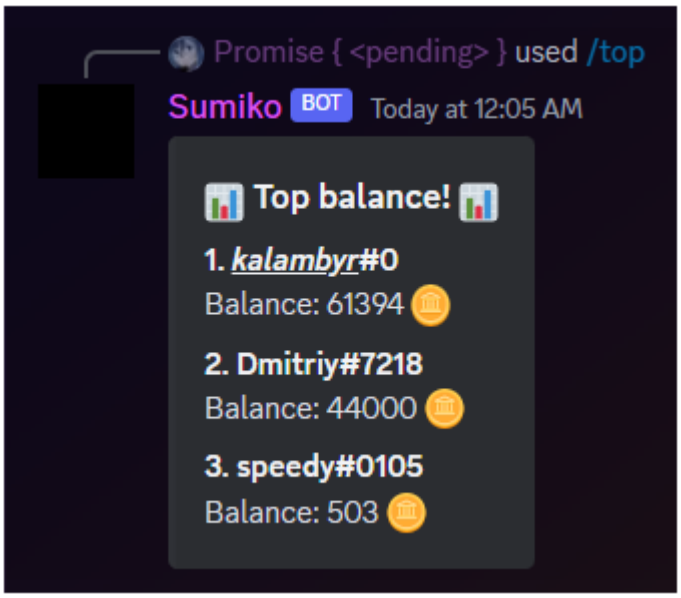


Рисунок 4.13 – Приклад використання команди /top з сортуванням по балансу

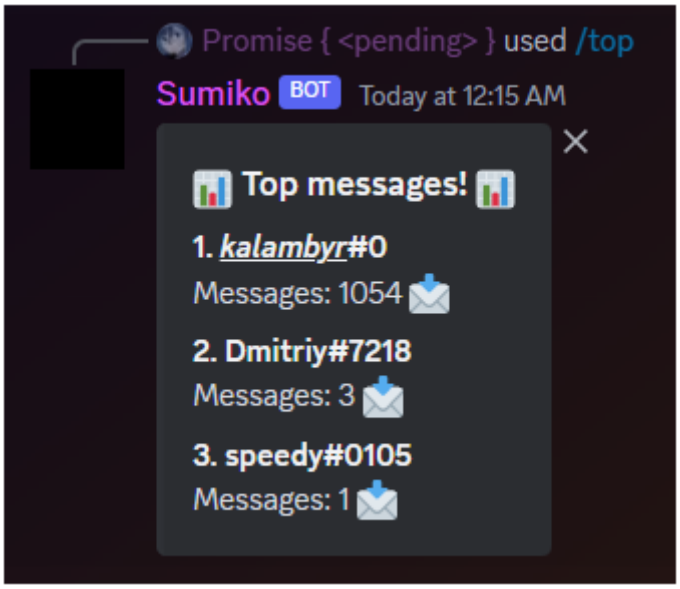


Рисунок 4.14 – Приклад використання команди /top з сортуванням по кількості повідомлень

– /serverinfo – команда /serverinfo надає інформацію про сервер Discord, таку як назва, кількість учасників, дата створення тощо. Це може бути корисною функцією для отримання загального огляду проєкту. Приклад використання цієї команди зображений на рис. 4.15.

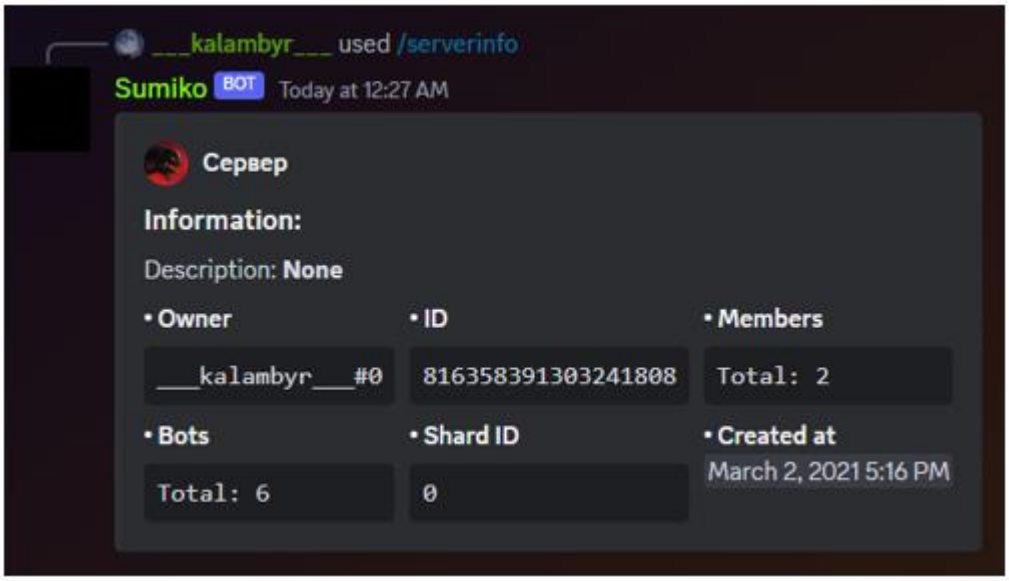


Рисунок 4.15 – Приклад використання команди /serverinfo

– /google – команда /google виконує пошук на Google та надає результати запиту. Користувачі можуть використовувати цю команду, щоб швидко знайти інформацію з Інтернету безпосередньо в Discord. Приклад використання цієї команди зображений на рис. 4.16.

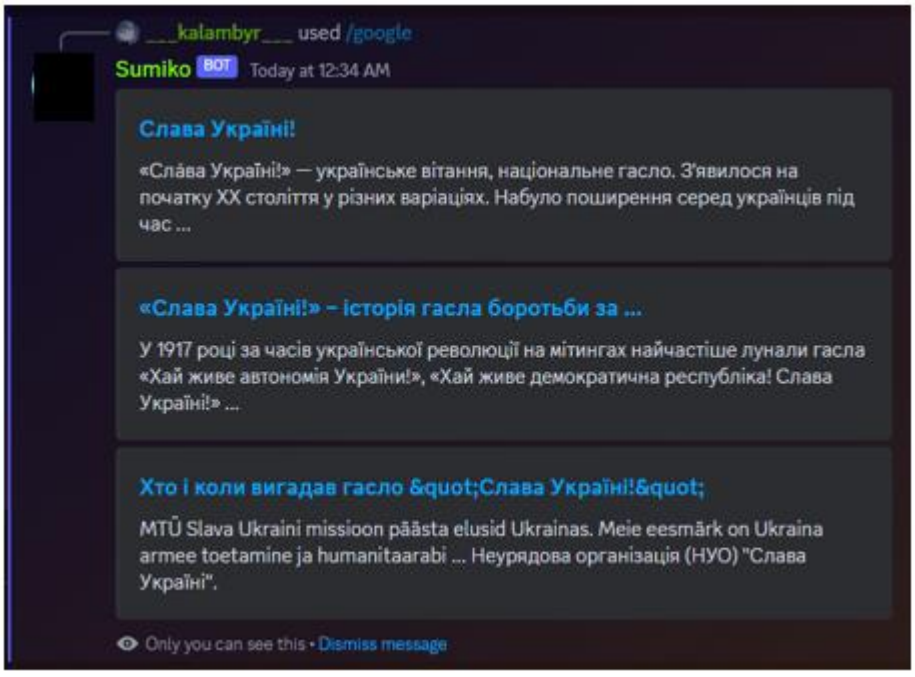


Рисунок 4.16 – Приклад використання команди /google

– /premium – команда /premium надає інформацію про платну преміум-версію чат-бота з додатковими функціями. Вона може містити переваги, доступні для користувачів, які оформлюють преміум-підписку.

– /profile – Показує профіль користувача у вигляді красиво оформленого Embed-повідомлення. Тут зазначені: ID, рівень, кількість повідомлень, статус преміуму (якщо є), кількість предметів тощо. Це свого роду “паспорт” учасника на сервері. Приклад використання цієї команди зображений на рис 4.17.

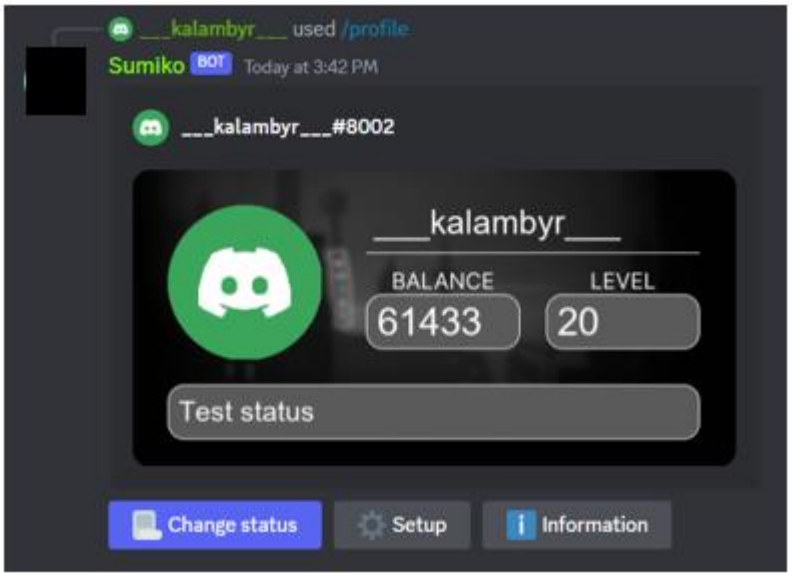


Рисунок 4.18 – Приклад використання команди /profile

– /report – Ця команда відкриває модальне вікно, де користувач може ввести опис проблеми або помилки. Після відправлення повідомлення потрапляє у внутрішній технічний канал — розробник одразу бачить, що щось не так, і може оперативно реагувати. Приклад використання цієї команди зображений на рис. 4.19.

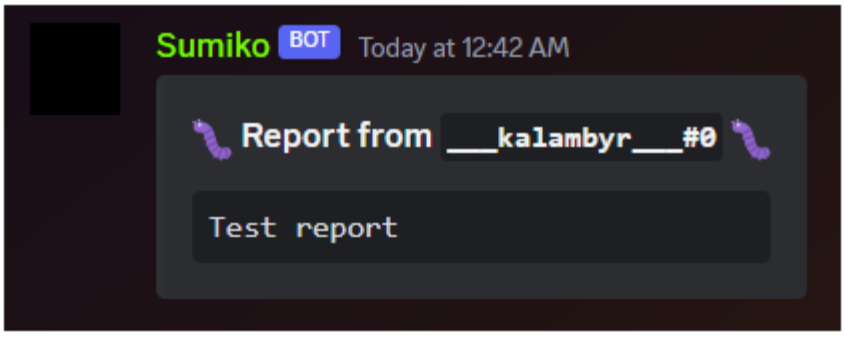


Рисунок 4.19 – Приклад використання команди /report

– /info – команда /info надає загальну інформацію про чат-бота. Вона містить версію бота, кількість доступних команд, загальну кількість гільдій та користувачів та іншу інформацію, яка допомагає користувачам краще розуміти бота. Приклад використання цієї команди зображений на рис. 4.20.

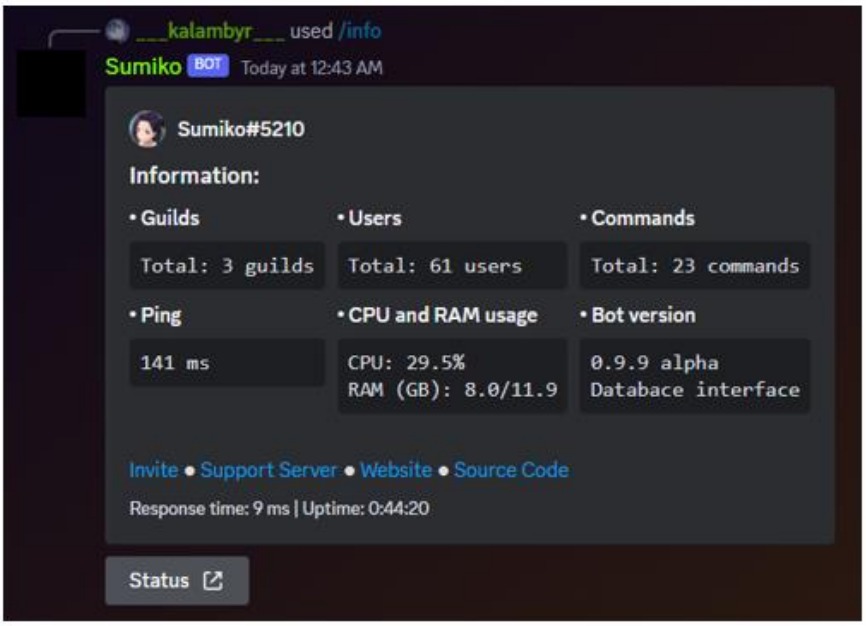


Рисунок 4.20 – Приклад використання команди /info

– /weather – Ця команда підтягує прогноз погоди для заданого міста. Реалізована через зовнішній API, який повертає актуальні погодні дані: температуру, стан неба, вологість, вітер тощо. У відповідь бот надсилає оформлену картку з погодною інформацією. Приклад використання цієї команди зображений на рис. 4.21.

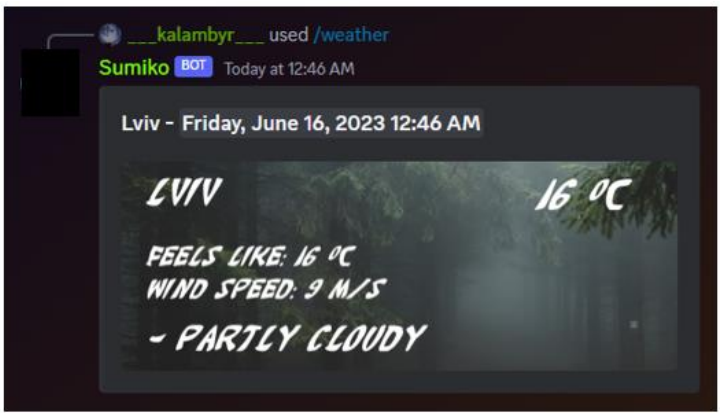


Рисунок 4.21 – Приклад використання команди /weather

Висновок до розділу

Експериментальне дослідження підтвердило працездатність та ефективність розробленого віртуального консультанта в умовах реального середовища Discord. Бот демонструє високий рівень взаємодії з користувачами, здатний відповідати на різні запити, підтримує діалогову форму спілкування та стійко функціонує при навантаженнях.

Отримані результати свідчать про доцільність використання такого рішення для автоматизації підтримки навчальних груп або інформаційних каналів.

ЗАГАЛЬНІ ВИСНОВКИ

У бакалаврській роботі було розглянуто процес розроблення автоматичного віртуального консультанта для месенджера Discord, що базується на використанні сучасних технологій програмування та інструментів штучного інтелекту. В результаті проведеного аналізу виявлено особливості функціонування популярних Discord-ботів, визначено вимоги до інтелектуального консультанта та обґрунтовано вибір відповідних платформ і засобів реалізації.

Для створення системи було обрано мову програмування Python разом із бібліотекою discord.py та інтеграцією з OpenAI API, що дозволяє реалізувати обробку природної мови, генерацію змістовних відповідей і ведення контекстного діалогу з користувачами. Також налаштовано зручне середовище розробки на базі Visual Studio Code, що забезпечує ефективну роботу з кодом, віртуальними середовищами та системами контролю версій.

Розроблений віртуальний консультант демонструє базову функціональність у режимі реального часу: сприймає запити користувачів, аналізує їх зміст, формує відповіді відповідно до заданої логіки та забезпечує інтерактивну взаємодію в межах сервера Discord. Такий підхід дозволяє автоматизувати інформаційну підтримку, підвищити зручність обслуговування спільнот та забезпечити масштабованість у подальшому розвитку проєкту.

Експериментальне дослідження підтвердило працездатність та ефективність розробленого віртуального консультанта в умовах реального середовища Discord. Бот демонструє високий рівень взаємодії з користувачами, здатний відповідати на різні запити, підтримує діалогову форму спілкування та стійко функціонує при навантаженнях.

Отримані результати свідчать про доцільність використання такого рішення для автоматизації підтримки навчальних груп або інформаційних каналів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Козаченко В.І. Штучний інтелект: теорія та практичні застосування. – К.: Наукова думка, 2020. – 320 с.
2. Посилання на офіційну документацію Discord для написання ботів. Режим доступу. – https://discord.com/developers/docs/intro
3. Політика Discord для розробників. Режим доступу. – https://discord.com/developers/docs/policies-and-agreements/developer-policy
4. Умови використання діскорд для розробників. Режим доступу. – https://discord.com/developers/docs/policies-and-agreements/developer-terms-of-service
5. PyNaCl (1.5.0). Режим доступу. –[https://github.com/pyca/pynacl]
6. Psutil (5.9.4). Режим доступу. – [https://github.com/giampaolo/psutil]
7. Python-dotenv (0.21.0). Режим доступу. – [https://github.com/theskumar/python-dotenv]
8. Openai (0.26.5). Режим доступу. – [https://github.com/openai/openai-python]

Додаток А

Коди модулів

```
import disnake
from disnake.ext import commands
from disnake import Option, OptionType, OptionChoice, TextInputStyle
from config import settings
import time
import sqlite3
from PIL import Image, ImageFont, ImageDraw
from io import BytesIO
# style select menu
class StyleSelectMenu(disnake.ui.StringSelect):
    def __init__(self):
        options = [
            disnake.SelectOption(label='Profile embed',
value='embed'),
            disnake.SelectOption(label='Profile card',
description='Premium only', value='card'),
        ]
        super().__init__(
            placeholder='Choose your profile style',
            min_values=1,
            max_values=1,
            options=options,
        )
       async def callback(self, inter: disnake.MessageInteraction):
        if cursor.execute("SELECT premium FROM users WHERE id =
{}".format(inter.author.id)).fetchone()[0] == 'False' and
self.values[0] == 'card':
            await
inter.response.send_message(f'{inter.author.mention}, you must have
premium for that!', ephemeral=True)
            return
        cursor.execute("UPDATE users SET profile = '{}' WHERE id =
{}".format(self.values[0], inter.author.id))
        connection.commit()
        await inter.response.send_message(f'Your profile style
changed to `{self.values[0]}`', ephemeral=True)
        print(f'[INFO] Changed profile style to {self.values[0]}')
```

```
# add select menu
class SelectMenuView(disnake.ui.View):
    def __init__(self):
        super().__init__()
        self.add_item(StyleSelectMenu())
# full buttons
class FullProfileButtons(disnake.ui.View):

    def __init__(self, member: disnake.Member):
        super().__init__(timeout=None)
        self.member = member
    @disnake.ui.button(label='Change status', emoji='',
style=disnake.ButtonStyle.blurple)
    async def button_change_status(self, button: disnake.ui.Button,
inter: disnake.CommandInteraction):
        if self.member == inter.author:
            await inter.response.send_modal(modal=NewStatusModal())
        else:
            await inter.response.send_message('You can`t do this!',
ephemeral=True)
        @disnake.ui.button(label='Setup', emoji='⚙',
style=disnake.ButtonStyle.grey)
    async def button_setup(self, button: disnake.ui.Button, inter:
disnake.CommandInteraction):
        if self.member == inter.author:
            profile_style = cursor.execute("SELECT profile FROM users
WHERE id = {}".format(inter.author.id)).fetchone()[0]
            embed = disnake.Embed(title='Settings',
color=settings['COLOR'])
            embed.set_author(name=inter.author,
url=f'https://discord.com/users/{inter.author.id}',
icon_url=inter.author.default_avatar.url if inter.author.avatar ==
None else inter.author.avatar.url)
            embed.add_field(name='• Profile style',
value=f'```{profile_style}```', inline=False)
            await inter.response.send_message(embed=embed,
view=SelectMenuView(), ephemeral=True)
        else:
```

```
            await inter.response.send_message('You can`t do this!',
ephemeral=True)
      @disnake.ui.button(label='Information', emoji='ℹ',
style=disnake.ButtonStyle.grey)
    async def button_user_info(self, button: disnake.ui.Button, inter:
disnake.CommandInteraction):
        user = self.member

        embed = disnake.Embed(color=user.color)
        if user.activity != None:
            embed.description = str(user.activity)
        embed.set_author(name=user,
url=f'https://discord.com/users/{user.id}',
icon_url=user.default_avatar.url if user.avatar == None else
user.avatar.url)
        embed.add_field(name='• Tag', value=f'```{user}```',
inline=True)
        embed.add_field(name='• ID', value=f'```{user.id}```',
inline=True)
        embed.add_field(name='• Bot', value=f'```{user.bot}```',
inline=True)
        embed.add_field(name='• Cteated at',
value=f'<t:{round(time.mktime(user.created_at.timetuple()))}:f>',
inline=True)
        embed.add_field(name='• Joined at',
value=f'<t:{round(time.mktime(user.joined_at.timetuple()))}:f>',
inline=True)
        embed.add_field(name='• Roles', value=',
'.join([f'{role.mention}' for role in user.roles]), inline=True)

        await inter.response.send_message(embed=embed, ephemeral=True)
 short buttons
class ShortProfileButtons(disnake.ui.View):

    def __init__(self, member: disnake.Member):
        super().__init__(timeout=None)
        self.member = member
        @disnake.ui.button(label='Information', emoji='ℹ',
style=disnake.ButtonStyle.grey)
    async def user_info(self, button: disnake.ui.Button, inter:
disnake.CommandInteraction):
```

```
        user = self.member
                embed = disnake.Embed(color=user.color)
        if user.activity != None:
            embed.description = str(user.activity)
        embed.set_author(name=user,
url=f'https://discord.com/users/{user.id}',
icon_url=user.default_avatar.url if user.avatar == None else
user.avatar.url)
        embed.add_field(name='• Tag', value=f'```{user}```',
inline=True)
        embed.add_field(name='• ID', value=f'```{user.id}```',
inline=True)
        embed.add_field(name='• Bot', value=f'```{user.bot}```',
inline=True)
        embed.add_field(name='• Cteated at',
value=f'<t:{round(time.mktime(user.created_at.timetuple()))}:f>',
inline=True)
        embed.add_field(name='• Joined at',
value=f'<t:{round(time.mktime(user.joined_at.timetuple()))}:f>',
inline=True)
        embed.add_field(name='• Roles', value=',
'.join([f'{role.mention}' for role in user.roles]), inline=True)

        await inter.response.send_message(embed=embed, ephemeral=True)
# change status
class NewStatusModal(disnake.ui.Modal):

    def __init__(self):
        components = [
            disnake.ui.TextInput(
                label='New status',
                placeholder='Enter your status',
                custom_id='status',
                style=TextInputStyle.long,
                min_length=1,
                max_length=60,
                required=True,
            )
        ]
        super().__init__(title='Set new status',
components=components, custom_id='new_satus')
```

```
    async def callback(self, inter: disnake.ModalInteraction):
        for key, value in inter.text_values.items():
            new_status = value
        cursor.execute("UPDATE users SET status = '{}' WHERE id =
{}".format(new_status, inter.author.id))
        connection.commit()
        await inter.response.send_message(f'Your status changed to
`{new_status}`', ephemeral=True)
        print('[INFO] Change status')
class Profile(commands.Cog):

    def __init__(self, client):
        self.client = client

    # profile
    @commands.slash_command(name='profile', description='Show member
profile',
        options=[
            Option('member', 'Choose member', OptionType.user,
required=False),
            Option('visible', 'True or False', OptionType.string,
required=False,
                choices=[
                    OptionChoice('True', 'True'),
                    OptionChoice('False', 'False'),
                ],
            ),
        ],
    )
    async def profile(self, inter: disnake.CommandInteraction, member:
disnake.Member=None, visible=False):
        member = member or inter.author
        # buttons
        full_view = FullProfileButtons(member)
        short_view = ShortProfileButtons(member)
        # set level
        user_messages = cursor.execute("SELECT messages FROM users
WHERE id = {}".format(member.id)).fetchone()[0]
        if user_messages > 50:
            array = []
            for i in range(0, user_messages, 50):
```

```
                if i != 0:
            array.append(i)

            cursor.execute("UPDATE users SET level = {} WHERE id =
{}".format(len(array), member.id))
            array.clear()
                # check premium
        if int(cursor.execute("SELECT premium_end_time FROM users
WHERE id = {}".format(member.id)).fetchone()[0]) < round(time.time()):
            cursor.execute("UPDATE users SET profile = 'embed' WHERE
id = {}".format(member.id))
            cursor.execute("UPDATE users SET premium = 'False' WHERE
id = {}".format(member.id))
            cursor.execute("UPDATE users SET premium_start_time = {}
WHERE id = {}".format(0, member.id))
            cursor.execute("UPDATE users SET premium_end_time = {}
WHERE id = {}".format(0, member.id))
        connection.commit()
        # get data from db
        member_status = cursor.execute("SELECT status FROM users
WHERE id = {}".format(member.id)).fetchone()[0]
        member_balance = cursor.execute("SELECT coins FROM users
WHERE id = {}".format(member.id)).fetchone()[0]
        member_level = cursor.execute("SELECT level FROM users WHERE
id = {}".format(member.id)).fetchone()[0]
        member_has_premium = cursor.execute("SELECT premium FROM
users WHERE id = {}".format(member.id)).fetchone()[0]
        member_profile_style = cursor.execute("SELECT profile FROM
users WHERE id = {}".format(member.id)).fetchone()[0]
        embed = disnake.Embed(color=settings['COLOR'])
        embed.set_author(name=member,
url=f'https://discord.com/users/{member.id}',
icon_url=member.default_avatar.url if member.avatar == None else
member.avatar.url)
        await inter.response.defer(ephemeral=visible)
        if member_profile_style == 'embed':
            embed.add_field(name='• Status',
value=f'```\n{member_status}\n```', inline=False)
            embed.add_field(name='• Balance',
value=f'```css\n{member_balance} coins\n```', inline=True)
```

```
            embed.add_field(name='• Level',
value=f'```css\n{member_level}\n```', inline=True)
            embed.add_field(name='• Has premium',
value=f'```css\n{member_has_premium}\n```', inline=True)
        else:
            background =
Image.open('./src/img/profile/profile_card.png')
            draw = ImageDraw.Draw(background)
            avatar = BytesIO(await member.display_avatar.read())
            ava = Image.open(avatar)
            ava = ava.resize((140, 140))

            mask = Image.new('L', ava.size, 0)
            draw1 = ImageDraw.Draw(mask)

            draw1.ellipse((0, 0, 140, 140), fill=290)
            background.paste(ava, (32, 32), mask=mask)
            font_nickname =
ImageFont.truetype(font='./src/fonts/arial.ttf', size=30)
            font_details =
ImageFont.truetype(font='./src/fonts/arial.ttf', size=35)
            font_status =
ImageFont.truetype(font='./src/fonts/arial.ttf', size=25)
            if len(member.display_name) >= 20:
                draw.text(xy=(220, 30),
text=f'{member.display_name[:17]}..', font=font_nickname,
fill='#ffffff')
            else:
                draw.text(xy=(220, 30), text=member.display_name,
font=font_nickname, fill='#ffffff')
            draw.text(xy=(223, 118), text=str(member_balance),
font=font_details, fill='#ffffff')
            draw.text(xy=(387, 118), text=str(member_level),
font=font_details, fill='#ffffff')
            if len(str(member_status)) >= 35:
                draw.text(xy=(42, 205),
text=f'{member_status[:32]}..', font=font_status, fill='#ffffff')
            else:
                draw.text(xy=(42, 205), text=str(member_status),
font=font_status, fill='#ffffff')
```

```
            with BytesIO() as a:
                background.save(a, 'PNG', subsampling=0, quality=1000)
                a.seek(0)
                embed.set_image(file=disnake.File(a,
'profile_card.png'))
        # check if author
        if member == inter.author:
            await inter.followup.send(embed=embed, view=full_view)
            await full_view.wait()
        else:
            await inter.followup.send(embed=embed, view=short_view)
            await short_view.wait()
def setup(client):
    global connection
    global cursor
    try:
        connection = sqlite3.connect('./databaces/economy.db')
        cursor = connection.cursor()
        print(f'[DB] [{__name__}] Databace connected')
    except Exception as e:
        print(f'[ERROR] {__name__}: {e}')
    client.add_cog(Profile(client))
    print(f'[DEBUG] {__name__} cog loaded')
```