ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інженерії програмного забезпечення

Пояснювальна записка
до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА ЧАТ-БОТА-ФАБРИКИ ДЛЯ ДОПОМОГИ МАЛОМУ БІЗНЕСУ В ТЕЛЕГРАМ МОВОЮ C# НА ПЛАТФОРМІ ASP.NET**»

Виконав: студент 4-го курсу, групи ПД-41 спеціальності

121 Інженерія програмного забезпечення

(шифр і назва спеціальності/спеціалізації)

Федоренко М.Л.

(прізвище та ініціали)

Керівник Негоденко О.В.

(прізвище та ініціали)

Рецензент

(прізвище та ініціали)

Київ - 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Бакалавр»
Спеціальність підготовки – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного
забезпечення

Негоденко О.В.

“ ” 2023 року

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТА

ФЕДОРЕНКА МАКСИМА ЛЕОНІДОВИЧ

(прізвище, ім’я, по батькові)

1. Тема роботи: “Розробка чат-бота-фабрики для допомоги малому бізнесу в телеграм мовою C# на платформі ASP.NET”

Керівник роботи: Негоденко О.В. к.т.н., доц.

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

Затверджені наказом вищого навчального закладу від «24» лютого 2023 року №26.

2. Строк подання студентом роботи “01” червня 2022 року
3. Вхідні дані до роботи:

Методи обробки та зберігання даних;

Інструменти з розробки програмного забезпечення: ASP.NET CORE 7.0, IDE JetBrains Rider 2023.1, JetBrains DataGrip 2023.1, Docker, Git, Linux

Server

Науково-технічні документації та література по розробці мікро сервісної архітектури.

Технічна документація АПІ по Telegram ботів

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
    4.1. Аналіз предметної області
    4.2. Технічне завдання
    4.3. Функціональні вимоги
    4.4. Нефункціональні вимоги
    4.5. Асоціативна мапа
    4.6. Діаграма класі використання
    4.7. Інструменти та засоби розробки програмного забезпечення
    4.8. Архітектура системи
    4.9. Опис програмного забезпечення
5. Перелік демонстраційного матеріалу
    5.1. Титульний слайд.
    5.2. Мета, об’єкт, предмет, наукова новизна дослідження.
    5.3. Актуальність.
    5.4. Аналіз аналогів.
    5.5. Технічні завдання.
    5.6. Програмні засоби та інструменти реалізації.
    5.7. Розробка архітектури.
    5.8. Реалізація програми.
    5.9. Висновки
    5.10. Апробація результатів дослідження.
    5.11. Кінцевий слайд.
6. Дата видачі завдання «25» лютого 2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	11.04-14.04	Виконано
2	Аналіз та дослідження існуючих аналогів	15.04-17.04	Виконано
3	Проектування системи	18.04-21.04	Виконано
4	Створення та тестування програмного рішення	21.04-05.05	Виконано
5	Підготовка розділу 1	05.05-07.05	Виконано
6	Підготовка розділу 2	07.05-09.05	Виконано
7	Підготовка розділу 3	08.05-11.05	Виконано
8	Вступ, висновки, реферат	11.05-12.05	Виконано
9	Розробка обов‘язкових демонстраційних матеріалів	12.05-15.05	Виконано
10	Попередній захист роботи та перевірка на плагіат	19.05-25.05	Виконано
11	Здача роботи	01.06	Виконано

Студент

______________ ______________

(підпис) (прізвище та ініціали)

Керівник роботи ______________ ______________

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи 73 с., 44 рис., 4 табл., 2 дод., 21 джерел.

ЧАТ-БОТ, ТЕЛЕГРАМ, МІКРОСЕРВІСИ, C#, ASP.NET CORE, API, REST, MULTITENANCY, WEBHOOK, BOTFATHER

Об’єкт дослідження - просування малого бізнесу за допомогою чат-бота в Telegram.

Предмет дослідження - Telegram чат-бот фабрика на базі мікросервісів з мультитенантною архітектурою.

Мета роботи – підвищити ефективність просування малого бізнесу за допомогою чат-бота-фабрики на платформі ASP.NET мовою програмування C# із використанням мікросервісної архітектури.

Методи дослідження – методи структурного аналізу і проектування, методи розробки програмного забезпечення, методи тестування програмного забезпечення, методи верифікації програмного забезпечення.

Для реалізації поставленої мети потрібно вирішити наступні завдання:

1. Проаналізувати існуючі сучасні аналогічні програмні забезпечення та виявити їх переваги та недоліки. І як результат створити порівняльну таблицю.
2. Розробити функціональні та нефункціональні вимоги до програмного забезпечення, основуючись на висновках раніше проаналізованих аналогів.
3. Дослідити технології, інструменти та програмні засоби для вирішення поставленої задачі.
4. Розробити програмне забезпечення для спрощення створення Telegram чат-ботів з урахуванням прийнятих раніше технологій та поставлених заздалегідь вимог.
5. Провести тестування програмного забезпечення

6. Пройти апробацію на Науково-технічних конференціях;

Практичне значення отриманих результатів: Даний продукт спрощує створення Telegram чат-ботів та допомагає малу бізнесу.

Для розробки застосунку використовуються платформа ASP.NET Core на мові програмування C#.

Галузь використання – підприємницька діяльність.

ЗМІСТ

ВСТУП ... 12
1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ... 14
1.1. Актуальність чат-ботів у бізнесі ... 14
1.2. SendPulse ... 17
1.3. ManyChat ... 20
1.4. Botpress ... 23
1.5. Flow XO ... 26
1.6. Таблиця порівнянь аналогів ... 29
2. ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ... 31
2.1. Технічне завдання ... 31
2.2. Функціональні вимоги ... 31
2.3. Нефункціональні вимоги ... 34
2.4. Асоціативна мапа ... 39
2.5. Діаграми класів використання ... 40
2.6. GOMS – аналіз інтерфейсу ... 41
3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ... 43
3.1. Інструменти та засоби розробки ... 43
3.1.1. Мова програмування C# ... 43
3.1.2. Платформа ASP.NET Core та мікросервіси ... 43
3.1.3. Apache Kafka ... 45
3.1.3.1. Apache Kafka у порівнянні з RabbitMQ ... 46
3.1.4. Docker ... 48
3.1.5. Система контролю версій Git ... 50
3.1.6. JetBrains Rider ... 50
3.1.7. JetBrains DataGrip ... 51
3.2. Архітектура системи ... 52
3.2.1. DDD та структура проєкту ... 52
3.2.2. Міжсервісна взаємодія ... 56
3.2.3. PostgreSQL та схема бази даних ... 58
3.2.4. Multi-tenant architecture ... 68
3.2.5. Система обробки команд - патерн програмування State ... 69
3.2.6. API клієнтського сервісу ... 74
3.3. Опис програми Kyoto Bot Factory ... 75
3.4. Тестування програмного забезпечення ... 80
ВИСНОВКИ ... 84

СПИСОК ЛІТЕРАТУРИ....85
ДОДАТОК А....87
ДОДАТОК Б....107

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API	Application Programming Interface
JSON	JavaScript Object Notation
DTO	Data Transfer Object
REST	Representational State Transfer
DDD	Domain-driven design
ОС	Операційна система
FAQ	Frequently asked questions
AI	Artificial intelligence
LINQ	Language Integrated Query
GC	Garbage Collector
UI	User interface

ВСТУП

Обґрунтування вибору теми та її актуальність: Сучасний світ не можливо представити без соціальних мереж, зараз людина проводить приблизно 30% свого часу гортаючи стрічку новин у Facebook, Telegram чи Instagram. Спираючись на дослідження Pew Research Centre з 21 країни 75% людей, які мають телефон надають перевагу саме текстовим повідомленням [1]. На це впливає багато факторів, в першу чергу зручність. В інтернеті люди можуть спілкуватися з ким завгодно з будь якої точки світу, головне потрібно мати підключення до інтернету. По друге менша інтимність, для багатьох молодих людей спілкування через телефон може бути дискомфортним та страшним, тому більшість людей віддають перевагу саме текстовим повідомленням. Їх можна відправити у будь який час, відредагувати, якщо це необхідно, прикріпити файл чи фотографію.

Об'єкт дослідження - просування малого бізнесу за допомогою чат-бота в Telegram.

Предмет дослідження – Telegram чат-бот фабрика на базі мікросервісів з мультитенантною архітектурою.

Мета роботи - підвищити ефективність просування малого бізнесу за допомогою чат-бота-фабрики на платформі ASP.NET мовою програмування C# із використанням мікросервісної архітектури.

Методи дослідження - методи структурного аналізу і проектування, методи розробки програмного забезпечення, методи тестування програмного забезпечення, методи верифікації програмного забезпечення.

Для реалізації поставленої мети потрібно вирішити наступні завдання:

1. Проаналізувати існуючі сучасні аналогічні програмні забезпечення та виявити їх переваги та недоліки. І як результат створити порівняльну таблицю.
2. Розробити функціональні та нефункціональні вимоги до програмного

забезпечення, основуючись на висновках раніше проаналізованих аналогів.

3. Дослідити технології, інструменти та програмні засоби для вирішення поставленої задачі.
4. Розробити програмне забезпечення для спрощення створення Telegram чат-ботів з урахуванням прийнятих раніше технологій та поставлених заздалегідь вимог.
5. Провести тестування програмного забезпечення
6. Пройти апробацію на Науково-технічних конференціях;

Практичне значення отриманих результатів: Даний продукт спрощує створення Telegram чат-ботів та допомагає малу бізнесу.

Для розробки застосунку використовуються платформа ASP.NET Core на мові програмування C#.

Практична значущість результатів: Розроблене програмне забезпечення спрощує створення Telegram чат-ботів та допомагає малому бізнесу розвиватися, адже вони зможуть швидше почати взаємодіяти зі своїми клієнтами через соціальну мережу.

Галузь використання – підприємницька діяльність.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Актуальність чат-ботів у бізнесі

Сучасний цифровий світ активно розвивається, і багато клієнтів очікують, що підприємства чи компанії будуть доступні 24/7. Вони вважають, що це дуже важливо, як і сама якість продукту. Можливість компанії підтримувати діалог з клієнтом у будь-який момент часу дозволяє підприємству утримувати свого клієнта протягом тривалого часу, а клієнту отримувати інформацію у той же момент, коли він у ній зацікавлений. Чат-боти служать рішенням для цих потреб клієнта. Вони можуть замінювати живий чат та інші форми спілкування, такі як електронні листи та телефонні дзвінки, і залишатися активними увесь час. Чат-боти можуть підтримувати декілька розмов одночасно і при цьому бути однаково ефективними. Більшість компаній вже залучають своїх клієнтів через соціальні мережі, а чат-боти можуть зробити цю взаємодію більш інтерактивною. Покупці рідко спілкуються з людьми в бізнесі, тому чат-боти відкривають канал спілкування, де клієнти можуть взаємодіяти без стресу від взаємодії з іншою особою. Спираючись на дослідження Pew Research Centre з 21 країни, 75% людей, які мають телефон, надають перевагу саме текстовим повідомленням [1]. Тому, якщо компанія надає можливість взаємодії саме таким способом, клієнт буде більш зацікавлений у співпраці з цією організацією. Також, чат-боти автоматизують деякі процеси, наприклад, постійних запитань чи більш логічні завдання по типу оформлення замовлення, перевірки статусу замовлення, оформлення заяв. Тим більше великі технологічні компанії, такі як Google, Apple і Facebook, розробили АПІ чат-ботів, які можуть обробляти такі послуги. Ця автоматизація процесу дає працівникам компанії час зосередитися на більш важливих завданнях і дозволяє клієнтам не чекати відповідей. Варто зазначити, що раніше організації покладалися на пасивну взаємодію з клієнтами та чекали, поки клієнт сам звернеться до компанії за послугами. Чат-боти вирішують і цю проблему, оскільки боти можуть

ініціювати розмови першими. Таким чином нагадуючи про компанії і її послуги. Що може заохотити колишнього клієнта прийти ще раз [2].

Тому не дивно, що у підприємців все частіше виникає потреба та бажання створити своїх власних чат-ботів, тому що це у перспективні покращить та прокачає їх бізнес. Але цей процес не є легким. Виникає багато проблем при реалізації цієї ідеї. Перш за все, це розробка самого бота, потрібно мати впевнені знання в області програмування для того, щоб самостійно розробити власного чат-бота. Це займає досить великий проміжок часу, якщо розробляти повноцінний функціонал. Наступна проблема це розгортання бота на серверах. Потрібно організувати оренду та подальше завантаження на хостинг свого рішення. Звичайно можна замовити розробку на outsource, але як правило, виникають труднощі у взаємодії з розробником і в подальшій підтримці бота, як додавання нового та сучасного функціоналу.

Для вирішення цієї задачі було розроблено чат-бота фабрики - це процес створення програмного забезпечення, яке надає можливість бізнес-клієнтам організувати свою роботу у месенджері Telegram без розробки бота власними силами. Цей новостворений бот буде мати різні функції, спрямовані на полегшення роботи та покращення результативності малого бізнесу. Головна задача даної роботи полягає у тому щоб надати підприємцю можливість швидко створити та налаштувати свого бота з усім необхідним базовим функціоналом через інтерфейс месенджера Telegram. Особливість такого підходу дозволить бізнесу одразу організувати роботу зі своїми клієнтами через власного чат-бота без великих зусиль. Все що потрібно це налаштувати бота під потреби підприємства та увімкнути функції що надаються рішенням. Фабрика має базові конфігурації ботів, де реалізований функціонал спрямований на базові потреби клієнта від чат-бота. Також, фабрика самостійно покриває обов'язки розгортання нового бота та надає можливість для взаємодії з ним одразу після активації.

Клієнтський бот, якого створило підприємство, повинен допомогти підтримувати взаємодію організації безпосередньо з клієнтом у будь-який момент

часу. Через нього вони можуть дізнаватися інформацію по типу FAQ, отримувати розсилку оголошень. Ці базові речі уже збільшать шанс продовження співпраці клієнта з підприємством і для цього не потрібно розробляти свої рішення з нуля.

Цільовою аудиторією бота-фабрики є люди, які активно користуються месенджерами. В основному можуть бути підприємці малого бізнесу, саме в напрямку їх інтересів було розроблене дане рішення. Ці люди зацікавлені у швидкому створенні невеликого Telegram боту для покращення взаємодії з клієнтом. Саме вони хочуть утримати якомога більше нових відвідувачів або спробувати їх зацікавити у продукті повторно, якщо вони забули чи втратили інтерес до нього.

Також, у фабриці можуть бути зацікавити людей, які займаються мистецтвом. Для творців важливо отримувати зворотній відгук від шанувальників про їх творчість або їм було б зручно викладати картини до каналу, щоб продемонструвати свої роботи усім охочим. Тому рішення досить універсальне у використані, можна налаштувати його під свої цілі.

Також можна розглянути звичайних користувачів як цільова аудиторія вихідного творіння бота фабрики. Ця аудиторія звичайні користувачі месенджера Telegram, що зацікавилися у створеному підприємством боті. Вони можуть користуватися ним і навіть не помічати що він базується під роботою фабрики.

У чат-бота-фабрики присутні стандартні обмеження, які встановлює сам Telegram для своїх ботів. Вони можуть трохи сповільнювати роботу, але вони не заважають комфортній взаємодії бота з користувачем при звичайних умовах. Надається 60 запитів в одну хвилину в один чат [3], це не критично для особистих повідомлень, але це може вплинути на спілкування з ботом в бесіди. Потенційно можуть бути обмеження на блокування деяких ботів, яких обслуговує фабрика, через те що вони можуть порушувати правила Telegram, наприклад почати робити спам повідомлень.

Розгортання нових ботів потребує ресурсів сервера на якому працює фабрика, при великих напливах можуть виникнути труднощі. Але архітектура фабрики

дозволяє балансувати та масштабувати систему.

Можна виділити декілька аналогічних програм, які розробляються у такому ж напрямку **SendPulse, ManyChat, Botpress, Flow XO.**

1.2. SendPulse

SendPulse - це інтегрована маркетингова платформа, яка пропонує широкий спектр інструментів для зв'язку з клієнтами, включаючи створення Telegram чат-ботів [4].

Переваги SendPulse для створення ботів:

- **Простота використання.** SendPulse має інтуїтивно зрозумілий і простий інтерфейс, що дозволяє легко створювати та налаштовувати Telegram-ботів без необхідності в програмуванні. Навіть користувачі без технічних навичок можуть швидко розпочати роботу з платформою. Цей інтерфейс представлений у вигляді онлайн конструктора на сайті.
- **Багатофункціональність.** SendPulse надає широкі можливості для створення розумних Telegram-ботів з автоматизованими відповідями, відправкою сповіщень, розсилкою повідомлень. Тут присутні базові логічні операції та послідовності дій. Можна налаштувати різні сценарії взаємодії з користувачами, щоб забезпечити персоналізовану та ефективну комунікацію.
- **Інтеграція з іншими каналами.** SendPulse підтримує інтеграцію з іншими каналами комунікації, такими як електронна пошта, SMS та Viber.
- **Аналітика та звіти.** SendPulse надає детальні аналітичні дані та звіти про активність вашого Telegram-бота. Є можливість отримувати інформацію про кількість підписників, кількість повідомлень і різного роду метрики для аналізу ефективності свого Telegram боту.
- **Інтеграція зі стороннім АПІ.** SendPulse надає можливість працювати зі зовнішніми запитами, обробляти цю інформацію на основі цього можна

налаштовувати поведінку та повідомлення чат-бота.

- **Life-chat.** Є можливість спілкуватися з користувачами від імені Telegram чат-боту, тобто підключатися до чатів з користувачами і вести персональну розмову.

Недоліки використання даного сервісу:

- **Обмеженість функціоналу.** Наскільки б не був зручний та потужний конструктор по створенню чат-боту, все таки він не може покрити усіх можливих ситуацій. Можливо для простих випадків використання, його буде досить, але якщо бот буде виконувати більш складні операції, за допомогою цього веб сервіс їх навряд чи вдасться покрити.
- **Вартість.** Так як SendPulse є комерційною платформою, вартість обслуговування впливає на можливості даного конструктора. Для деяких користувачів це може бути вагомим фактором не починати працювати з цим сервісом.
- **Залежність від SendPulse.** Мається на увазі, що неможливо або дуже проблематично перенести створеного бота на інші платформи. Якщо буде прийнято рішення використовувати інший аналог, то перенос усього функціоналу може зайняти великий проміжок часу.
- **Обмеження розробників.** SendPulse не надає API або можливість власноруч запрограмувати важкі логічні операції. Тобто всі можливості доступні лише з інтерфейсу конструктора.

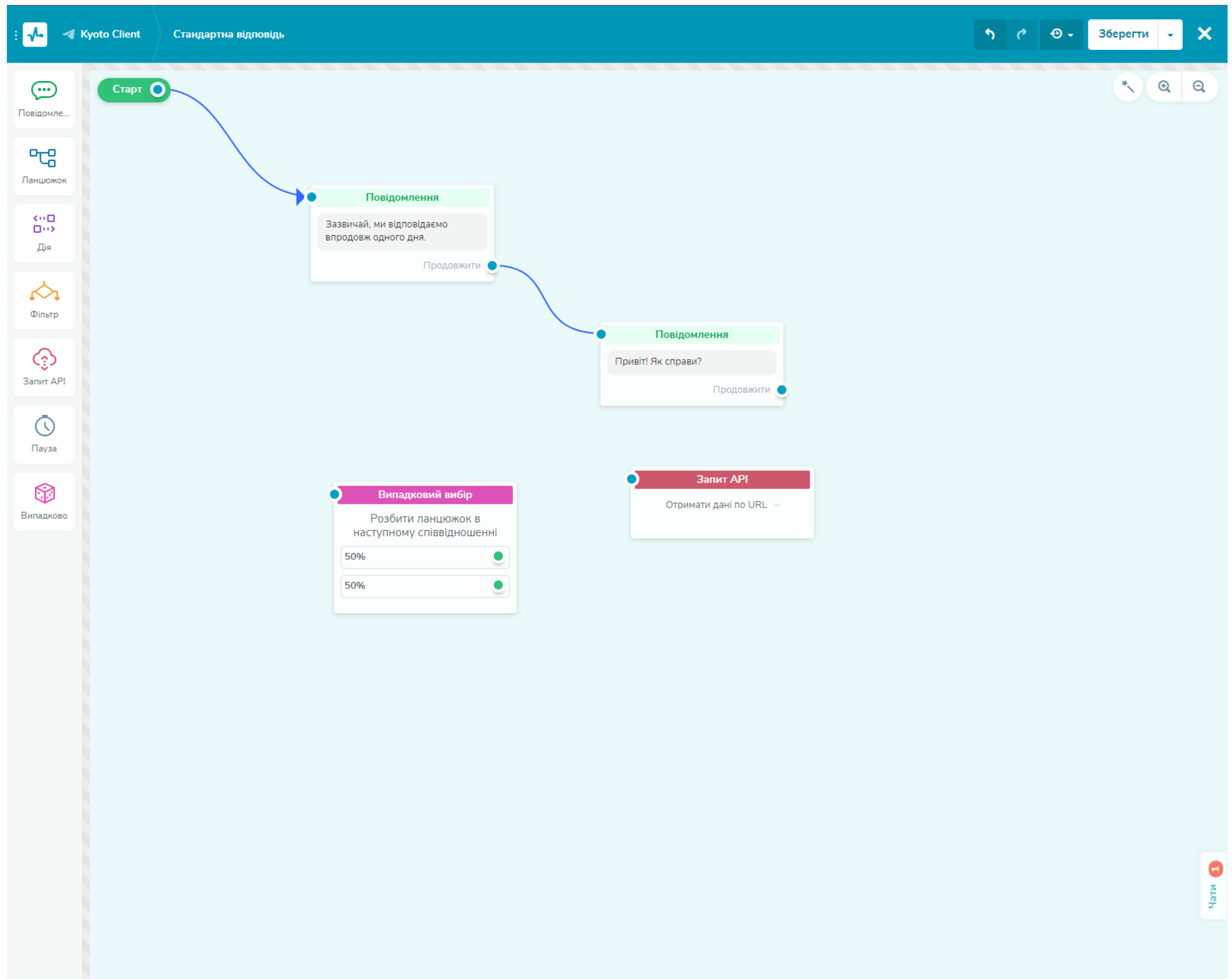


Рисунок 1.1 – Приклад конструктора ланцюжка подій SendPulse

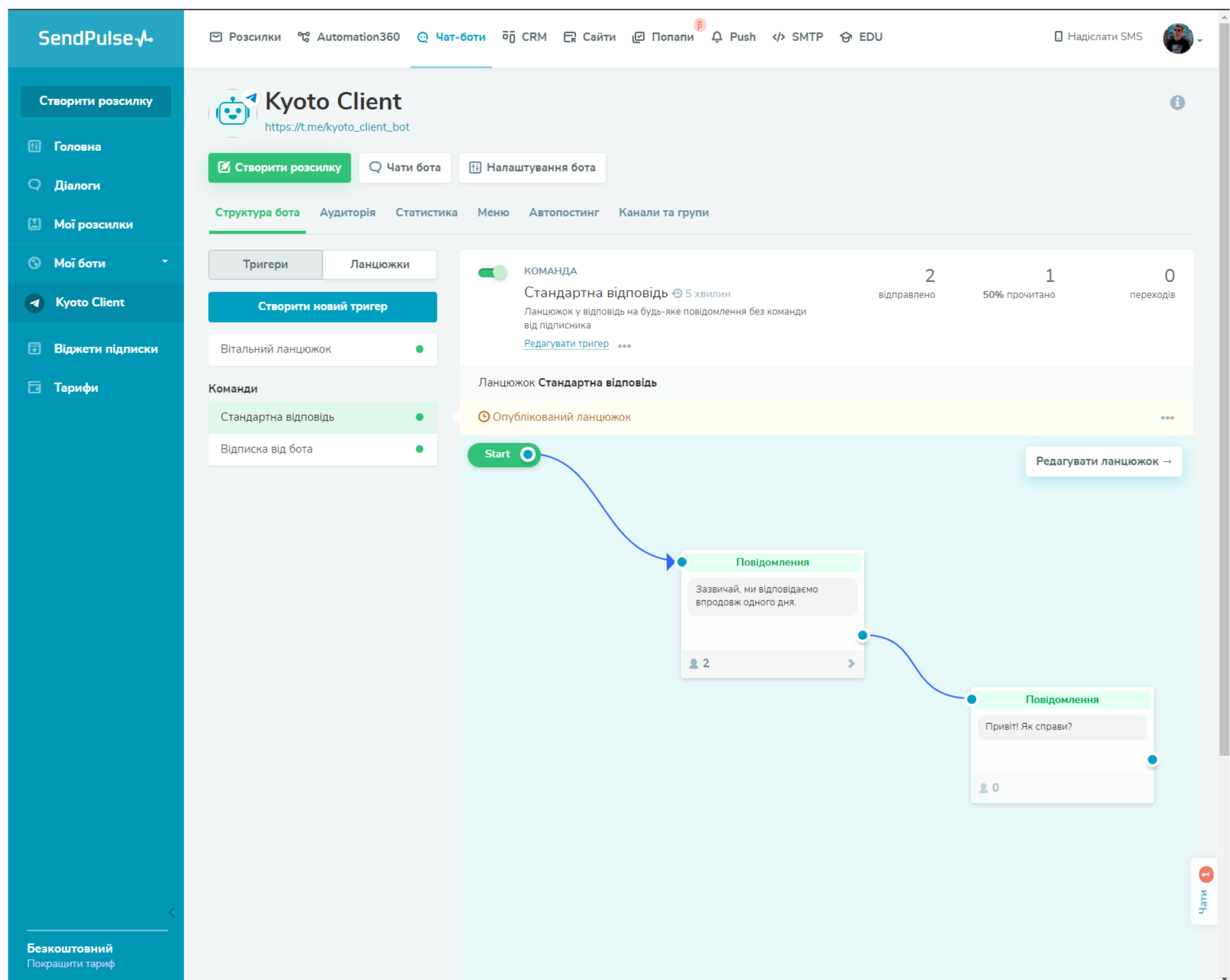


Рисунок 1.2 – Головне меню налаштувань власного бота у SendPulse

1.3. ManyChat

ManyChat - це платформа для створення чат-ботів, включаючи Telegram-ботів, без необхідності в програмуванні. Вона надає легкий у використанні інтерфейс та широкий набір функцій, що дозволяє створювати потужні та ефективні боти для спілкування з користувачами [5].

Переваги ManyChat для створення ботів:

- **Простота використання.** ManyChat має простий інтерфейс, який дозволяє користувачам спілкуватися зі своїми контактами так само, як у Telegram Messenger. На головному екрані програми відображається список ваших

контактів.

- **Аналітика та звіти.** ManyChat надає можливість отримувати цінну інформацію про ефективність роботи свого бота прямо в сервісі.
- **Багатофункціональність.** Надає можливості створювати логічні та пов’язані між собою ланцюжки подій з різноманітними тригерами для активації наступної дії бота.

Недоліки використання даного сервісу:

- **Обмеження шаблону.** Шаблонів невелика кількість. Крім того, шаблони не дуже налаштовуються, тому немає можливості створити свої унікальні для свого бізнесу.
- **Обмеженість функціоналу.** Як притаманно для більшості подібних сервісів, відсутня можливість максимального налаштування чат-боту, тому що єдиним інтерфейсом взаємодії та налаштування є тільки веб-конструктор.
- **Залежність від ManyChat.** Проблематично перенести створеного бота на інші платформи [6].

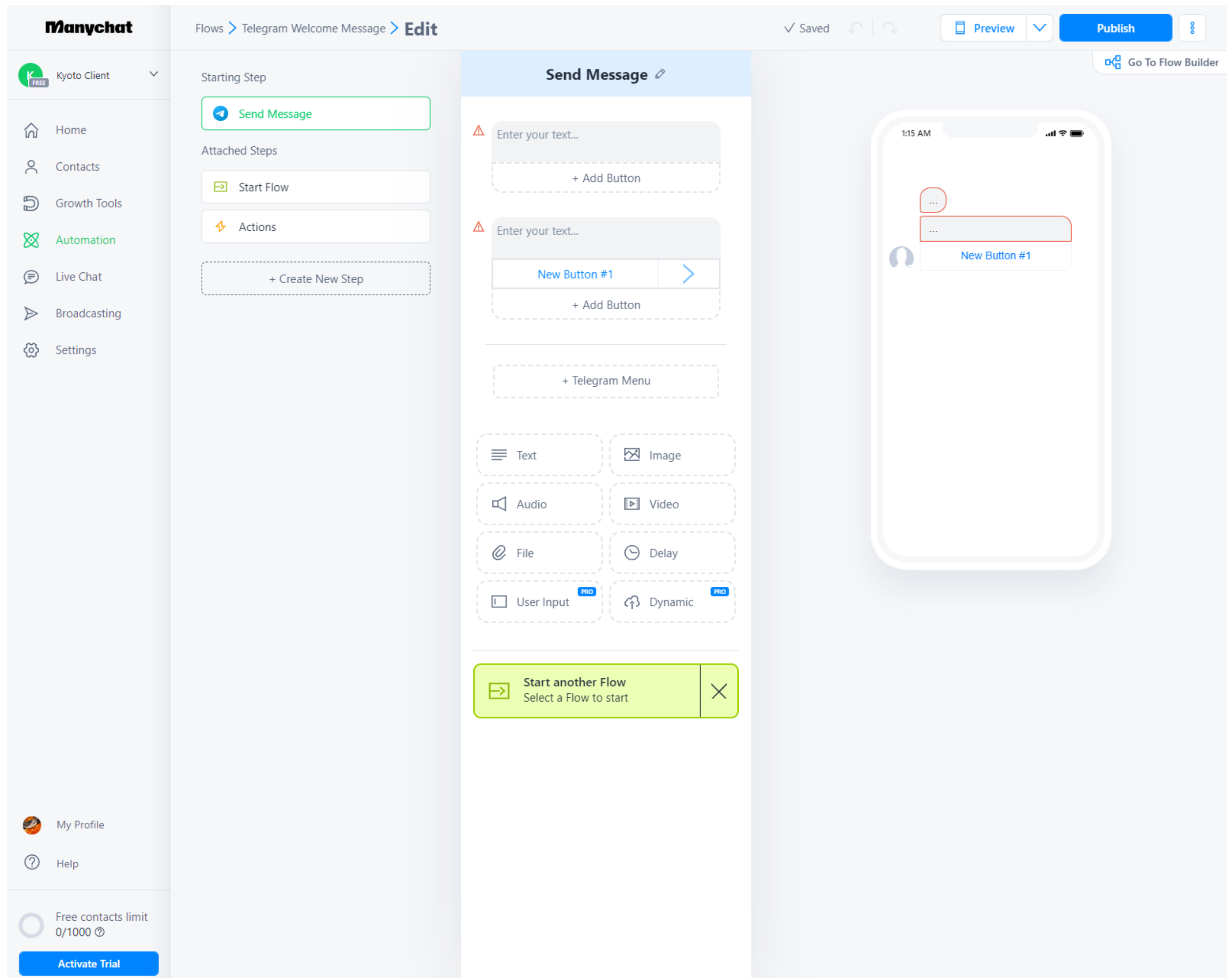


Рисунок 1.3 – Меню редагування автоматичного повідомлення в ManyChat

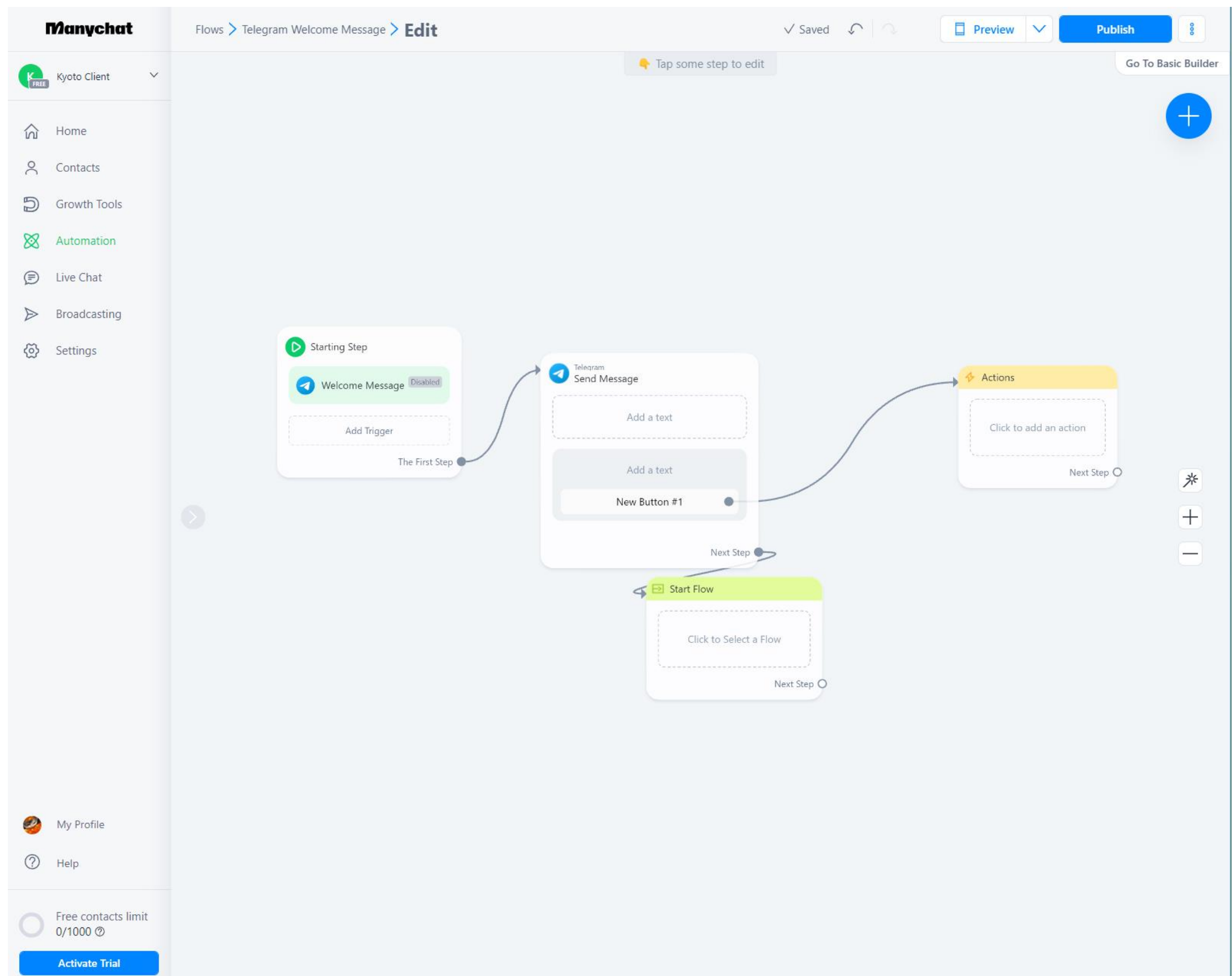


Рисунок 1.4 – Конструктор флоу в ManyChat

1.4. Botpress

Botpress - це відкрите програмне забезпечення для створення і керування чат-ботами. Він надає зручний інтерфейс та набір інструментів для розробки ботів, які можуть використовуватись у різних сферах, включаючи веб-сайти, месенджери, мобільні додатки і багато іншого. Також цей сервіс позиціонує себе як: "*Перший конструктор чат-ботів нового покоління на базі OpenAI*". Користувач може створити бота по типу ChatGPT для свого особистого використання чи бізнесу [7].

Переваги Botpress для створення ботів:

- **OpenSource**. Даний сервіс має відкрите програмне забезпечення та

розповсюджується під ліцензією MIT. Це надає великі можливості для розробників.

- **Гнучкість.** Даний сервіс надає дуже великі можливості з налаштування логіки чат-ботів. Тут можна легко створювати діалогові потоки, налаштовувати відповіді на повідомлення і налаштовувати інші параметри бота.
- **Аналітика.** Даний сервіс надає багату та основну аналітику по використанню створено Telegram боту. Клієнт може з легкістю дізнатися статистику по використанню бота. Наприклад, популярні запити, активність користувача.
- **AI**. Можливість працювати з OpenAI та конструювати бота з використанням цієї технології. Дуже потужна можливість для сучасних ботів.

Недоліки використання даного сервісу:

- **Складність використання.** Звичайно тут присутній стандартний конструктор ланцюжка подій бота, але для використання сервісу можуть знадобитися певні технічні знання, можливо навіть в області програмування на JavaScript, для налаштування деяких функцій.
- **Споживання ресурсів**. Botpress для своєї роботи використовує значних комп'ютерних ресурсів, особливо при великій кількості користувачів, це може навантажити сервера і погіршити роботу бота.
- **Безпека.** Сервіс немає вбудованих механізмів аутентифікації та авторизації, тому розробнику необхідно самостійно думати над засобами безпеки для свого бота.
- **Відсутня технічна підтримка.** Так, як даний сервіс OpenSource у нього відсутня спеціалізована технічна підтримка, через що можуть виникнути труднощі у розробці. Відповіді на такі питання потрібно буде шукати на різних форумах.

Рисунок 1.5 – Конструктор ланцюжка подій в Botpress

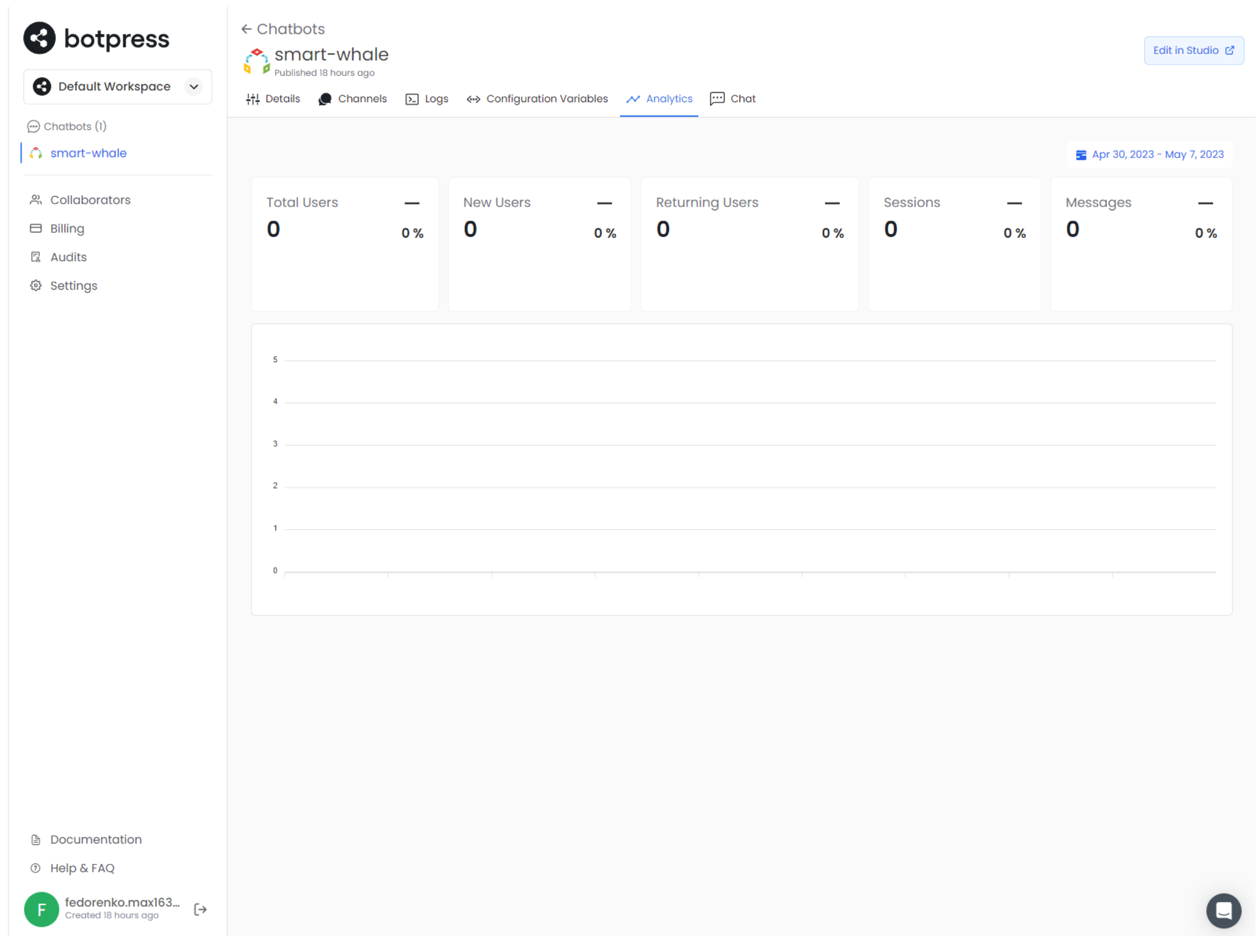


Рисунок 1.6 – Панель аналітики в Botpress

1.5. Flow XO

Flow XO — це потужний продукт для автоматизації, який дозволяє швидко та просто створювати неймовірні рішення для чат-ботів зі штучним інтелектом, які допомагають спілкуватися та взаємодіяти з клієнтами на різноманітних сайтах, програмах і платформах соціальних мереж [8].

Цей сервіс не є унікальним у порівнянні з іншими подібними аналогами. Flow XO також надає конструктор створення ланцюжків подій з стандартними процедурами розгалуження, обробки різних видів тригерів для подальшої активації дій бота. Також тут присутні заготовлені шаблони та можливості здійснювати

розсилки. Присутній LiveChat для спілкування з користувачами через самого бота. Є можливість підключати та інтегруватися з іншими соціальними сервісами, а також підключати CRM-системи.

Лише можна зазначити, що Flow XO має доволі не погану документацію та відео заняття де детально розповідається, як правильно користуватися сервісом, щоб комфортною створювати власного бота. Також тут є можливість звертатися по допомогу до експертів. І також конструктор ланцюжка побій відрізняється від аналогів, якщо більшість сервісів пропонували картку з блоками, які пов’язані стрілками то Flow XO ця послідовність будується з послідовних блоків-рядків, які можна додавати та налаштовувати у флоу.

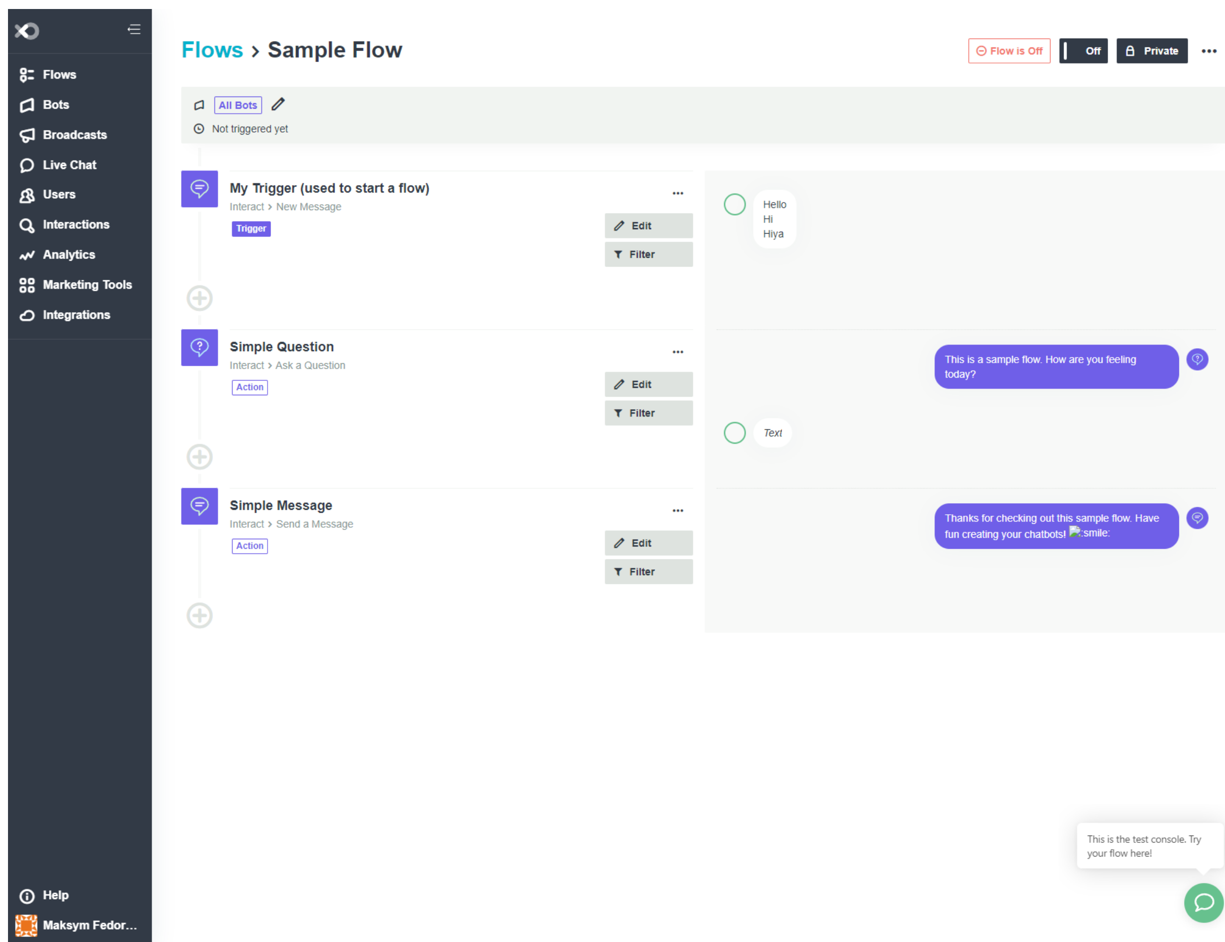


Рисунок 1.7 – Побудова флоу в Flow XO

З недоліком можна зазначити стандартні проблеми веб-серверів по створенню Telegram ботів:

- **Обмеженість.** Неможливо впроваджувати важкі та специфічні алгоритми роботи бота тому, що сервіс дозволяє налаштовувати бота лише через свій інтерфейс.
- **Безпека.** Так як бот розгортається на серверах Flow XO, неможливо організувати повну безпеку обробки даних клієнтів.
- **Залежність від платформи.** Якщо бот був створений за допомогою цього сервісу, його важко буде перенести на інші аналогічні платформи. Потрібно буде витратити певний час, щоб повторно налаштувати свого бота уже на базі іншого сервісу.

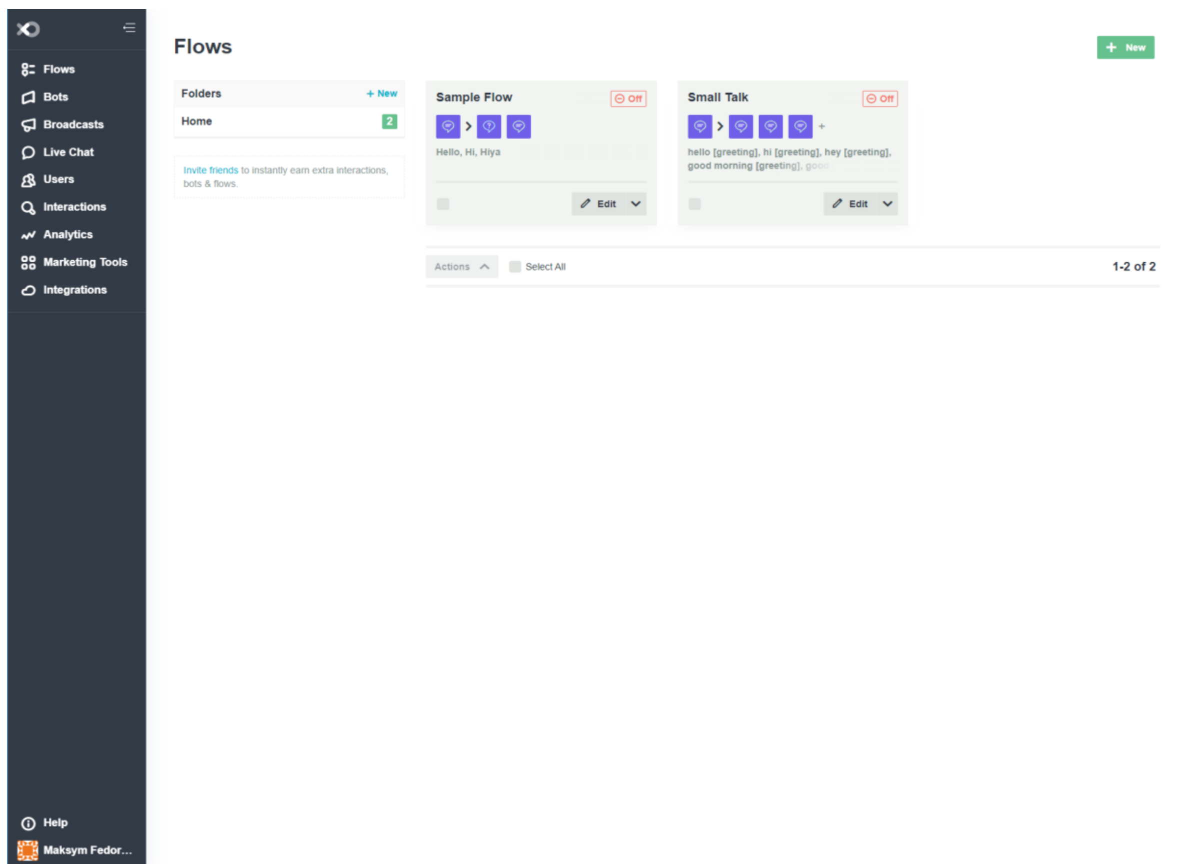


Рисунок 1.8 – Панель управління флоу в Flow XO

1.6. Таблиця порівнянь аналогів

Таблиця 1.1 – Характеристик сервісів для створення Telegram ботів

Показник	SendPulse	ManyChat	Botpress	Flow XO	Kyoto Bot Factory
Налаштування через зовнішні ресурси	+	+	+	+	-
Налаштування через Telegram чат	-	-	-	-	+
Налаштування за допомогою АПІ	-	-	-	-	+
Life Chat	+	+	-	+	-
Збір відгуків	-	-	-	-	+
Генерування QR-Code	-	-	-	-	+
Інтеграція з AI	+	+	+	-	
Аналітика та звіти	+	+	+	+	+
OpenSource	-	-	+	-	+

Продовження таблиці 1.1 – Характеристик сервісів для створення Telegram ботів

Інтеграція з іншими сервісами	+	+	+	+	-
Шаблонні функції	-	-	-	+	+
Обмеженість функціоналу через ціну	+	+	-	+	-
Навантаженість інтерфейсу	Середня	Середня	Висока	Середня	Низька
Технічна підтримка	Присутня	Присутня	Відсутня, тільки через форуми	Присутня	Відсутня, тільки через форуми
Розгортання на свої серверах	+	+	-	+	+

2. ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Технічне завдання

Розробити чотири мікросервіси на платформі ASP.NET Core з використанням мови програмування C#. Перший мікросервіс буде відповідати за отримання, а другий за відправлення повідомлень в Telegram. Третій мікросервіс, буде являти собою фабрику і буде створювати нових чат-ботів. Четвертий мікросервіс буде відповідати за обробку запитів до новостворених чат-ботів.

Всі мікросервіси повинні бути упаковані у контейнери і керуватимуться за допомогою програмного забезпечення Docker. Взаємодія між сервісами має здійснюватися за допомогою брокера повідомлень Apache Kafka.

Для реалізації запитів до Telegram зі сторони бота використовувати власна бібліотеку TBot.

Розробка мікросервісів повинна дотримуватися архітектури DDD (Domain-Driven Design). Крім цього, для підтримки кількох чат-ботів на одному екземплярі програмного забезпечення, бот буде реалізовувати архітектуру Multi-tenancy.

Як основу базу даних потрібно використовувати PostgreSQL. Для взаємодії з нею через програмний код, використовувати бібліотеку EntityFramework.

Середовищем розробки обрати JetBrains Rider - для програмування коду, JetBrains DataGrip - для роботи з базами даних, GrayLog - для візуалізації логів, Kafka-UI - для візуалізації Apache Kafka.

2.2. Функціональні вимоги

Дослідивши популярні Telegram боти для допомоги клієнтам, такі як Rozetka HelpCenter, mta.ua bot, Зоряна – Київстар було визначено основні функції, якими має

володіти чат-бот такого призначення. Як результат, можна навести такі функціональні вимоги, що стоються як чат-бот-фабрики так і бота що вона створює:

Авторизація. Користувач повинен мати можливість зареєструватися у чат-бот фабрики. Для цього у нього повинна бути можливість поділитися своїм контактом через спеціальні методи що надаються Telegram Bot API. Для реєстрації будуть отримані такі дані про користувача номер телефону, Telegram ід, ім'я та прізвище. Подальша аутентифікація повинна відбуватися автоматично через сам месенджер, клієнтові не потрібно робити повторних дій для реєстрації чи логіну.

Реєстрація та розгортання нового бота. Користувач повинен мати можливість зареєструвати свого Telegram бота за допомогою фабрики та запустити його. Клієнт має заздалегідь згенерувати токен свого нового бота через BotFather та надати його системі. Після цього повинні будуть виконані операції реєстрування нового бота в інфраструктурі та створення окремої бази даних для нього. Даний чат-бот буде прив'язаний до користувача по його внутрішньому Telegram ід, тому доступ до налаштувань та управління, буде мати лише творець.

Зміна текстів реєстрації для клієнтських ботів. Користувач, що зареєстрував свого Telegram чат-бота, повинен мати можливість зміни текст, який будуть бачити його клієнти, що користуються його ботом. Зміна тексту має відбувати у 3 етапи. Перший, змінна привітального повідомлення, другий, зміна повідомлення про успішну реєстрацію та третій зміна тексту про помилку у ході цього процесу.

Додавання та видалення повідомлень для розсилки. Користувач повинен мати можливість додати повідомлення для розсилки своїм клієнтам. У нього повинна бути можливість обрати тип події розсилки. На вибір є 2 види:

- Глобальна розсилка повідомлень усім клієнтам у певний момент часу.
- Повідомлення у відповідь на текст користувача. Цей механізм буде перевіряти вміст повідомлення що відправив користувач на наявність ключових елементів та слів. Якщо такі будуть присутні, бот автоматично відправить заздалегідь заготовлене повідомлення.

Також, користувач повинен мати можливість видалити заготовлене повідомлення.

Включення та виключення збору відгуків від клієнтів. Власник Telegram чат-бота, повинен мати можливість збирати відгуки від своїх клієнтів. У нього також повинна бути можливість переглядати їх. Зі сторони клієнтів, відгук можна залишити за допомогою спеціальної команди у клієнтського бота. Відгуки повинні містити шкалу оцінювання у вигляді зірочок, де 1 це поганий сервіс, а 5 це відмінні послуги. Також обов'язково повинна бути можливість залишити відгук анонімно.

Встановлення збірника відповідей на питання, які найбільш часто задають користувачі. Користувач може встановити FAQ для свого Telegram бота. У нього повинна бути завжди можливість його відредагувати. Даний функціонал повинен відображати раніше заготовлений текст за допомогою спеціальної команди у бота клієнта. Цей текст можна редагувати використовуючи форматування MarkdownV2, що підтримується Telegram.

Отримання статистики використання бота клієнтами. Творець Telegram чат-бота, повинен мати можливість отримувати не велику статистику використання його бота клієнтами. Таку як:

- Скільки було зроблено предзамовлень послуг
- Скільки клієнтів зареєструвалося
- Скільки клієнтів залишило відгук
- Середня оцінка по всім відгукам
- Коли був зареєстрований Telegram бот

Цю інформацію користувач повинен мати можливість отримати у будь який момент часу через меню налаштувань чат бота у фабриці.

Відключення чат-бота. Користувач повинен мати можливість вимкнути свого Telegram бота. Цей процес не знищує накопичену інформацію. Лише вимикає прослуховування оновлень з Telegram.

Видалення чат-бота. Користувач, що вже створив свого Telegram бота, повинен мати можливість видалити його повністю зі серверу фабрики. Цей процес знищує базу даних та усі інфраструктурні налаштування пов’язані з цим екземпляром Telegram чат-бота.

Відгук про роботу чат-бот фабрики. Користувач, що був раніше зареєстрований у фабриці. Повинен мати можливість залишити скаргу про роботу фабрики або пропозицію до того, як можна покращити цей систему або якого функціоналу не вистачає для покриття своїх потреб.

Передзамовлення послуги. Клієнт, який взаємодіє зі створеним ботом, повинен мати можливість передзамовити якусь певну послугу. У нього буде можливість оформити заявку та оповістити про це творця бота. Таким чином, підприємець що надає послуги, зможе заздалегідь підготуватися до приходу клієнта або отримати інформацію про те що, певна послуга цікавить таку то кількість клієнтів.

2.3. Нефункціональні вимоги

Продуктивність. Продуктивність програмного забезпечення Kyoto Bot Factory, залежить від характеристик сервера де розгорнуті мікросервіси та інфраструктура системи. Фабрика повинна бути побудована таким чином, щоб рішення можна було горизонтально масштабувати. Сервіси не повинні залежати один від одного, якщо притримуватися також реалізація наддасть можливість розміщувати певні сервісні модулі на різних серверах. А для збільшення потоку обробки можна дублювати мікросервіси, що дозволить збільшити обробку одночасних запитів до системи.

Фабрика повинна витримувати великі навантаження на систему, адже вона повинна обслуговувати не лише своїх користувачів, а клієнтів, які користуються створеними Telegram ботами.

Щодо відправки повідомлень у відповідь, існує штучне обмеження від самого т Telegram. Тому система здатна відправляти лише 20 повідомлень у хвилину в один чат Telegram.

Характеристика обробки навантажень з урахуванням одного серверу та одного екземпляру кожного мікросервіса:

Таблиця 2.1 – Характеристика обробки навантажень

Характеристика	Кількість запитів/Одиниця часу
Обробка повідомлень від користувачів фабрикою	1000/хвилину
Обробка повідомлень від користувачів створених ботів	2500/хвилину

Сумісність. Сервіси повинні підтримувати наступні показники сумісності: Запити до клієнтського сервісу повинні відбувати за допомогою HTTPS запитів. Та використовувати формат JSON для передачі об'єктних даних до системи.

- Система повинна бути сумісна з операційною ОС Linux
- Система повинна підтримувати да бути сумісною з такими бібліотеками, як TBot, Newtonsoft.Json, EntityFramework

Взаємодія з програмним забезпеченням. Взаємодія повинна відбуватися безпосередньо через інтерфейс чату Telegram. Чат-бот фабрика повинна надавати можливість користувачу користуватися усім функціоналом системи, використовуючи об'єкти взаємодії, які надає Telegram для чат-ботів. Побудова роботи повинна бути максимально простою для розуміння. У користувача має бути завжди можливість відмінити обробку тої чи певної команди за допомогою */cancel*.

Також для налаштування свого бота, користувач повинен мати можливість виконувати АПІ запити до системи і впливати на роботу свого Telegram бота.

В свою чергу, АПІ що надає мікросервіс для роботи з клієнтськими ботами повинен відповідати наступним вимогам:

- АПІ притримуються принципу REST
- До кожної кінченої точки має бути написати відповідна документація.
- Запити повинні підтримувати та повертати стандартні HTTP статус коди.

Надійність. Система повинна обробити будь який не передбачений сценарій поведінки. У разі звалу обробки команди, повинно з’явитися повідомлення для клієнта про збій команди, але при цьому система повинна продовжувати обробляти запити та взаємодіяти з користувачем.

Система при обробці команд, повинна перевіряти вхідні дані і у разі невідповідності, показувати клієнтову відповідне повідомлення та надавати можливість на повторну спробу, при цьому пояснити у чому саме помилився клієнт.

Система повинна гарантувати правильно роботу з пам'яттю та не допускати її витоки. Усі з’єднання з базами даних після закінчення роботи повинні бути роз'єднанні та очищені.

Доступність. Система повинна бути доступна завжди. У разі падіння одного з мікросервісів, його обов’язки повинні перейти на інший доступний сервіс. У разі збою усього сервера, максимальний downtime повинен складати не більше 1 години. Під час оновлення системи, максимальний час downtime не повинен перебільшувати 30 хвилин.

Безпека системи. Система не повинна зберігати повідомлення користувачів, окрім відгуків, на які вони надали згоди.

- Токен Telegram боту повинен зберігатися у базі у зашифрованому вигляді, ключі розшифрування відомий лише мікросервісу що відповідає за надіслання повідомлень.
- Безпека аутентифікації та авторизації лежить на стороні месенджера Telegram.

- API запити, які можна здійснювати до клієнтського сервісу повинні бути захищені авторизацією та потребувати токен доступу, який надає бот фабрика. Токен повинен діяти не більше 4 годин.

Також, система повинна підтримувати такі стандарти безпеки, як ISO/IEC 27000-27008.

Можливість модифікації. Система повинна підтримувати мікросервісну архітектуру та бути розбитою на певні модульні сервісі зі своєю зоною відповідальності. Це дозволить модифікувати та покращувати певні модулі не впливаючі на інші.

Також система повинна притримуватися патерн архітектурного дизайну DDD для розбиття програмного забезпечення на певні шари відповідальності. І що дуже важливо підтримувати такий принцип побудови коду як SOLID.

Тестування. Мінімальне покриття системи Unit тестами повинно складати не менше 10%. Також система повинна мати інтеграційні тести при запуску проєкту. Покриття ручними тестами має складати не менше 90%.

Масштабованість. Система повинна повністю підтримувати горизонтальне масштабування.

Системні потреби. Для роботи сервісів необхідна ОС Linux на базі дистрибутива Debian 11. Мінімальні системні потреби для забезпечення функціонування системи:

Таблиця 2.2 – Системні характеристики

Апаратна складова		
	Мінімальні	Максимальні
Процесор	4x Xeon 64bit, 2.1 GHz	16x Xeon 64bit, 3.5 GHz
ОЗУ	8 Gb	16 Gb

Продовження таблиці 2.2 – Системні характеристики

Диск	20 Gb	50 Gb
Програмна складова		
	Версія	
PostgreSQL	13	
Apache Kafka	7.4.0	
Redis	7.2-rc	
Nginx	1.18	
Docker	20	
Kyoto.Telegram.Receiver	latest	
Додаткове програмне забезпечення для розробників		
Kafka-UI	latest	
Graylog	4.2	
MongoDb	4.4.19	
Elasticsearch	7.10.2	

І також обов'язково необхідний доступ до широкосмугового інтернету.

2.4. Асоціативна мапа

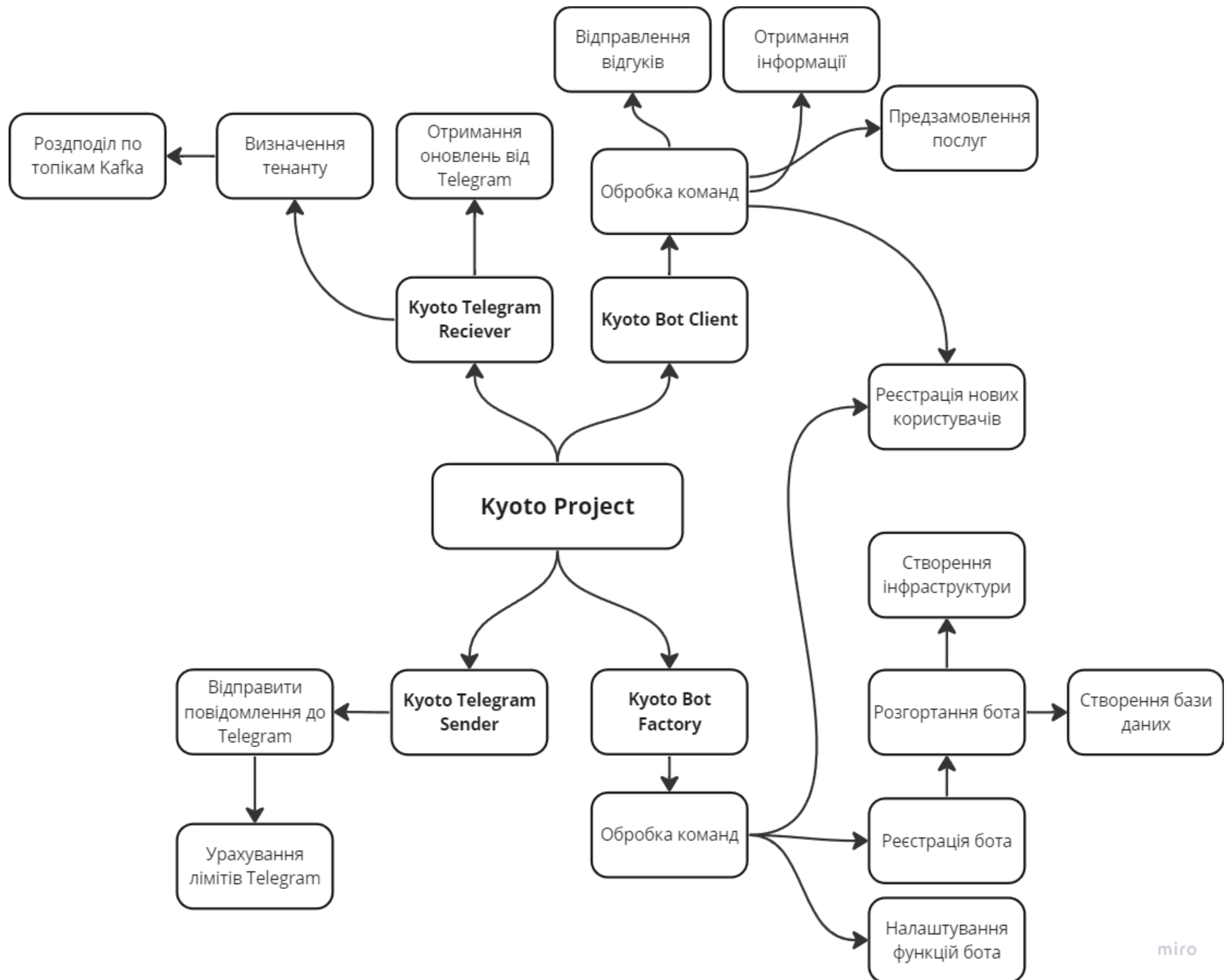


Рисунок 2.1 - Асоціативної мапа - архітектурне представлення функціоналу проєкту

2.5. Діаграми класів використання

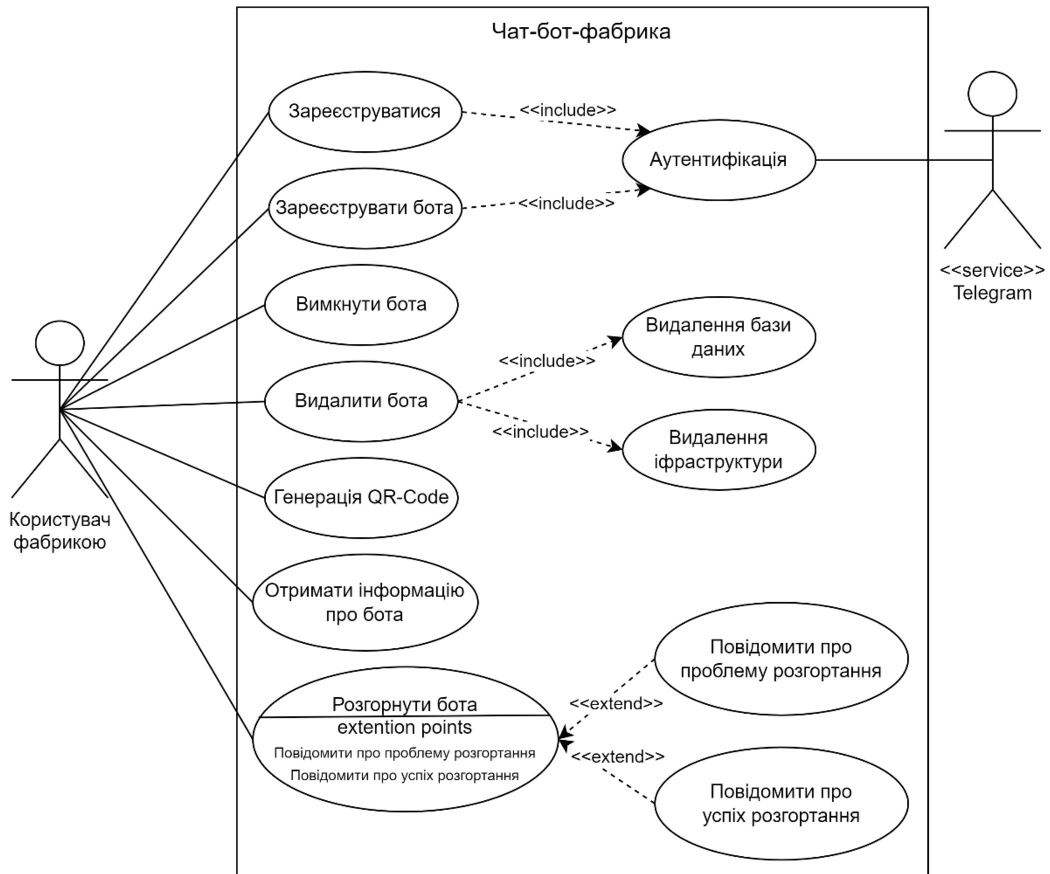


Рисунок 2.2 - Діаграма використання – Користувач фабрикою

Рисунок 2.3 - Діаграма використання – Клієнт

2.6. GOMS – аналіз інтерфейсу

GOMS - це спеціальна модель, яка допомагає у спостереженні за взаємодією людини з комп’ютером. Цей термін випливає з книги «Психологія взаємодії людини з комп’ютером», а саме автори вносять таке поняття, як «набір цілей, набір операторів, набір методів для досягнення цілей і набір правил вибору для вибору з конкуруючі методи для досягнення цілей.". Цей метод дуже корисний для якісного прогнозування того, як людина буде користуватися запропонованим інтерфейсом системи.

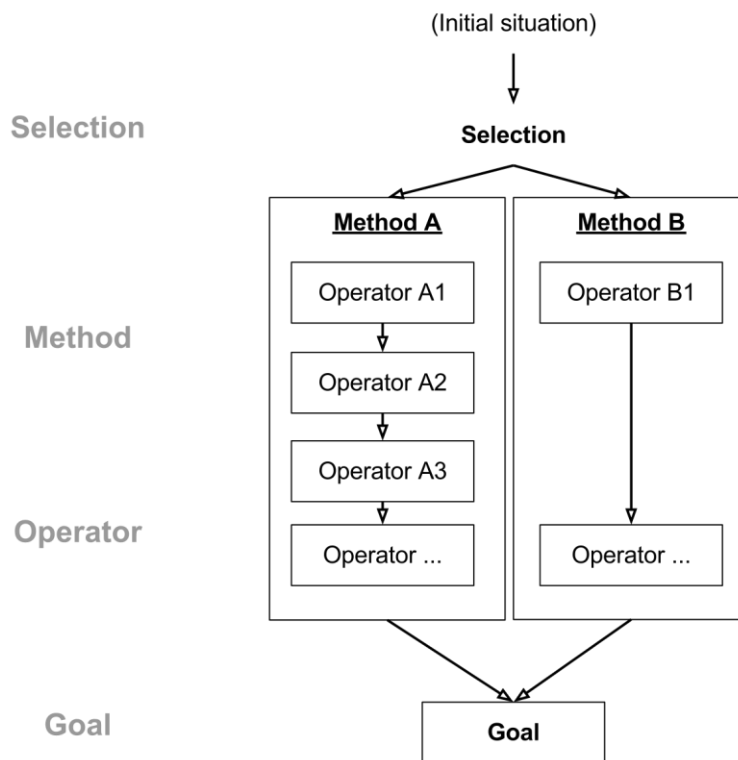


Рисунок 2.4 - Концепції моделі GOMS та їхні взаємозв’язки [25]

Для аналізу було обрано такі дії з клавіатури та миші притаманні для роботи з інтерфейсом:

- **K** – Натиск на клавішу (приблизно 0.3 секунди)

- **P** – Вказати мишею конкретну ціль (приблизно 1.4 секунди (звичайно тут вливає DPI миші, яку кожен користувач налаштовує для себе особисто))
- **H** – Перенос руки на клавіатуру або миш (приблизно 0.5 секунд)
  Час дії було проаналізовано в ході особистого дослідження.
- **M** – Роздум.
- **R** – Очікування відповіді від системи.

Для покращення ефективності використання саме розробленого рішення було проведено GOMS-аналіз. Для порівняння, можна обрати систему SendPulse та оцінити, скільки часу потрібно користувачеві для того щоб налаштувати свого Telegram чат-бота на відправку повідомлення у певний момент часу, за результатами експерименту можна зробити висновок, що для реалізації цієї функції з умовою, що користувач уже знаходився у візуальному конструкторі, можна приблизно порахувати, що на виконання цих дій у нього було витрачено приблизно **24.4** секунди. У той час, при використанні чат-боту фабрики час задіяний на таку ж саму операцію оцінюється у **14.8** секунди, що у **1.64** рази швидше чим у SendPulse. Це дослідження не є абсолютним, тому що потрібно враховувати людській фактор, а також швидко дії системи.

Також, з пріоритетних відмінностей використання інтерфейсу можна зазначити, що бот працює у вигляді діалогу, користувачеві не потрібно активно пересувати миш на великі дистанції тому, що наступне повідомлення буде з'являтися знизу екрану, що доволі зручно для взаємодії з віртуальною клавіатурою, особливо, коли користувач знаходиться у програмі з мобільного девайсу.

3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Інструменти та засоби розробки

3.1.1. Мова програмування C#

C# є доволі сучасною, типобезпечною та об’єктно орієнтованою мовою програмування. Вона є частиною .NET платформи, що розроблена корпорацією Microsoft для створення широкого спектра програмного забезпечення, включаючи веб-сервіси, мобільні програми та додатки для робочого столу.

Основні особливості C# включають:

- **Сильна типізація.** C# вимагає від програмістів вказувати конкретні типи даних, які використовуються, що допомагає уникнути помилок, пов'язаних з несумісними типами даних.
- **Об'єктно-орієнтоване програмування (ООП).** C# підтримує основні концепції ООП, такі як наслідування, поліморфізм, інкапсуляція, і абстракція.
- **Підтримка LINQ**. Ця технологія дозволяє програмістам виконувати запити до даних прямо з мови C#, що підвищує продуктивність і забезпечує типобезпечність.
- **Автоматичне керування пам'яттю.** C# використовує сміттєвий збирач GC, що автоматично звільняє неактивну пам'ять, знижуючи ризик витоку пам'яті.
- **Багатопоточність.** C# має розширену підтримку багатопоточності та асинхронності, що дозволяє створювати високопродуктивні програми [17].

3.1.2. Платформа ASP.NET Core та мікросервіси

Telegram бота можна реалізувати двома способами.

Перший, це консольна програма. Цей підхід зручний у тому випадку, якщо потрібно швидко розробити невеликого Telegram бота без складної взаємодії та функціоналу. Такого бота простіше запускати на серверах або локально. Він набагато

менше використовує ресурсів, адже не реалізовує усі шари абстракцій та інфраструктури.

Другий підхід це розробка мікросервісів. Він підходить краще для ботів, які будуть взаємодіяти з великою кількістю клієнтів. Мікросервіси зручно масштабувати та розподіляти ресурси між серверами. Також, цей підхід може обробляти великий обсяг складних запитів. Бекенд краще підходить для інтеграції з іншими сервісами, наприклад отримувати оновлення з Telegram можна через Webhook, що набагато спрощує розробку системи та підвищує ефективність системи. Натомість, консольний додаток використовує long polling, що потребує постійного підключення до клієнта до сервера через що можуть виникати затримки при отриманні інформації і це також впливає на масштабування проєкту.

Для розробки чат бота фабрики було обрано другій підхід, реалізацію через мікросервіси тому, що бот повинен обробляти та обслуговувати не тільки себе, а також і інших ботів, що будуть працювати паралельно з фабрикою. Через що потрібно зробити великий акцент на архітектуру складову та оптимізацію роботи системи.

Сервіси були розроблені на платформі ASP.NET Core 7.0, мовою програмування C# 11.0.

ASP.NET - це дуже популярна платформа для розробки та створення веб-додатків під платформою .NET. ASP.NET Core - це версія продукту з відритим програмним кодом. Її особливість у тому що вона працює на macOS, Linux і Windows. Ця платформа набагато продуктивніша у порівнянні з Java Servlet та Node.js [9].

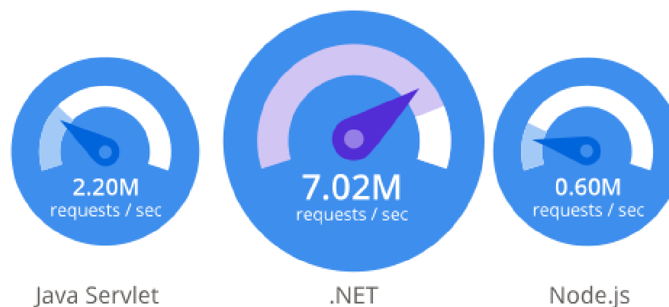


Рисунок 3.1 Дані базуються на офіційних тестах TechEmpower Round 21 [10].

З особливостей платформи можна зазначити що ця платформа, як уже було зазначено раніше, досить високо продуктивна завдяки спеціальній оптимізації, це забезпечує швидке виконання веб-застосунків. Під капотом він використовує обробник HTTP Kestrel, який до того ж може працювати в асинхронному режимі. Ця можливість забезпечує швидку обробку багатошарових запитів до веб-застосунку.

Варто зазначити, що дана платформа успішно підтримує горизонтальне масштабування. Це дозволяє розподіляти навантаження на декількох серверах, що навіть можливість додаткам мати високу доступність навіть при великих навантаженнях. Цей аспект надзвичайно важливий при розробці великих багато сервісних проєктів.

Досить важливою складовою платформи являється безпека. ASP.NET Core надає потужні механізми для захисту веб-застосунків. Він містити вбудовану підтримку аутентифікації та авторизації.

3.1.3. Apache Kafka

Взаємодія між сервісами відбувається за допомогою брокера повідомлень Apache Kafka. Це програмне забезпечення з відкритим вихідним кодом, яка дозволяє розробляти програми, керовані подіями в реальному часі [11]. Один сервіс може виробити повідомлення (подію), певний структурований контракт, інший же сервіс

буде виступати у ролі споживача та очікувати можливі повідомлення цього типу. У разі надходження події до обробника, він починає виконувати поставлену перед ним задачу. Цей принцип необхідний для реалізації мікросервісної архітектури та горизонтального масштабування.

Рисунок 3.2 Приклад повідомлення Кафки. Подія відправки повідомлення до Telegram

3.1.3.1. Apache Kafka у порівнянні з RabbitMQ

Особливості Apache Kafka можна яскраво виділити порівнюючи її з теж досить популярним брокером повідомлень RabbitMQ. Хоч по суті своїй вони однакові, але вони мають кардинально інші точки зору щодо реалізації [12].

Таблиця 3.1 Порівняльна характеристика між Kafka та RabbitMQ з інтернет порталу Simplilearn [12]

Характеристика	**RabbitMQ**	**Kafka**
Продуктивність	4000-10000 повідомлень в секунду	1 мільйон повідомлень в секунду
Зберігання повідомлень	На основі підтвердження	На основі політики (наприклад, 30 днів)
Тип даних	Трансакційний	Оперативний
Споживчий режим	Розумний брокер/тупий споживач	Тупий брокер/розумний споживач
Топологія	Тип обміну: прямий, розгорнутий, тематичний, на основі заголовка	На основі публікації/підписки
Випадки використання	Прості варіанти використання	Великі дані/висока пропускна здатність

Додаткові особливості:

- **Потужність системи при навантаженнях.** Основна ціль Kafka це забезпечення високої пропускної здатності та масштабованості. Вона дозволяє легко додавати нові вузли та розподіляти навантаження між ними. RabbitMQ також

масштабується, але може виявитися менш ефективним у ситуаціях з великим обсягом даних.

- **Принцип збереження даних.** Kafka зберігає повідомлення у вигляді журналів на диску. Це забезпечує високу стійкість до відказів. Також такий підхід дозволяє забезпечити послідовне зчитування даних з високою пропускною здатністю. RabbitMQ використовує ерланг-засновану базу даних Mnesia для зберігання даних, що може бути менш ефективним для великих обсягів даних.
- **Реплікації.** Kafka підтримує реплікацію на рівні топіків, що забезпечує високий рівень стійкості до відмов. RabbitMQ також підтримує реплікацію, але її реалізація менш гнучка.
- **Гарантії доставки повідомлень.** Kafka пропонує гарантію "*at least once*" та "*exactly once*" доставки повідомлень, що дозволяє забезпечити доставку повідомлень без втрати даних. RabbitMQ також підтримує різні рівні гарантії доставки, але вони можуть бути менш ефективними в ситуаціях з високим обсягом даних.
- **Потокова обробка даних.** Kafka підтримує потокову обробку даних за допомогою бібліотеки Kafka Streams, що дозволяє розробникам легко створювати розподілені потокові застосунки. RabbitMQ немає аналогічної функціональності "з коробки", але може бути інтегрований з іншими рішеннями для потокової обробки.

3.1.4. Docker

Docker - це програмне забезпечення з відкритим вихідним кодом, яка дозволяє розробникам автоматизувати розгортання, масштабування та управління додатками в контейнерах. Контейнери - це легкі, портативні та самодостатні середовища, які пакують додатки та їх залежності разом. Так звані контейнери використовують ядро хостовий ОС і для запуску програмного додатку потребують лише необхідних бібліотек та залежностей, що робить образи легшими, у порівнянні з традиційними

віртуальними машинами. Особливість такого підходу до розгортання проєкту дозволяє дуже легко переносити програмне забезпечення на будь-яке середовище, головне, щоб на ньому був встановлений Docker [18].

Також, варто зазначити що програмне забезпечення підтримує контроль версій, це додає розробникам гнучкості, адже з’являється можливість відкочуватися до попередніх версій.

Таким чином, Docker спрощує розробку, розгортання та управління додатками. Він дозволяє розробникам створювати, ділитися та запускати додатки послідовно в різних середовищах, що робить його важливим інструментом для сучасної розробки програмного забезпечення.

```
1   version: "3.9"
2
3   services:
4     kyoto.bot.factory:
5       container_name: kyoto.bot.factory
6       image: gsmelford/kyoto.bot.factory:latest
7       restart: always
8       ports:
9         - "7110:80"
10      environment:
11        - KyotoBotFactorySettings:BaseUrl=https://app.kyotobot.org
12        - KyotoBotFactorySettings:ReceiverEndpoint=/receiver/api/update
13        - BotTenantSettings:Key=main
14        - BotTenantSettings:Token=
15        - KafkaBootstrapServers=kafka
16        - DatabaseSettings:Host=database
17        - DatabaseSettings:Port=
18        - DatabaseSettings:Database=
19        - DatabaseSettings:Username=
20        - DatabaseSettings:Password=
21        - Serilog:MinimumLevel:Default=Information
```

Рисунок 3.3 - Приклад налаштування сервісу у Docker Compose

3.1.5. Система контролю версій Git

Git - це дуже розповсюджена розподілена система контролю версій, яка дозволяє розробникам ефективно керувати та відстежувати зміни у своєму коді. Цей інструмент необхідний для синхронізації роботи у тому випадку коли над проєктом працює 2 та більше людей. Також Git дає можливість розгалуження, що дозволяє розробникам працювати над новими функціями або виправленнями помилок в окремих гілках, не впливаючи на основний код. Також із особливостей даного інструменту можна зазначити що Git є безкоштовним і має відкритий вихідний код, що означає, що він активно розробляється і підтримується великою спільнотою зацікавлених в ньому людей, що робить його високо надійним і багатофункціональним [19].

3.1.6. JetBrains Rider

JetBrains Rider - це кросплатформне інтегроване середовище розробки (IDE) для розробки .NET, розроблене компанією JetBrains. Воно підтримує декілька мов програмування, включаючи C#, F# та VB.NET, і орієнтоване на різні платформи, такі як .NET Framework, .NET Core та Xamarin.

Основні переваги Rider:

- **Продуктивність.**
- **Кросплатформеність.** Rider доступний для Windows, macOS та Linux
- **Інтелектуальний редактор коду.** У Rider пропонує розширений аналіз коду, контекстно-орієнтоване завершення коду та рефакторинг.
- **Отладка.** Rider має потужний відладчик з підтримкою точок зупинки, годинників та оцінок змінних, що полегшує діагностику та виправлення проблем у коді.
- **Інтегрований контроль версій.**
- **Розширюваність.** Rider підтримує широкий спектр плагінів.
- **Підтримка баз даних.** Rider включає інтегрований клієнт баз даних.

- **Модульне тестування.** Ця IDE надає вбудовану підтримку популярних фреймворків для тестування .NET, таких як NUnit, xUnit та MSTest.

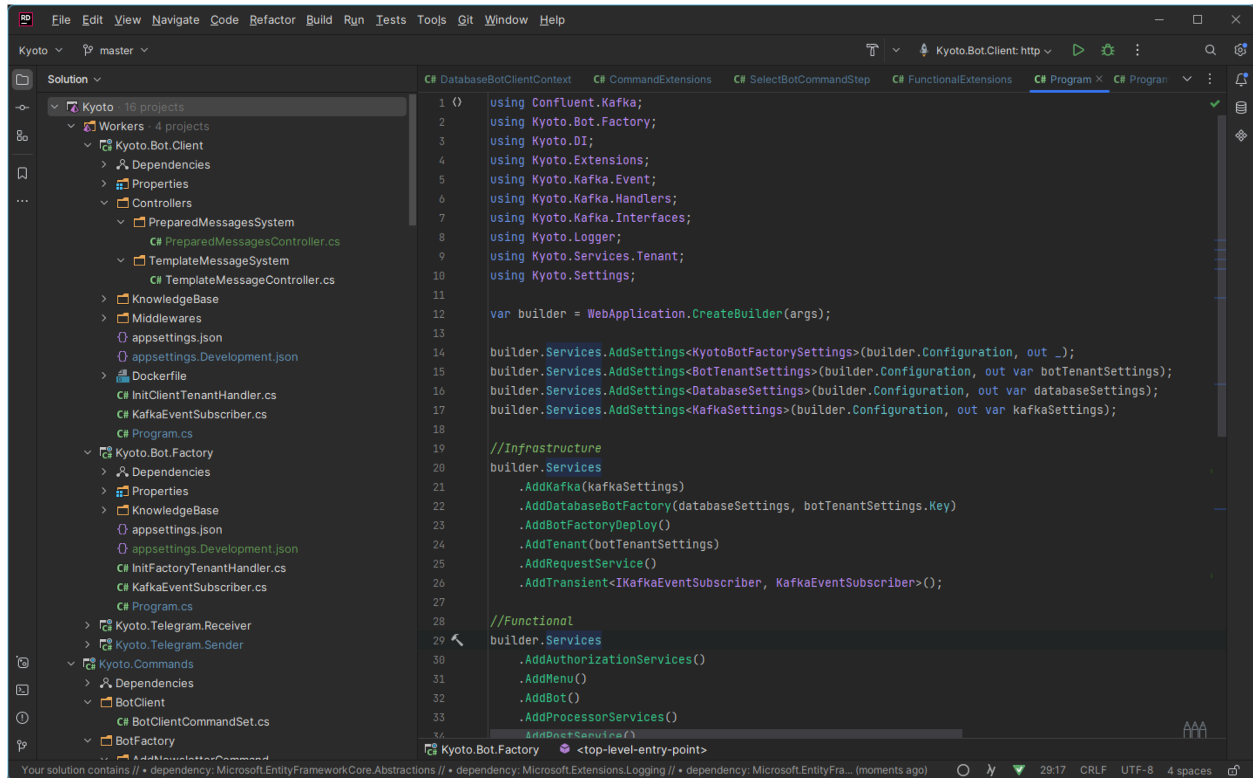


Рисунок 3.4 - IDE JetBrains Rider

3.1.7. JetBrains DataGrip

JetBrains DataGrip - це крос-платформне інтегроване середовище розробки (IDE), створене для взаємодії з базами даних та SQL. Воно підтримує широкий спектр систем управління базами даних (СУБД), включаючи MySQL, PostgreSQL, Oracle, Microsoft SQL Server та багато інших.

Основні переваги DataGrip:

- **Інтелектуальний редактор SQL.**
- **Крос-платформеність**. DataGrip доступний для Windows, macOS і Linux.
- Підтримка декількох рушіїв.
- **Керування схемами баз даних.** DataGrip надає інструменти для управління та редагування схемам баз даних.

- **Виконання та аналіз SQL-запитів.**
- **Розширюваність.** DataGrip підтримує широкий спектр плагінів.

3.2. Архітектура системи

Надзвичайно важливим аспектом у розробці програмного забезпечення відіграє продумування архітектури. Вона визначає структуру проєкту, компоненти, сервіси та їх взаємозв'язки, що має великий вплив на підтримку, модифікацію та подальшу стабільність програми.

Добре продумана архітектура полегшує масштабування програми відповідно до її розвитку. Якщо проєкт правильно продумано з дотриманням усіх популярних принципів програмування, таких як, наприклад, SOLID, програмний продукт легше доповнювати новим функціоналом або оновлювати що уже існує.

Архітектура, що дотримується модульності та незалежності компонентів системи, полегшує автоматичне тестування та забезпечує більшу якість програмного забезпечення.

3.2.1. DDD та структура проєкту

Domain-Driven Design це проєктування з орієнтацією на домен, що ставить у центр уваги бізнес-домен та намагається відобразити його як програмну модель.

Програмне забезпечення притримується принципів DDD цей підхід має на меті полегшити розробку складних програм об’єднавши пов’язані частини програмного забезпечення в модель [13]. Зв’язок моделі та реалізації допомагає під час обслуговування та постійній розробці тому, що код можна інтерпретувати на основі розуміння моделі [14].

Основна мета DDD це створити програмне забезпечення, яке максимально буде відповідає бізнес-потребам. Цей підхід виділяє декілька основних та ключових понятій:

- **Агрегати** - Групи пов'язаних об'єктів, які взаємодіють між собою як єдине ціле.

- **Сутності** - Це об'єкт, який має унікальну ідентичність, а не просто значення. Сутності зазвичай зберігаються в базі даних.
- **Значущі події (Domain Events)** - це події, які відбуваються всередині домену і мають значення для бізнес-процесів.
- **Сервіси** - виконують бізнес-операції, які не належать до конкретних об'єктів або агрегатів. Сервіси надають високорівневу функціональність та взаємодіють з різними об'єктами, тут відбувається основна логіка.
- **Репозиторії** - Відповідають за доступ до постійного сховища даних для збереження та отримання об'єктів. Репозиторії приховують деталі роботи з базою даних і забезпечують інтерфейс для роботи з доменними об'єктами.

Отже, DDD пропонує підхід до розробки програмного забезпечення, який зосереджується на моделюванні бізнес-домену.

Структура проєкта побудована з урахуванням шарів. Тобто перший рівень відповідає за взаємодію зі зовнішніми запитами. Другий шар зосереджений на вирішені бізнес задач та займається логічною обробкою. Та третій рівень відповідає за взаємодію з базами даних та операціями пов’язаними з цим.

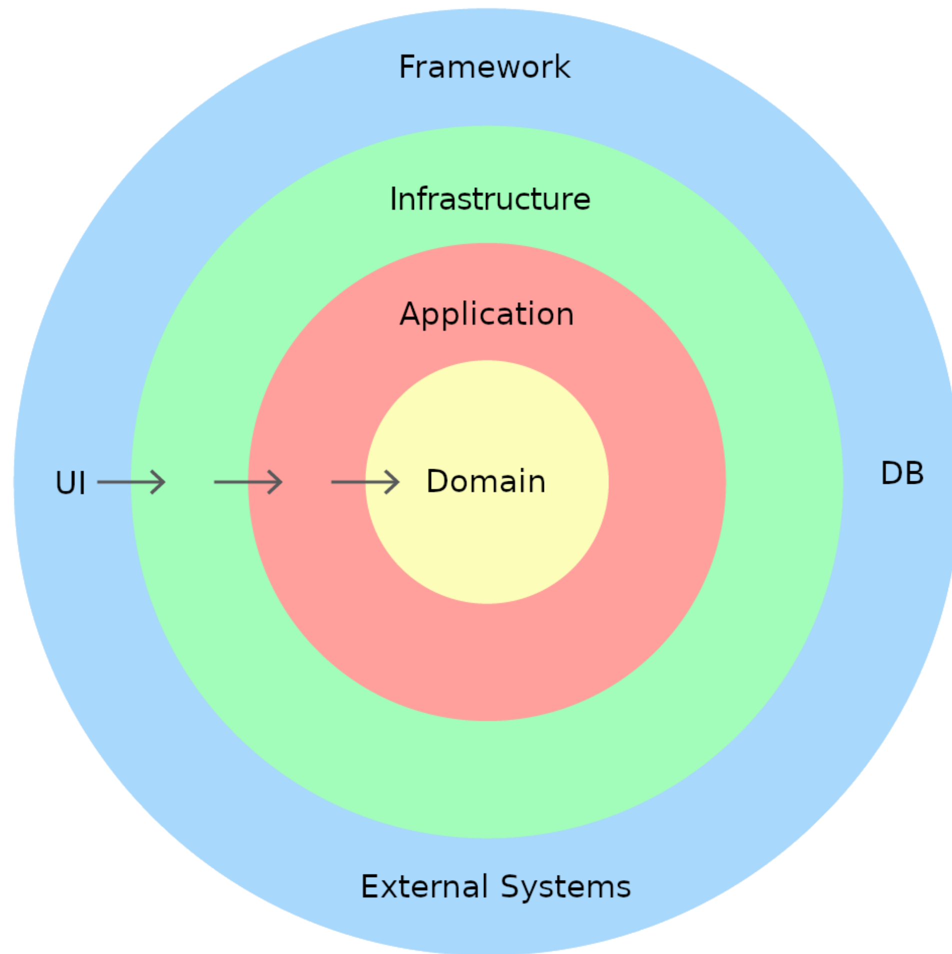


Рисунок 3.5 Domain Driven Design layers [13]

Таблиця 3.1 - Структура проєкту Kyoto Bot Factory

Назва		Призначення
Kyoto.Bot.Client	Мікросервіс	Обробляє запити створених Telegram ботів.
Kyoto.Bot.Factory	Мікросервіс	Обробляє запити безпосередньо до основного боту, що управляє іншими

Продовження таблиці 3.1 – Структура проєкту Kyoto Bot Factory

Kyoto.Telegram.Receiver	Мікросервіс	Отримує повідомлення від Telegram та розподіляє їх
Kyoto.Telegram.Sender	Мікросервіс	Відповідає за відправку повідомлень до Telegram
Kyoto.Database	Бібліотека	Собі репозиторії та моделі бази даних. Відповідає за роботу з базою даних
Kyoto.DI	Бібліотека	Методи розширення для зручного підключення залежностей до проєкту
Kyoto.Domain	Бібліотека	Зберігаються домені моделі та інтерфейси
Kyoto.Dto	Бібліотека	Зберігаються об’єкт передачі даних
Kyoto.Extensions	Бібліотека	Методи розширення, які допомагають у різних ситуаціях
Kyoto.Kafka	Бібліотека	Логіка роботи з Кафкою
Kyoto.Kafka.Events	Бібліотека	Зберігаються об’єкти подій (повідомлення) Кафки
Kyoto.Kafka.Handlers	Бібліотека	Обробники подій (повідомлень) Кафки

Продовження таблиці 3.1 – Структура проєкту Kyoto Bot Factory

Kyoto.Logger	Бібліотека	Логіка обробки логування
Kyoto.Services	Бібліотека	Сервіси з основною бізнес логікою
Kyoto.Settings	Бібліотека	Зберігаються об’єкти налаштувань системи
Kyoto.Command.Tests	Бібліотека	Юніт тести команд
Kyoto.Service.Tests	Бібліотека	Юніт тести сервісів

3.2.2. Міжсервісна взаємодія

Чат-бот фабрика базується на мікросервісній архітектурі тому, що це найпопулярніший та самий перевірений багатьма проектами архітектурний стиль і шаблон для розробки та розгортання складних продуктів.

Мікросервісна архітектура, сама по собі несе можливість прискорити розробку програмного забезпечення через те що одна сутність сервісу позиціонує себе як незалежну одиницю модуля проєкту і концентрує логіку розробки, саме для певної задачі [15].

Було створено 4 мікросервіси. Все починається з Kyoto.Telegram.Receiver. У цьому мікросервісі є лише один метод, який отримує повідомлення з Telegram. Його основна задача полягає в тому, щоб розподілити отримані повідомлення по топіках Kafka з урахуванням відповідного тенанта. Для визначення, в який тенанта потрібно відправити повідомлення, сервіс отримує це значення з заголовка запиту, надісланого з Telegram. Далі повідомлення відправляється до одного з обробників, відповідного повідомлення:

- **Command** - відповідає за обробку глобальних команд бота, такі як /start та /cancel
- **Message** - це обробник, який обробляє звичайні текстові повідомлення від користувачів. Він проводить аналіз отриманого повідомлення, щоб визначити, чи це просто текстове повідомлення, дія на перемалювання меню чи активація команди.
- **CallbackQuery** - як правило використовується у ході обробки команди. Тому одразу відправляється у сервіси, що відповідають за цей процес.

Ця логіка може відбуватися як на стороні фабрики Kyoto.Bot.Factory, так і на стороні клієнтського сервісу Kyoto.Bot.Client, все залежить звідки та під яким тенантом було отримане повідомлення від Telegram. Після того як сервіси виконали певні логічні операції, вони можуть відправити наступне повідомлення кафки до Kyoto.Telegram.Sender, який відповідає за відправку повідомлень до Telegram. Для відправки повідомлень використовується TBot бібліотека, вона також за допомогою Redis слідкує за лімітами запитів.

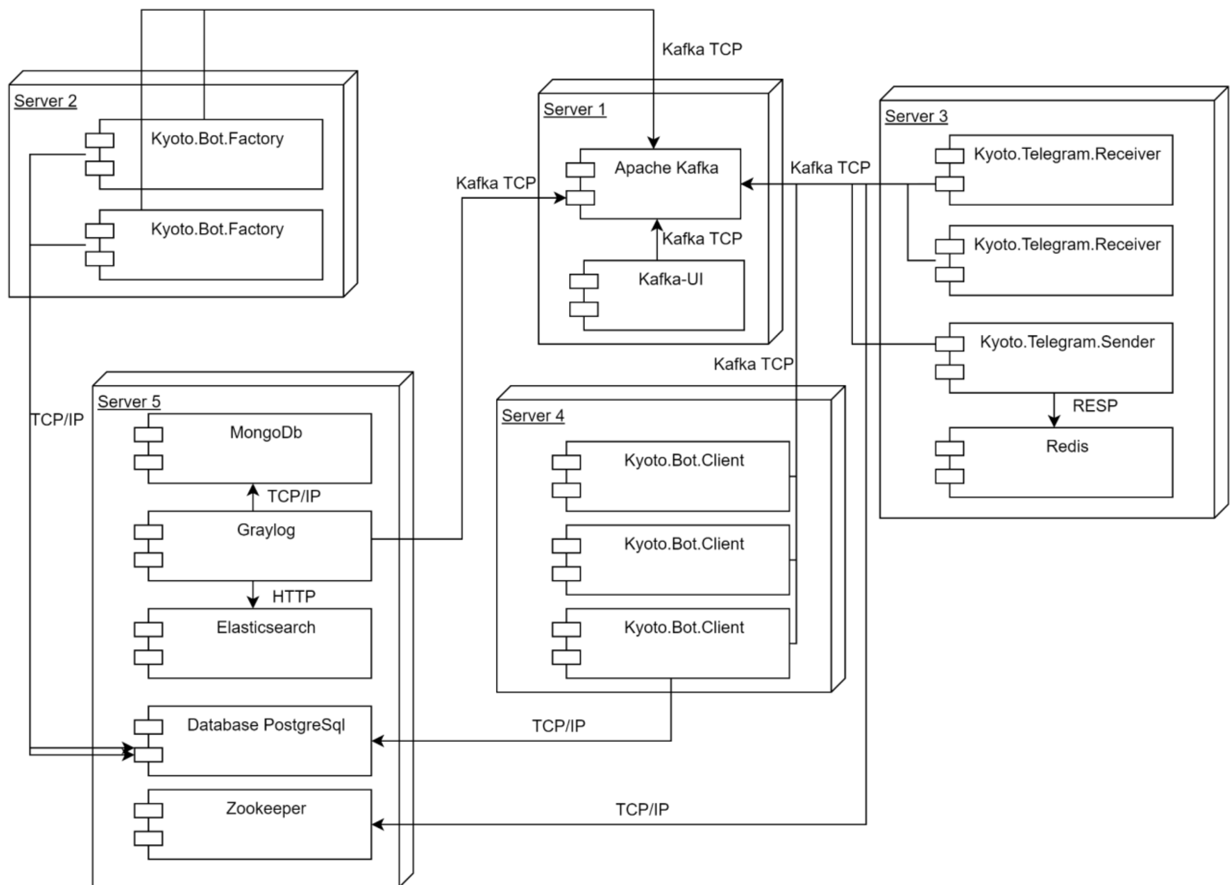


Рисунок 3.6 - Діаграма розгортання

3.2.3. PostgreSQL та схема бази даних

Програмне забезпечення використовує PostgreSQL тому, що цу дуже потужна система керування реляційними базами даних, яка працює на принципах об'єктно-реляційного підходу. Її особливості:

- **Об'єктно-реляційна модель**. PostgreSQL підтримує стандартні реляційні типи даних (числа, строки тощо), а також складні структури даних, такі як масиви та hstore (для ключ-значення), json (для даних JSON) та інші.
- **Сучані можливості SQL**. Він підтримує багато функцій SQL, включаючи підзапити, внутрішні та зовнішні об'єднання, транзакції, зберіганні процедури, тригери та багато іншого.

- **Сумісність з ACID**. PostgreSQL дотримується принципів ACID (атомарність, сумісність, ізоляція, стійкість), що гарантує надійність та правильність.
- **Масштабованість та продуктивність**. PostgreSQL відомий своєю високою продуктивністю та можливістю масштабування. Його можна використовувати для маленьких проектів та великих баз даних, що містять терабайти даних.
- **Безпека**. PostgreSQL має розширені можливості забезпечення безпеки, включаючи різні рівні авторизації, шифрування даних на рівні диска та підтримку SSL [22].

Програмне забезпечення має архітектуру мультитенантності, що дозволяє створювати окремі бази даних для кожного нового боту під час їх розгортання. Ці бази даних повністю ізольовані від інших тенантів, що забезпечує високий рівень безпеки та конфіденційності. Під час першого розгортання кожна база даних заповнюється базовою інформацією, необхідною для роботи боту.

Перш за все потрібно розглянути такі таблиці як User та ExternalUser (див. рис. 3.5). Ці дві сутності пов’язані між собою зв’язком один до одного. Тут сформована деяка абстракція для подальшого розвитку проєкту. Виділяється сутність загальна користувача та додаткова, як зовнішній користувач. У даній реалізації це Telegram клієнт. У цих таблицях зберігається інформація про користувачів, така як Ім’я, Призвіще, Телефон, username та ID зовнішнього користувача (Telegram User Id).

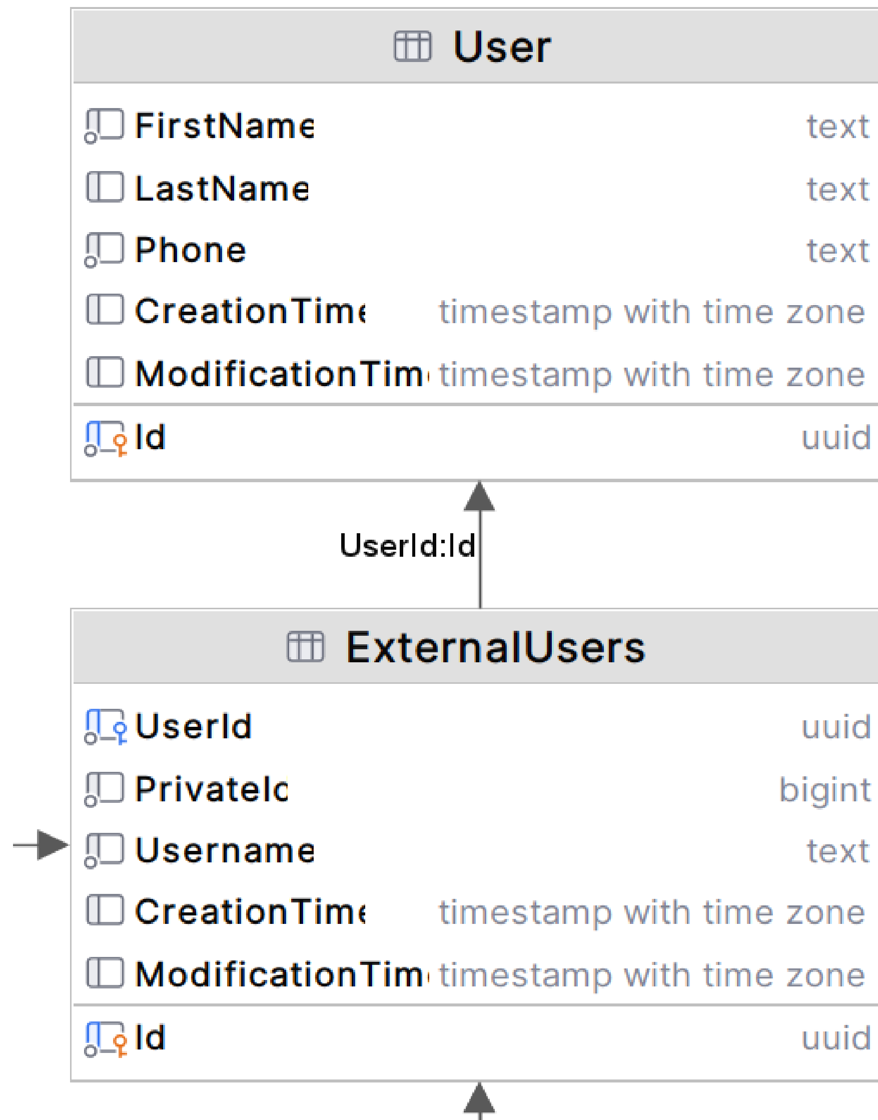


Рисунок 3.7 - Таблиці бази даних. User та ExternalUser

Наступна важлива таблиця, це Bot (див. рис. 3.6) у ній зберігаються дані про зареєстрованих Telegram ботів та стани у яких на даний момент вони перебувають. Саме головне це те що таблиця у собі містить токен чат-бота та username як назва тенанту. Ця таблиця пов’язана з ExternalUser зв’язком багато до одного. Також ця сутність доступна лише для контексту бази даних фабрики.

Bots	
ExternalUserId	uuid
PrivateId	text
FirstName	text
Username	text
CanJoinGroups	boolean
CanReadAllGroupMessage:	boolean
SupportsInlineQueries	boolean
Token	text
IsEnable	boolean
IsDeployed	boolean
CreationTime	timestamp with time zone
ModificationTime	timestamp with time zone
Id	uuid

Рисунок 3.8 - Таблиця бази даних. Bot

У бота також існують такі таблиці у парі, як EventMessage і PreparedMessage та TemplateMessageType і TemplateMessage (див. рис. 3.7). Ці по парні таблиці пов’язані між собою зв'язком один до багатьох.

Сутності EventMessage і PreparedMessage відповідають за збереження подій відправки повідомлення та саме повідомлення відповідно. Вони використовуються для функціоналу пов’язаним з розсилкою повідомлень.

Сутності TemplateMessageType і TemplateMessage зберігають тип шаблонного повідомлення та саме повідомлення відповідно.

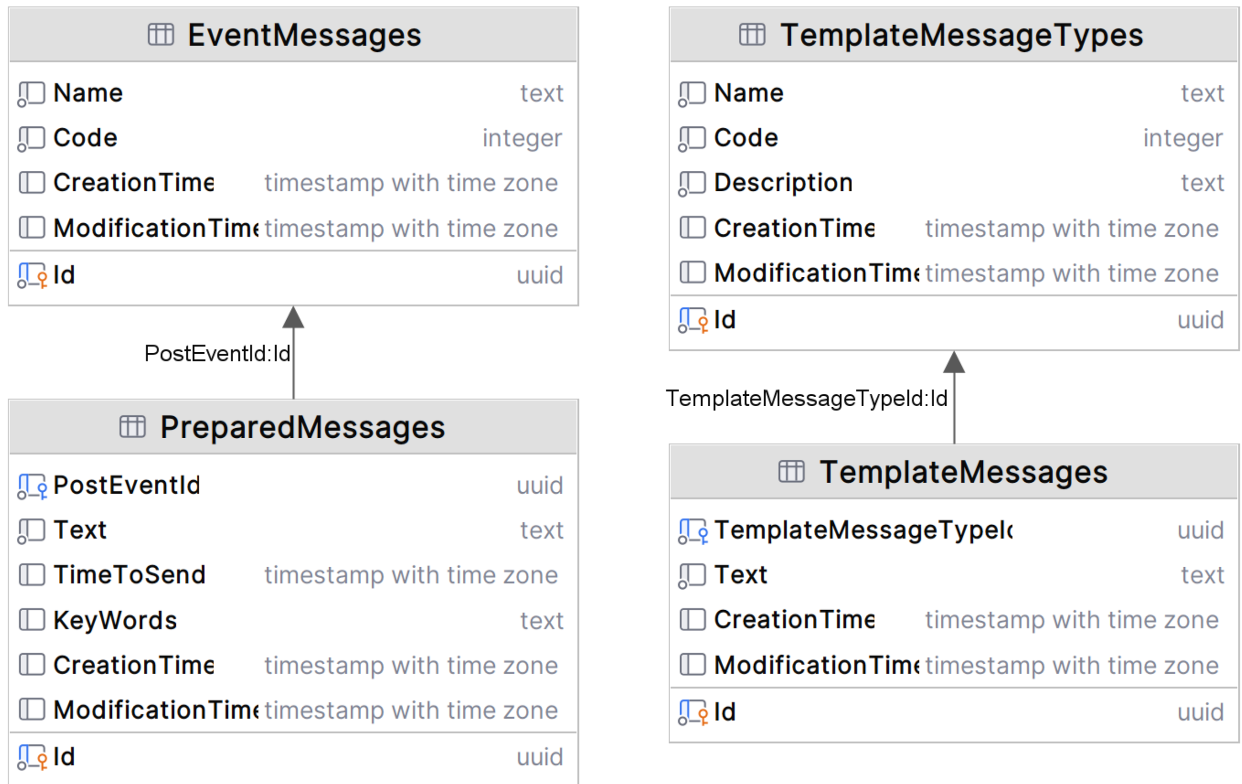


Рисунок 3.9 - Таблиці бази даних. EventMessage і PreparedMessage та TemplateMessageType і TemplateMessage

Таблиця Command (див. рис. 3.8) зберігає у собі стан виконуючої команди. Вона не явно прив’язана до ExternalUser. Ця таблиця потрібна для збереження кроку та стану команди. А також ця сутність зберігає у собі проміжні дані, які були згенеровані у ході роботи команди.

Commands	
SessionId	uuid
ChatId	bigint
Name	text
ExternalUserId	bigint
Step	integer
State	integer
AdditionalDat	text
CreationTime	timestamp with time zone
ModificationTim	timestamp with time zone
Id	uuid

Рисунок 3.10 - Таблиця бази даних. Command

Таблиця MenuPanel (див. рис. 3.8) зберігає у собі панелі меню, що присутні у чат-бота. До цієї таблиці зв’язком одна до багатьох прив’язується таблиця MenuButton (див. рис. 3.8), вона у свою чергою містить назви кнопок, що будуть міститися у панелі.

Також ця сутність має спеціальний системний код для того, щоб програмне забезпечення розуміло, яку саме команду ця кнопка активує.

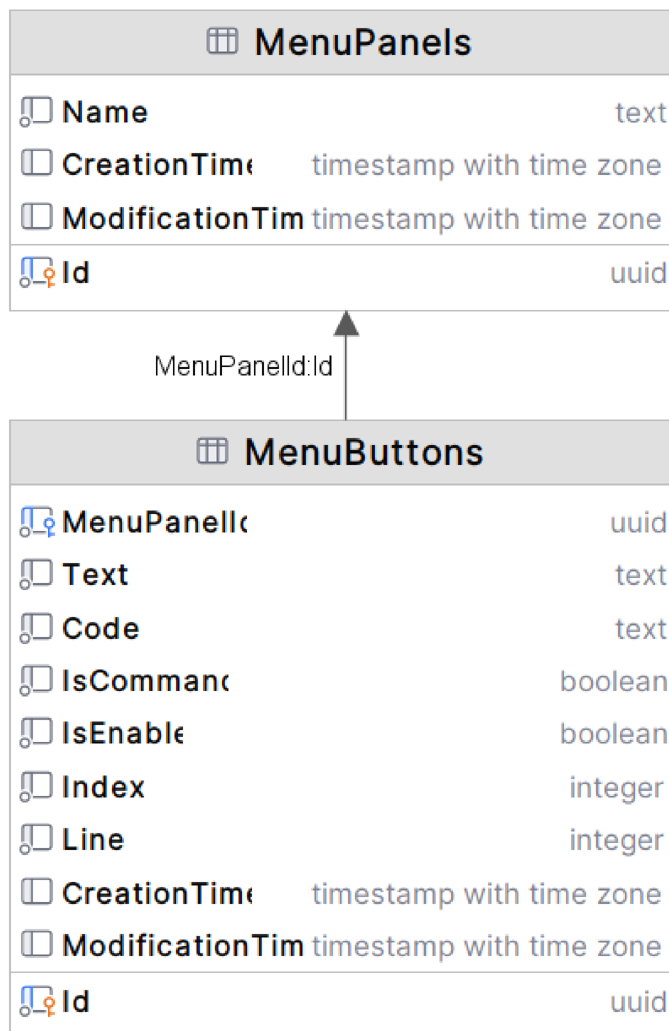


Рисунок 3.11 Таблиця бази даних. MenuPanel та MenuButton

PreOrderService та Service (див. рис. 3.9) таблиці відповідають за збереження інформації про замовлення та про те хто його оформив. Service - це сама послуга, вона зв'язана з PreOrderService як один до багатьох. У свою чергу PreOrderService додатково пов'язана теж як один до багатьох з таблицею ExternalUser. Таким чином ми можемо побачити у базі даних, хто зробив передзамовлення та якої послуги саме.

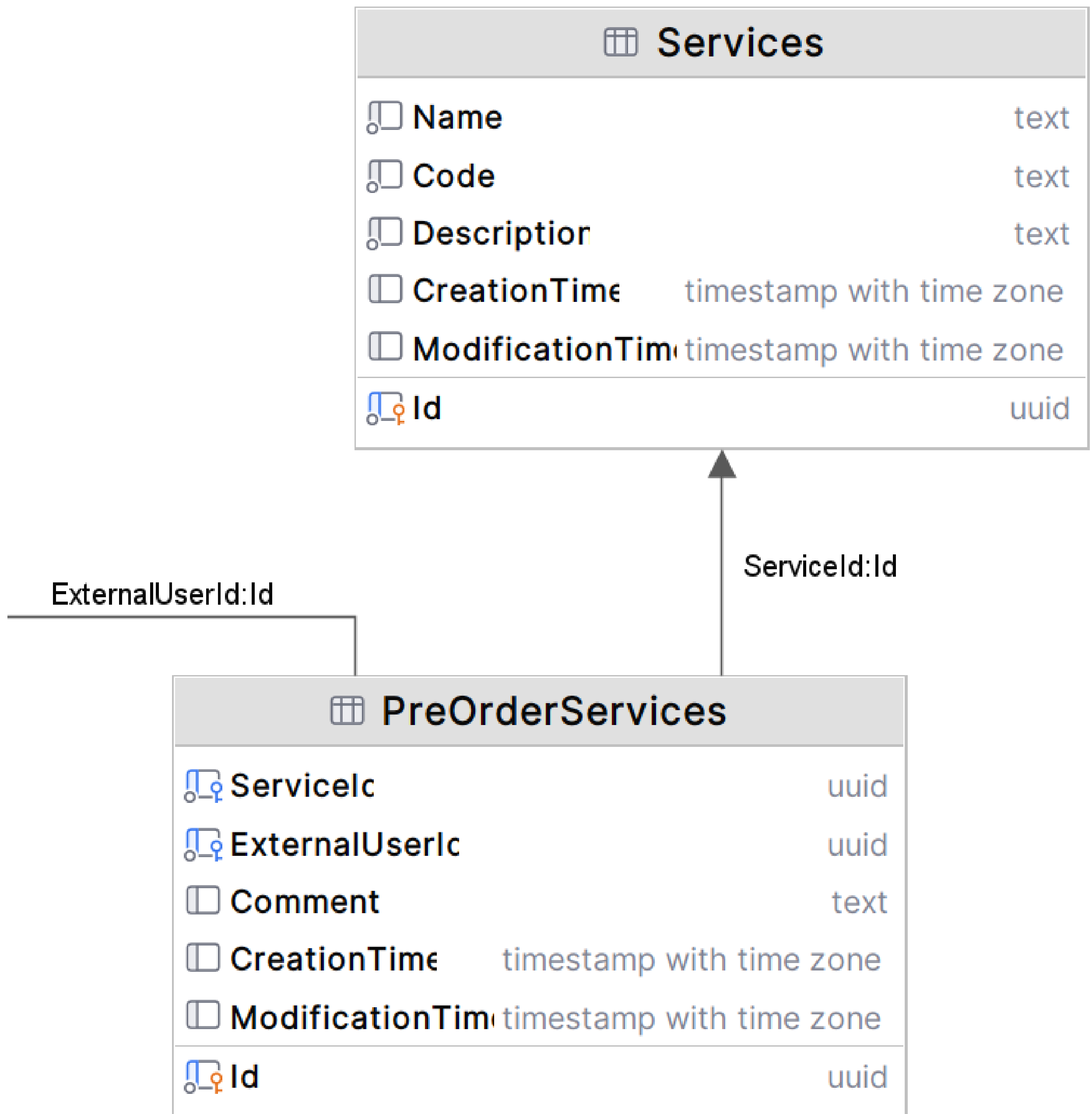


Рисунок 3.12 Таблиці бази даних. PreOrderService та Service

Зворотні відгуки клієнтів зберігаються у таблицю Feedback (див. рис. 3.10). Вона пов’язана з таблицею ExternalUser як один до багатьох, але цей зв’язок не є обов’язковим, адже зворотній відгук можна залишити анонімно. Ця таблиця містить у собі, відгук, його тип та кількість зірочок.

Feedbacks	
ExternalUserIc	uuid
Text	text
Type	text
StarCount	integer
CreationTime	timestamp with time zone
ModificationTim	timestamp with time zone
Id	uuid

Рисунок 3.13 Таблиця бази даних. Feedback

Також, для правильної роботи системи, у базі даних присутні системні таблиці такі як, __EFMigrationHistory (див. рис. 3.11), вона була створена автоматично за допомогою фреймворка EntityFramework для зберігання історії міграцій. І таблиця SystemStatus (див. рис. 3.11), яка була створена для зберігання різного роду станів системи.

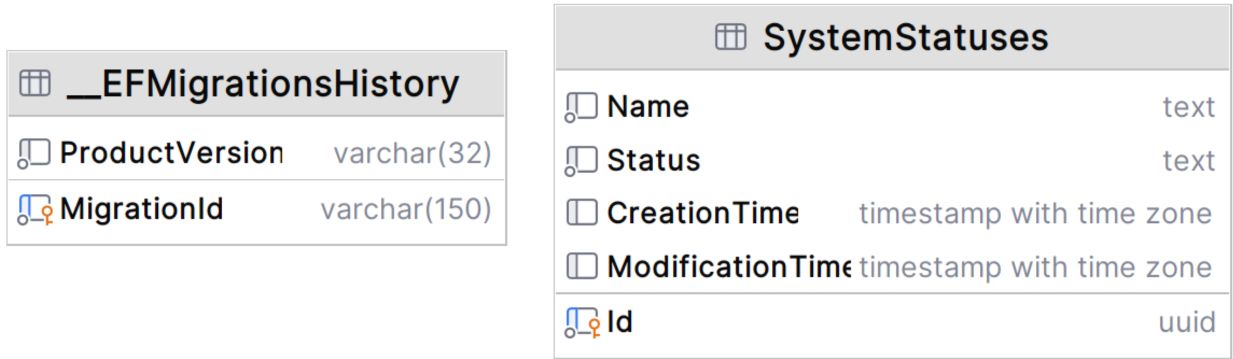


Рисунок 3.14 - Таблиці бази даних. __EFMigrationHistory та SystemStatus

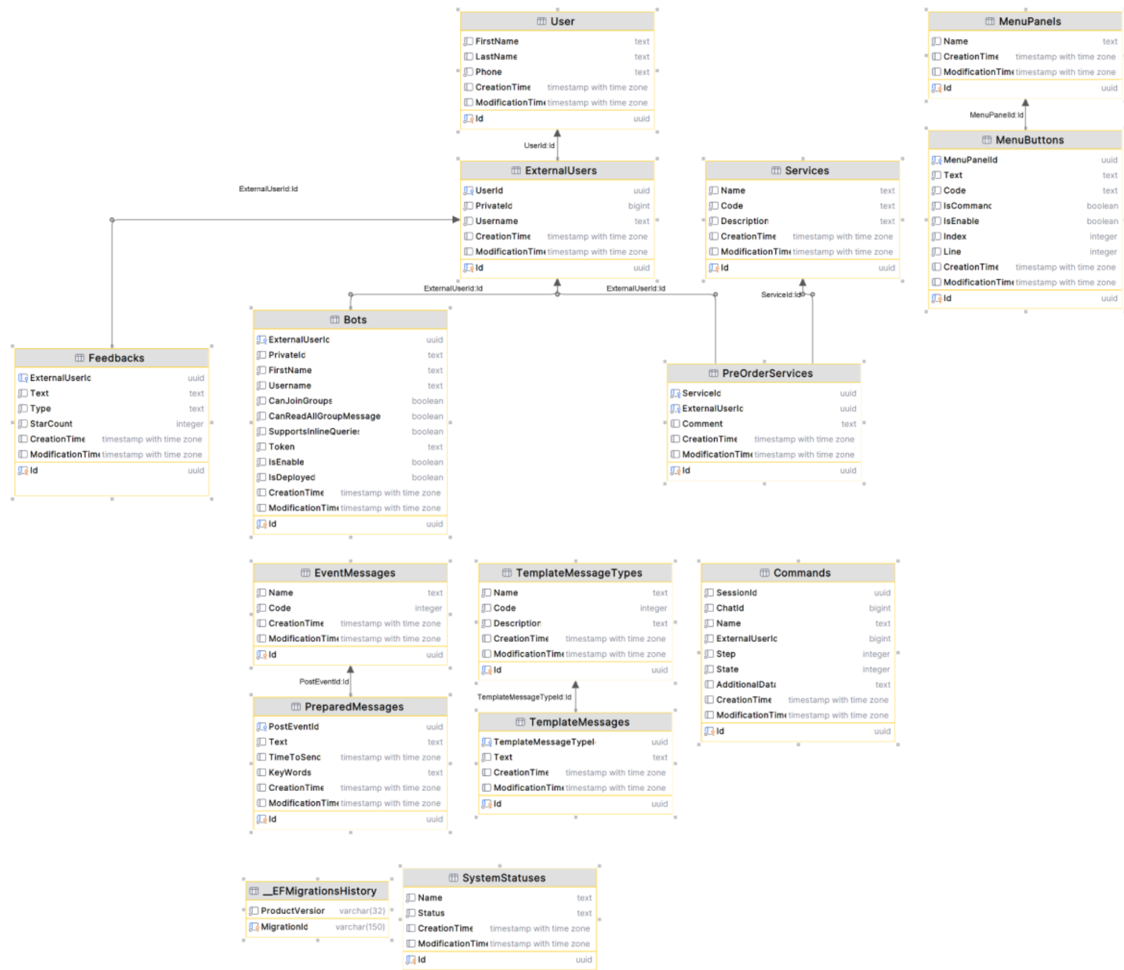


Рисунок 3.15 Таблиці бази даних. Повне відображення бази даних проєкту

3.2.4. Multi-tenant architecture

Multi-tenant architecture - це архітектурний підхід до розробки програмного забезпечення, що дає можливість запускати багато клієнтів на одному екземплярі програмного забезпечення, забезпечуючи окремий контекст для кожного з них [17].

З ключових особливостей Multi-tenant architecture можно виділити наступне:

- Зниження витрат на апаратне забезпечення, енергію та місце на диску.
- Ефективне використання ресурсів тому, що одна інфраструктура використовується для обслуговування всіх користувачів.
- Можливість швидкої масштабованості та підтримка різних клієнтів.
- Полегшений процес оновлення програмного забезпечення для всіх користувачів.

Kyoto Bot Factory - підтримує мультитенанту архітектуру для підтримки декількох Telegram ботів. Для кожного з ботів при розгортанні створюється власна база даних тенант.database (див. рис. 3.13). Це надає можливість ізолювати дані кожного клієнта при цьому використовуючи один і той же додаток.

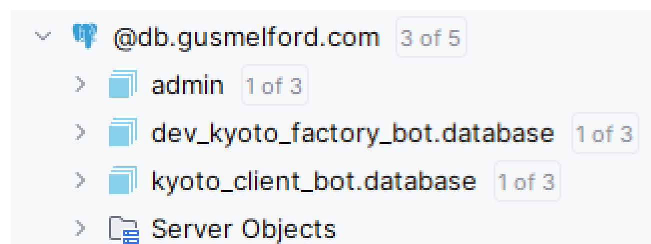


Рисунок 3.16 - Бази даних для різних тенантів

Також, для коректної роботи системи, розподілення по тенантам відбувається і у Кафці (див. рис. 3.13). Для того щоб повідомлення доходили до конкретних обробників та не пересікалися з іншими ботами.

dev_kyoto_factory_bot.CallbackQueryEvent	10	0
dev_kyoto_factory_bot.CommandEvent	10	0
dev_kyoto_factory_bot.FactoryRequestEvent	10	0
dev_kyoto_factory_bot.MessageEvent	10	0
dev_kyoto_factory_bot.RequestEvent	10	0
kafka_bot_factory.logs	1	0
kyoto_client_bot.CallbackQueryEvent	10	0
kyoto_client_bot.CommandEvent	10	0
kyoto_client_bot.MessageEvent	10	0
kyoto_client_bot.RequestEvent	10	0
kyoto_client_bot.TemplateMessageEvent	10	0

Рисунок 3.17 - Розділення топіків Кафки по різним тенантам

3.2.5. Система обробки команд - патерн програмування State

Головна задача бота правильно та якісно обробляти команди користувача. Враховуючи те що у чат бота може бути багато різних команд, потрібно було створити абстрактний механізм, який би дозволив зручно та швидко додавати нові команди під єдиний інтерфейс. Була розроблена система, що в основу своєї реалізації взяла патерн програмування State.

Команда у розумінні архітектури бота, це певна взаємодія з клієнтом в умовах одного контексту, наприклад реєстрація нового бота. Цей процес відбувається у декілька етапів. Отримання токена та підтвердження реєстрації від клієнта. Цей етап являє собою певний кроки, що у свою чергу складаються з двох станів (див. рис. 3.14):

1. *Запит від бота на певну дію з боку користувача.*
2. *Відповідь або дія клієнта*

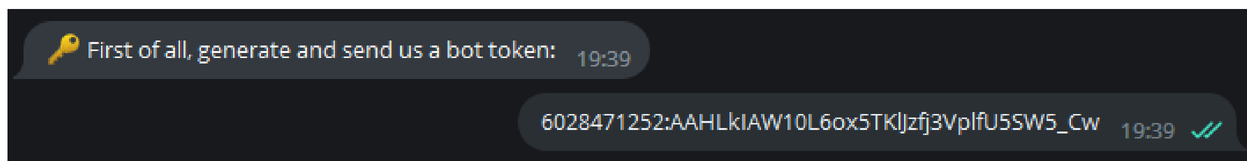


Рисунок 3.18 Запит бота та відповідь клієнта

Таким чином одна команда складається з блоків таких кроків. Усі етапи об’єднуються під однією фабрикою, що у конкретний момент часу генерує необхідний крок та активує його у коді.

Для зберігання проміжних даних про виконання команди та її прогрес в базі даних використовується таблиця Command. Ці дані передаються за допомогою класу, що відображає стан, що неявно застосовує патерн програмування "Стан". У обробник команд передається контекст виконання команди, який впливає на наступну роботу системи.

Виклики відбуваються за допомогою панелі меню внизу месенджера. А також є 2 глобальні команди /start (для початку взаємодії з ботом) та /cancel (для відміни будь якої команди).

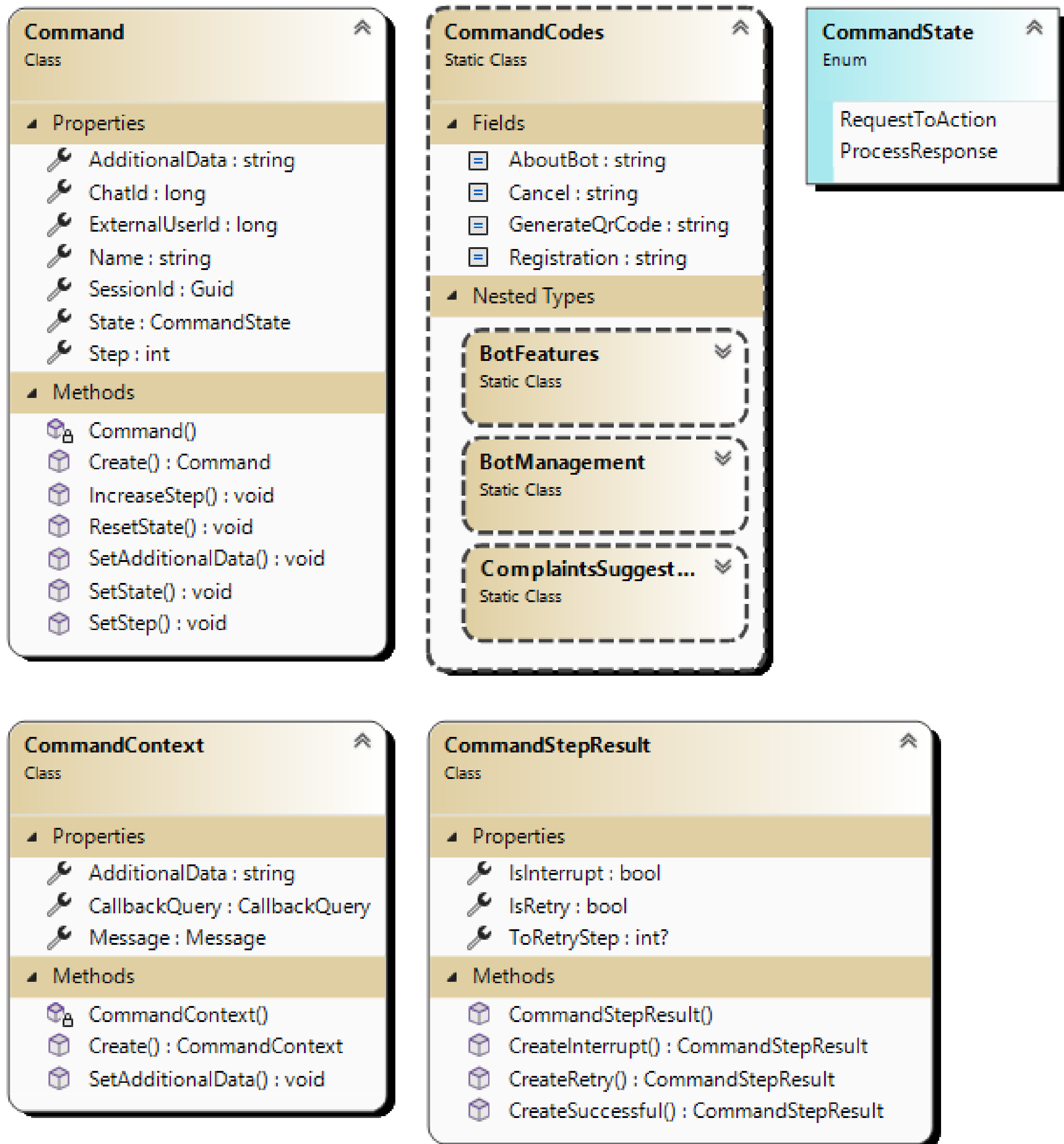


Рисунок 3.19 Діаграма класів - Kyoto.Domain

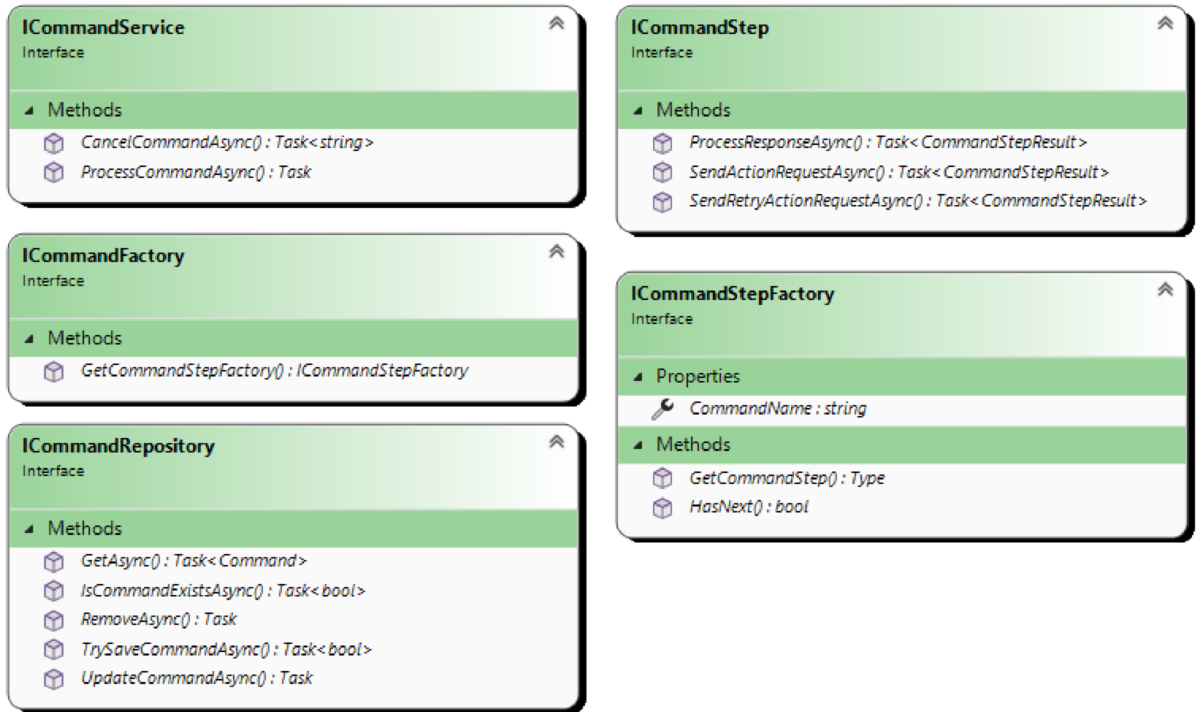


Рисунок 3.20 Діаграма класів - Kyoto.Domain

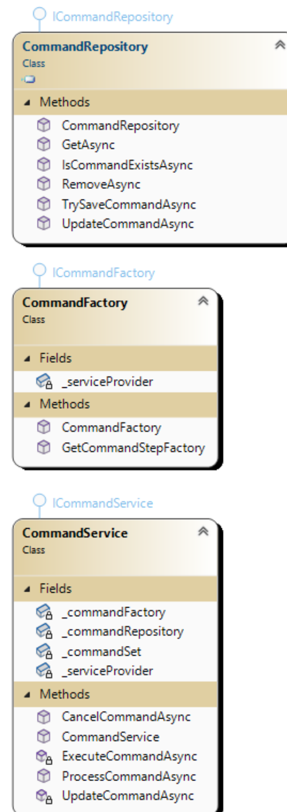


Рисунок 3.21 Діаграма класів - Kyoto.Services

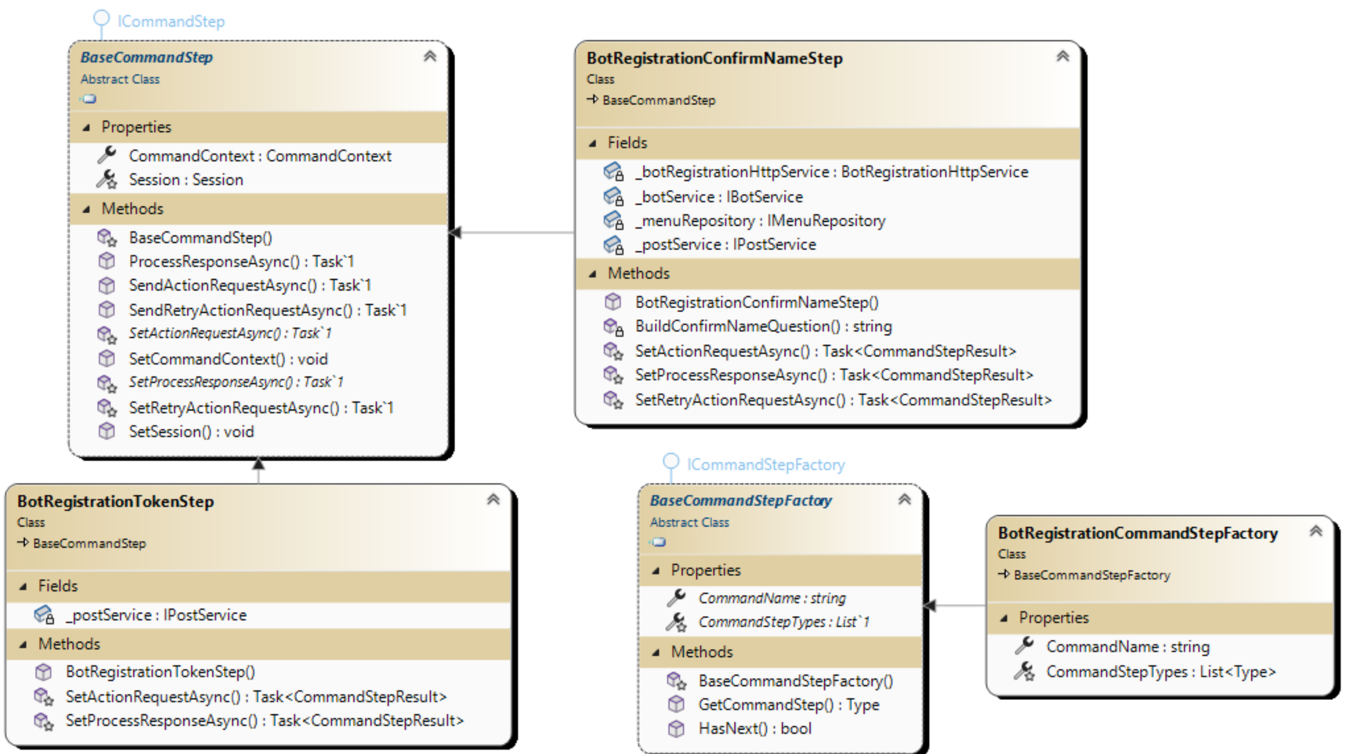


Рисунок 3.22 Діаграма класів - Kyoto.Commands

3.2.6. API клієнтського сервісу

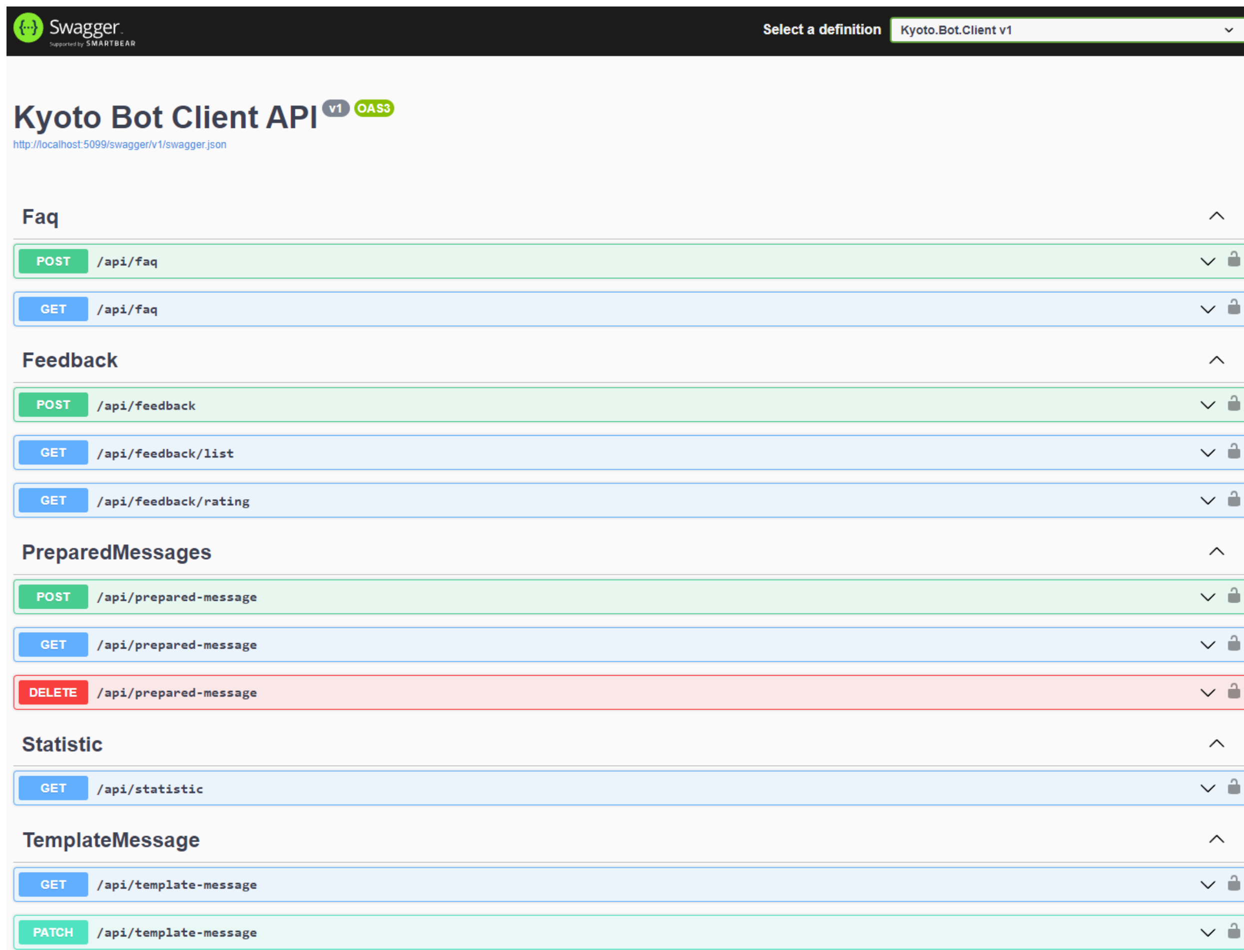


Рисунок 3.23 Swagger Kyoto Bot Client

Для Kyoto Bot Client сервісу було розроблене API для налаштування клієнтського бота. Через ці запити фабрика взаємодіє з мікросервісом і також у клієнтам є можливість без UI змінювати налаштування для свого сервісу, використовуючи ключ, який надає фабрика.

Присутні такі методи:

- **POST /api/faq** - встановити FAQ.
- **GET /api/faq** - отримати FAQ
- **POST /api/feedback** - записати новий зворотній відгук.

- **GET /api/feedback/list** - отримати список відгуків (використовується пейджинація)
- **GET /api/feedback/rating** - отримати рейтингову статистику
- **POST /api/prepared-message** - записати підготовлене повідомлення
- **GET /api/prepared-message** - отримати заготовлене повідомлення
- **DELETE /api/prepared-message** - видалити заготовлене повіломлення
- **GET /api/statistic** - отримати статистику використання бота
- **GET /api/template-message** - отримати шаблоне повідомлення
- **PATCH /api/template-message** - оновити шаблоне повідомлення

3.3. Опис програми Kyoto Bot Factory

Для того щоб почати роботу з Telegram ботом фабрикою потрібно вести команду **/start**. Як правило Telegram сам пропонує її для старту (див. рис. 3.16). Далі бот попросить поділитися своїми контактними даними для того щоб продовжити співпрацю. Ці дані необхідні для успішної реєстрації. Щоб поділитися контактом, користувачу потрібно натиснути кнопку внизу, яка автоматично з'явиться перед клієнтом.

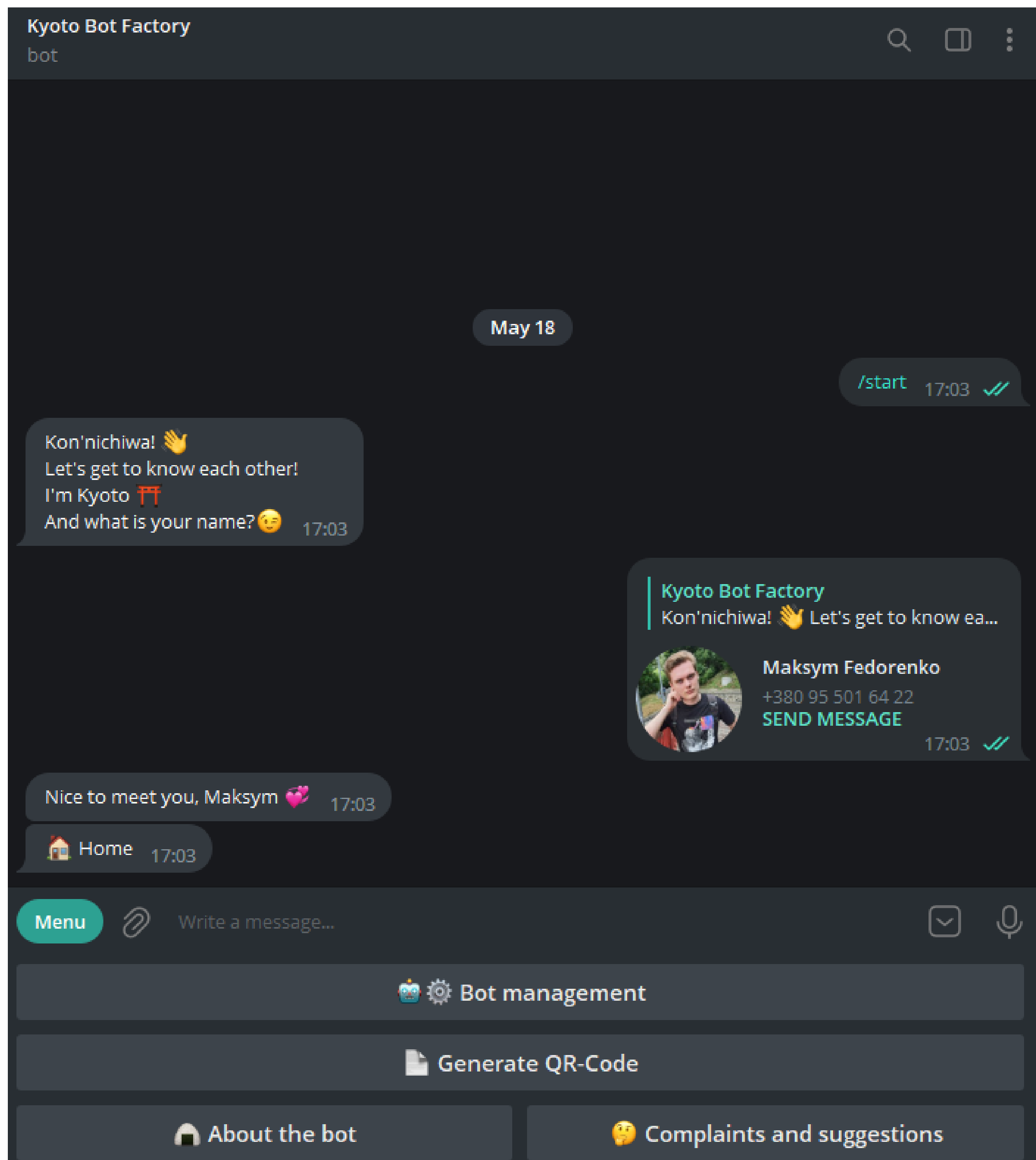


Рисунок 3.24 Початок роботи з Kyoto Bot Factory

Після реєстрації у користувача з'явиться домашнє меню. В меню спочатку буде 4 кнопки:

- 🤖 ⚙️ **Bot management** - управління Telegram ботами

- 🪄 **Features of the bot** - Функції, які можна налаштувати для бота. (Спершу вони не доступні, до того моменту поки користувач не зареєстрував першого бота)
- 🧐 **Complaints and suggestions** - тут можна залишити відгук про роботу фабрики
- 📄 **Generate QR-Code** - команда для генерації QR-Code

Для подальшої роботи з фабрикою користувачеві потрібно зайти до розділу 🤖⚙️ Bot management та створити зареєструвати свого телеграм бота. Перед ним відкриється меню з 5 кнопками управляня телеграм ботом (див. рис. 3.17):

- 🏗 **Register a new bot** - команда для реєстрації нового бота
- 🚀 **Deploy bot** - команда для розгортання бота (створення інфраструктури та запуск прослуховування оновлень з телеграму)
- 😴 **Disable bot** - вимкнути прослуховування оновлень з телеграму
- ❌ **Delete bot** - видалити усю інфраструктуру та самого бота
- ⬅️ **Back** - 🏠 **Home** - повернутися на домашню сторінку

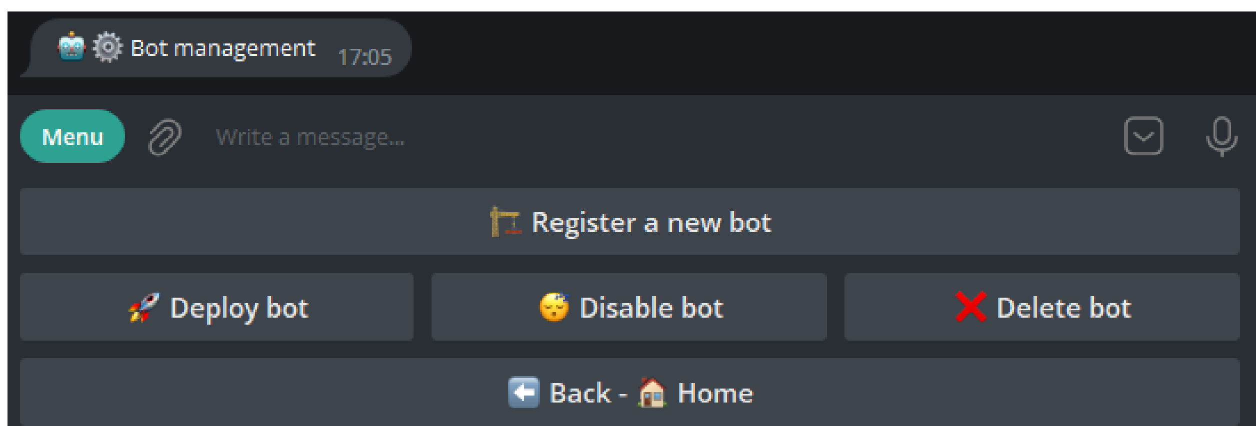


Рисунок 3.25 - Меню управління телеграм ботами у Kyoto Bot Factory

Реєстрація бота відбувається у два етапи, клієнт повинен надати токен телеграм бота попередньо отримавши у BotFather. Та надати його фабриці. Після чого чат-бот

запросить підтвердження реєстрації, якщо користувач погодився, його бот буде успішно зареєстрований у системі (див. рис. 3.17).

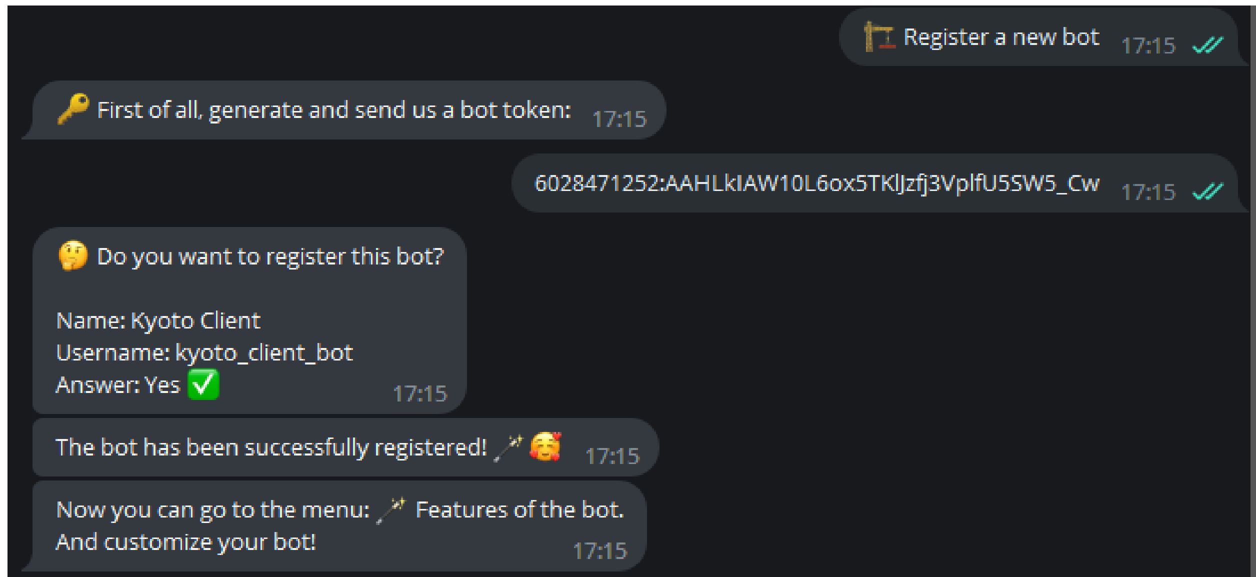


Рисунок 3.26 - Реєстрація нового телеграм бота

Після цього користувачу потрібно розгорнути свого бота за допомогою команди 🚀 Deploy bot (див. рис. 3.18). Цей процес займає приблизно одну хвилину. Як тільки процес завершився успішно, фабрика оповістить користувача про це, а також телеграм бот користувача напише йому в особисті повідомлення див. рис. 3.19).

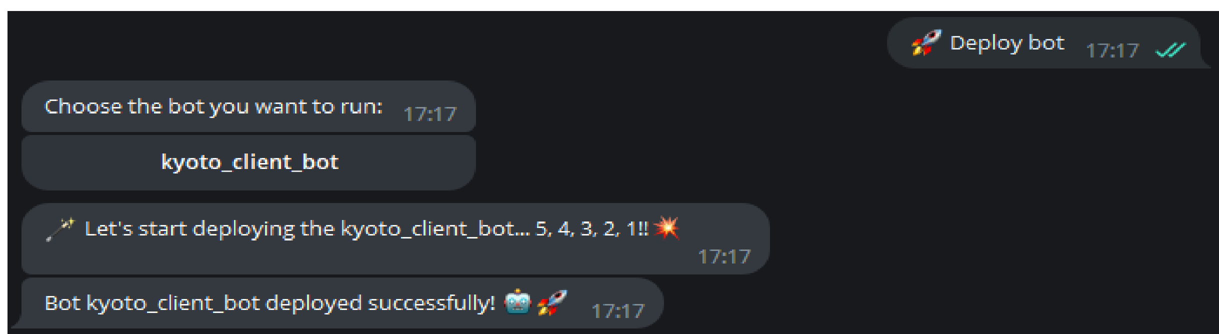


Рисунок 3.27 – Процес розгортання нового бота

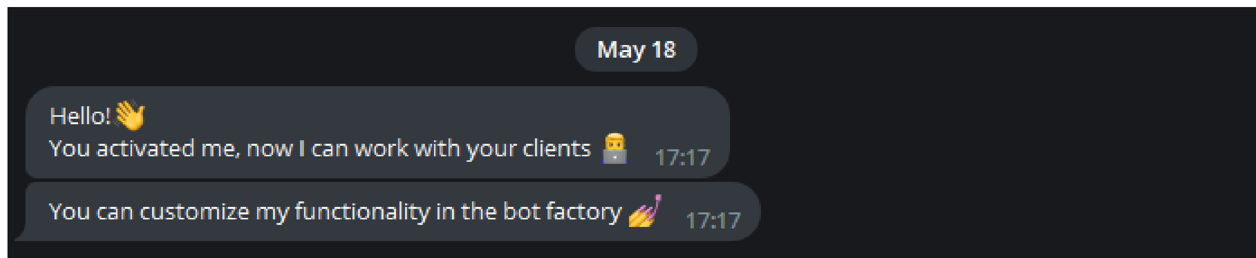


Рисунок 3.28 – Повідомлення телеграм після успішної реєстрації

Далі користувач може зайти у меню 🪄 Features of the bot та налаштувати той функціонал який йому потрібен.

- 🙋‍♂️ **Set FAQ** - встановити збірник відповідей на питання
- ✉️ **Add newsletter** - додати нове повідомлення для розсилки
- ❌ **Remove newsletter** - видалити повідомлення для розсилки
- 👨‍💻 **Change Registration Texts** - змінити текст реєстрації
- **i Get Statistics** - отримати статистику
- 🗣️ **Enable Feedback** - включи отримання зворотних відгуків
- 📃 **Show feedbacks** - показати список відгуків
- ⬅️ **Back** - 🏠 **Home** - повернутися на домашню сторінку

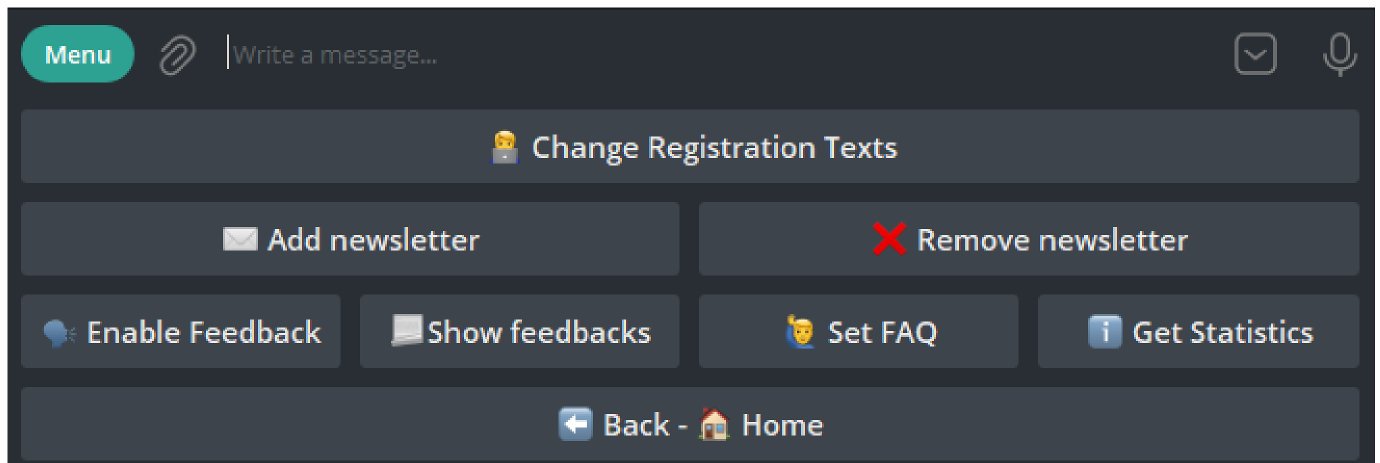


Рисунок 3.29 - Функції, які можна налаштувати для бота

Кожна з функціональностей виконує свої дії та надає користувачу зручно налаштовувати бота під свої бізнес потреби. Після налаштувань, клієнт користувача фабрикою буде мати змогу виконувати ті дії, що дозволив власник бота.

3.4. Тестування програмного забезпечення

Unit tests - це тип тестування програмного забезпечення, в якому окремі частини коду перевіряються на роботоздатність. Ці "одиниці" можуть бути окремими функціями, методами, процедурами або класами в коді.

Основна мета юніт-тестування - переконатися, що кожна окрема частина коду працює правильно. Це допомагає знайти і виправити баги на ранніх стадіях розробки, що може значно зменшити вартість виправлення помилок в майбутньому.

Юніт-тести допомагають підтримувати якість коду протягом часу. Якщо код змінюється і ці зміни ламають щось, юніт-тест допоможуть показати це. Такий підхід до тестування робить код більш надійним і стабільним.

Також варто зазначити, що юніт-тестування - це важлива частина практики розробки, відомої як test-driven development (TDD), коли юніт-тести пишуться перед самим кодом і слугують як вимоги до функціональності коду.

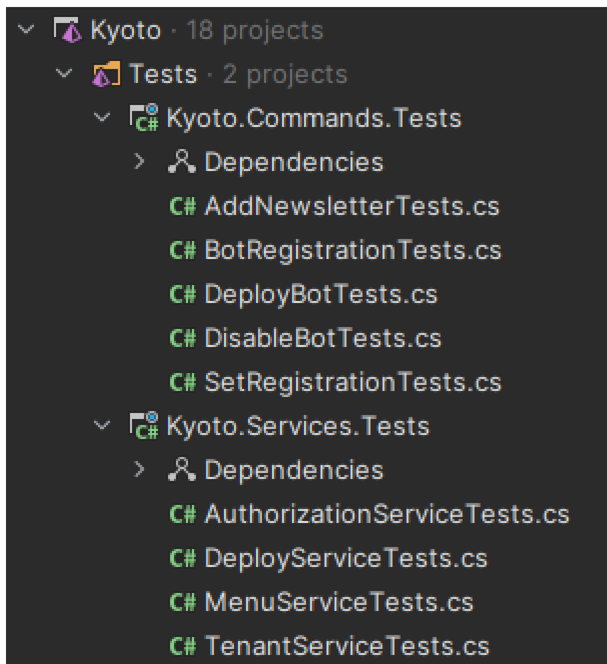


Рисунок 3.30 – Юніт-тести рішення Kyoto

```
[Fact]
public async Task TryGetMenuCommandCode_DoesNotExist_WhenMenuButtonTextEmpty()
{
    //Arrange
    string menuButtonText = string.Empty;
    _menuRepository.IsMenuButtonAsync(menuButtonText).Returns( returnThis: false);

    //Act
    var (isExist, command :string ) = await _menuService.TryGetMenuCommandCodeAsync(menuButtonText);

    //Assert
    Assert.False(isExist);
    Assert.Equal( expected: command,  actual: string.Empty);
}

[Fact]
public async Task TryGetMenuCommandCode_IsExists_WhenMenuButtonCommandText()
{
    //Arrange
    const string menuButtonText = "Command";
    const string commandCode = "CommandCode";
    _menuRepository.IsMenuButtonAsync(menuButtonText).Returns( returnThis: true);
    _menuRepository.GetMenuButtonCodeAsync(menuButtonText).Returns(commandCode);

    //Act
    var (isExist, command :string ) = await _menuService.TryGetMenuCommandCodeAsync(menuButtonText);

    //Assert
    Assert.True(isExist);
    Assert.Equal( expected: commandCode,  actual: command);
}
```

Рисунок 3.31 - Приклад юніт-тестів для методу TryGetMenuCommandCodeAsync з MenuService

```
[Fact]
public async Task SetActionRequest_WhenPostServiceSuccess()
{
    //Arrange
    const string text = "TestMessage";
    var successResult = CommandStepResult.CreateSuccessful();
    _postService.SendTextMessageAsync( session: Arg.Any<Session>(), text).Returns(Task.CompletedTask);
    var botRegistrationTokenStep = new BotRegistrationTokenStep(_postService);

    //Act
    var result = await botRegistrationTokenStep.SendActionRequestAsync();

    //Assert
    Assert.Equal( expected: successResult.IsRetry,  actual: result.IsRetry);
    Assert.Equal( expected: successResult.IsInterrupt,  actual: result.IsInterrupt);
    Assert.Equal( expected: successResult.ToRetryStep,  actual: result.ToRetryStep);
}

[Fact]
public async Task ProcessResponse_WhenPostServiceSuccess()
{
    //Arrange
    const string token = "TOKEN";
    var commandContext = CommandContext.Create(new Message { Text = token },  callbackQuery: null);
    var botRegistrationTokenStep = new BotRegistrationTokenStep(_postService);
    botRegistrationTokenStep.SetCommandContext(commandContext);

    //Act
    await botRegistrationTokenStep.ProcessResponseAsync();

    //Assert
    Assert.NotNull(botRegistrationTokenStep.CommandContext.AdditionalData);
    Assert.NotEmpty(botRegistrationTokenStep.CommandContext.AdditionalData);
}
```

Рисунок 3.32 - Приклад юніт-тестів для методів кроку реєстрації токена

Ручне тестування - це процес перевірки програмного забезпечення вручну з метою знайти помилки. Воно включає різні види тестування, такі як тестування функціональності, тестування інтерфейсу, регресійне тестування, сценарії використання та інші.

Ручне тестування потрібне для перевірки різних аспектів програмного забезпечення, які не можуть бути автоматизовані або для яких автоматизація не ефективна. Наприклад, перевірка зручності користування інтерфейсом (UI),

визначення відповідності вимогам бізнесу та забезпечення відповідності програмного забезпечення вимогам клієнтів часто здійснюється шляхом ручного тестування.

Проте, не можна стверджувати, що ручне тестування гірше автоматичного - вони просто мають різні застосування.

Автоматизоване тестування використовує спеціальні програми для виконання заданих тестових сценаріїв. Це особливо корисно при великому обсязі тестування, коли потрібно перевірити роботу системи під навантаженням, при регресійному тестуванні або коли тестування потребує високої точності.

Отже, ручне тестування та автоматичне тестування часто використовуються разом в рамках комплексної стратегії тестування. Ручне тестування використовується для вирішення питань, що вимагають гуманітарного розуміння, в той час як автоматичне тестування використовується для швидкого та ефективного виконання повторюваних тестових задач [23].

ВИСНОВКИ

У результаті виконання дипломної роботи розроблено Telegram чат-бот фабрику. Її актуальність полягає у тому, що наразі процес створення свого власного Telegram-бота доволі важка задача для людей, які не спеціалізуються у програмуванні.

1. Проведено аналіз предметної області та виявлено чотири веб-сервіси зі схожим набором функціоналу, а саме: SendPulse, ManyChat, Botpress, Flow XO. Визначено переваги і недоліки даних веб-сервісів.
2. Розроблено технічне завдання, визначено функціональні та нефункціональні вимоги щодо розробки програмного забезпечення.
3. Досліджено найсучасніші засоби та інструменти розробки програмного забезпечення з урахуванням особливостей предметної області.
4. Розроблено програмне забезпечення для спрощення створення особистого Telegram чат-бота з використанням мікросервісної архітектури та підвищення ефективності просування малого бізнесу.
5. Протестовано програмний додаток на стабільність роботи.
6. Робота пройшла апробацію: II Всеукраїнська наукова конференція студентів та молодих вчених "Наукові досягнення та відкриття сучасної молоді" за темою "Використання чат боту для взаємодії з клієнтом через соціальні мережі"; III Всеукраїнська науково-практична конференція "Сучасні інтелектуальні інформаційні технології в науці та освіті" за темою "Інтеграція штучного інтелекту з месенджером телеграм для покращення взаємодії користувачів з чат-ботами компанії"; Всеукраїнська науково-технічна конференція «Сучасний стан та перспективи розвитку IoT» за темою "Розробка чат-бота-фабрики для допомоги малому бізнесу в телеграм"
7. Програмне забезпечення можна застосовувати у просуванні свого власного підприємства за допомогою соціальної мережі Telegram використовуючи чат-бот.

СПИСОК ЛІТЕРАТУРИ

1. Global Digital Communication: Texting, Social Networking Popular Worldwide [Електронний ресурс] – Режим доступу до ресурсу: https://www.pewresearch.org/global/2011/12/20/global-digital-communication-texting-social-networking-popular-worldwide/.
2. Chatbot [Електронний ресурс] – Режим доступу до ресурсу: https://www.techtarget.com/searchcustomerexperience/definition/chatbot.
3. Telegram Bot Api [Електронний ресурс] – Режим доступу до ресурсу: https://core.telegram.org/bots/api.
4. SendPulse Chatbot for Telegram [Електронний ресурс] – Режим доступу до ресурсу: https://sendpulse.com/features/chatbot/telegram.
5. ManyChat [Електронний ресурс] – Режим доступу до ресурсу: https://manychat.com/signup.
6. ManyChat Review: An In-depth Look 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://botpenguin.com/manychat-review/.
7. Botpress [Електронний ресурс] – Режим доступу до ресурсу: https://botpress.com/
8. Flow XO [Електронний ресурс] – Режим доступу до ресурсу: https://flowxo.com/.
9. What is ASP.NET Core? [Електронний ресурс] – Режим доступу до ресурсу: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet-core.
10. Web Framework Benchmarks [Електронний ресурс] – Режим доступу до ресурсу: https://www.techempower.com/benchmarks/#section=data-r21&hw=ph&test=plaintext.
11. Apache Kafka [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/apache-kafka.
12. Kafka vs RabbitMQ: Biggest Differences and Which Should You Learn? [Електронний ресурс] – Режим доступу до ресурсу: https://www.simplilearn.com/kafka-vs-rabbitmq-article.

13. Domain Driven Design: Layers [Електронний ресурс] – Режим доступу до ресурсу: https://www.hibit.dev/posts/15/domain-driven-design-layers.
14. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software / Eric Evans.. – 560 с.
15. Surianarayanan C. Essentials of Microservices Architecture: Paradigms, Applications, and Techniques / C. Surianarayanan, G. Gopinath, R. Pethuru., 2019. – 314 с.
16. What is multi-tenant? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/multi-tenant.
17. Skeet J. C# in Depth / Jon Skeet., 2019. – 528 с.
18. What is Docker? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/docker.
19. What is Git? [Електронний ресурс] – Режим доступу до ресурсу: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git.
20. JetBrains Rider [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/rider/.
21. JetBrains DataGrip [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/datagrip/.
22. Postgresql [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/about/.
23. Ручне та автоматизоване тестування [Електронний ресурс] – Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/ruchne-ta-avtomatizovane-testuvannya/.
24. Newell A. The Psychology of Human Computer Interaction / A. Newell, P. Thomas, S. Card.. – (Lawrence Erlbaum Associates). – (ISBN 0-89859-859-1).
25. GOMS [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/GOMS.

ДОДАТОК А

Приклади коду

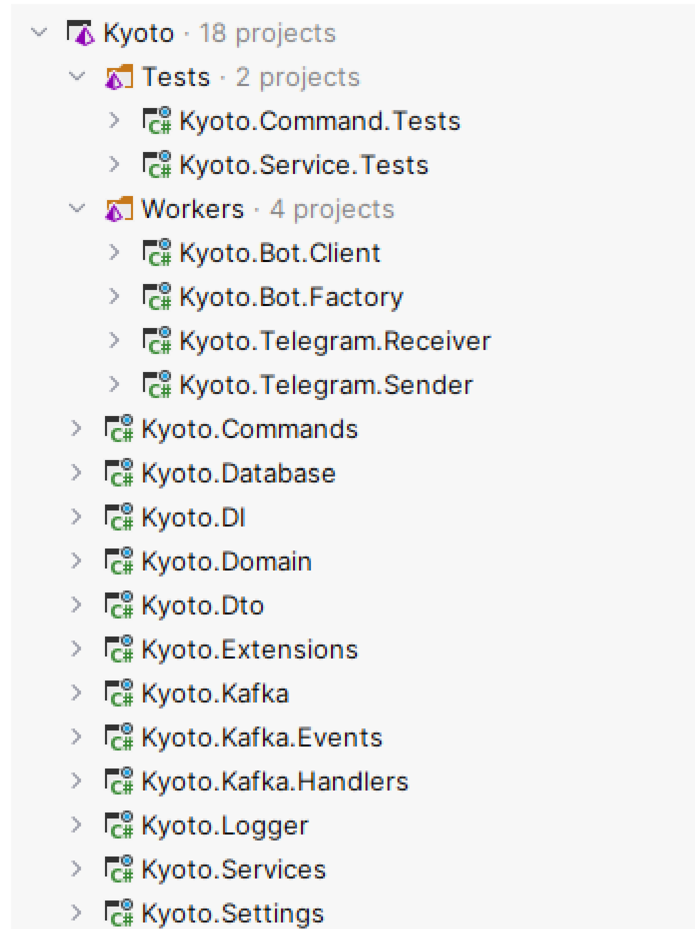


Рисунок 1 – Структура рішення Kyoto

```csharp
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSettings<KyotoBotFactorySettings>(builder.Configuration, out _);
builder.Services.AddSettings<BotTenantSettings>(builder.Configuration, out var botTenantSettings);
builder.Services.AddSettings<DatabaseSettings>(builder.Configuration, out var databaseSettings);
builder.Services.AddSettings<KafkaSettings>(builder.Configuration, out var kafkaSettings);

//Infrastructure
builder.Services
    .AddKafka(kafkaSettings)
    .AddDatabaseBotFactory(databaseSettings, botTenantSettings.Key)
    .AddBotFactoryDeploy()
    .AddTenant(botTenantSettings)
    .AddRequestService()
    .AddTransient<IKafkaEventSubscriber, KafkaEventSubscriber>();

//Functional
builder.Services
    .AddAuthorizationServices()
    .AddMenu()
    .AddBot()
    .AddProcessorServices()
    .AddPostService()
    .AddUser()
    .AddFactoryCommands()
    .AddCommandSystem()
    .AddPreparedMessages()
    .AddTemplateMessage();

//HttpServices
builder.Services
    .AddHttpServices();

//Logging
builder.Logging.AddLogger(builder.Configuration, kafkaSettings);

var app :WebApplication = builder.Build();
AppContext.SetSwitch("Npgsql.EnableLegacyTimestampBehavior", isEnabled: true);

var kafkaConsumerFactory = app.Services.GetRequiredService<IKafkaConsumerFactory>();
var consumerConfig = new ConsumerConfig{BootstrapServers = kafkaSettings.BootstrapServers};
await kafkaConsumerFactory.SubscribeAsync<InitTenantEvent, InitFactoryTenantHandler>(consumerConfig);
await kafkaConsumerFactory.SubscribeAsync<RemoveTenantEvent, RemoveTenantHandler>(consumerConfig, groupId: $"{nameof(RemoveTenantHandler)}-Factory");

await app.Services.SubscribeToRequestTenantEventAsync(kafkaSettings);
await app.Services.SendRequestBotTenantsAsync();

await app.RunAsync();
```

Рисунок 2 – Program – проєкту Kyoto.Bot.Facotry

```
using Kyoto.Kafka.Event;
using Kyoto.Kafka.Handlers;
using Kyoto.Kafka.Interfaces;
using Kyoto.Services.Tenant;
using Kyoto.Settings;

namespace Kyoto.Bot.Factory;

1 usage  Maksym Fedorenko
public class KafkaEventSubscriber : BaseKafkaEventSubscriber
{
    Maksym Fedorenko
    public KafkaEventSubscriber(
        IKafkaConsumerFactory kafkaConsumerFactory,
        KafkaSettings kafkaSettings) : base(kafkaConsumerFactory, kafkaSettings)
    {
    }

    0+3 usages  Maksym Fedorenko
    public override async Task SubscribeAsync(string tenantKey)
    {
        await KafkaConsumerFactory.SubscribeAsync<MessageEvent, MessageHandler>(ConsumerConfig, topicPrefix: tenantKey);
        await KafkaConsumerFactory.SubscribeAsync<CallbackQueryEvent, CallbackQueryHandler>(ConsumerConfig, topicPrefix: tenantKey);
        await KafkaConsumerFactory.SubscribeAsync<CommandEvent, CommandHandler>(ConsumerConfig, topicPrefix: tenantKey);
        await KafkaConsumerFactory.SubscribeAsync<DeployStatusEvent, DeployStatusHandler>(ConsumerConfig, groupId: $"{nameof(DeployStatusHandler)}-Factory");
    }
}   Fedorenko, 13-May-23 19:19 • Added menu processing
```

Рисунок 3 – KafkaEventSubscriber – підписка на топіки Kafka

```
using ...

namespace Kyoto.Bot.Factory;

1 usage  Maksym Fedorenko
public class InitFactoryTenantHandler : IKafkaHandler<InitTenantEvent>
{
    private readonly IKafkaEventSubscriber _kafkaEventSubscriber;
    private readonly IDeployService _deployService;

    Maksym Fedorenko
    public InitFactoryTenantHandler(IKafkaEventSubscriber kafkaEventSubscriber, IDeployService deployService)
    {
        _kafkaEventSubscriber = kafkaEventSubscriber;
        _deployService = deployService;
    }

    Maksym Fedorenko
    public async Task HandleAsync(InitTenantEvent initTenantEvent)
    {
        var isNew = BotTenantFactory.Store.AddOrUpdateTenant(
            BotTenantModel.Create(
                initTenantEvent.TenantKey,
                initTenantEvent.Token,
                initTenantEvent.IsFactory));

        if (isNew && initTenantEvent.IsFactory)
        {
            await _kafkaEventSubscriber.SubscribeAsync(initTenantEvent.TenantKey);
            await _deployService.DeployAsync(new InitTenantInfo
            {
                TenantKey = initTenantEvent.TenantKey,
                TemplateMessages = "TemplateMessages.json"
            }); // Task
        }
    }
}   Fedorenko, 13-May-23 19:19 · Added menu processing
```

Рисунок 4 – InitFactoryTenantHandler – ініціалізація тенантів

```
using ...

namespace Kyoto.Telegram.Receiver.Controllers;

[ApiController]
[Route("api/update")]
public class UpdateController : ControllerBase
{
    private readonly IUpdateService _updateService;
    private const string TelegramTenantHeader = "X-Telegram-Bot-Api-Secret-Token";

    public UpdateController(IUpdateService updateService)
    {
        _updateService = updateService;
    }

    public async Task GetUpdate([FromBody, Required] UpdateDto updateDto)
    {
        Request.Headers.TryGetValue(TelegramTenantHeader, out var tenantKey);
        await _updateService.HandleAsync(tenantKey.ToString(), updateDto.ToDomain());
    }
}
```

Рисунок 5 – UpdateController – контролер для прийняття оновлень з Telegram

```
namespace Kyoto.Domain.CommandSystem.Interfaces;

1 usage  14 inheritors  Maksym Fedorenko
public interface ICommandStep
{
    1 implementation  Maksym Fedorenko
    Task<CommandStepResult> SendActionRequestAsync();
    1 implementation  Maksym Fedorenko
    Task<CommandStepResult> ProcessResponseAsync();
    1 implementation  Maksym Fedorenko
    Task<CommandStepResult> SendRetryActionRequestAsync();
}
```

Рисунок 6 – ICommandStep – інтерфейс для реалізації кроку обробки команди

```
◇ 0+2 usages  Maksym Fedorenko *
protected override async Task<CommandStepResult> SetActionRequestAsync()
{
    var botModel = CommandContext.AdditionalData!.ToObject<BotModel>();
    botModel = await _botRegistrationHttpService.EnrichBotInfoAsync(botModel);
    await _postService.SendConfirmationMessageAsync(Session, text: BuildConfirmNameQuestion(botModel));
    CommandContext.SetAdditionalData(botModel.ToJson());
    return CommandStepResult.CreateSuccessful();
}
```

Рисунок 7 – Приклад реалізації SetActionRequestAsync

```
◇ 0+1 usages  Maksym Fedorenko
protected override async Task<CommandStepResult> SetProcessResponseAsync()
{
    var botModel = JsonConvert.DeserializeObject<BotModel>(CommandContext.AdditionalData!)!;

    if (CommandContext.CallbackQuery!.Data == CallbackQueryButtons.Confirmation)
    {
        await _postService.DeleteMessageAsync(Session);
        await _postService.SendTextMessageAsync(Session,
            text: $"{BuildConfirmNameQuestion(botModel)}\nAnswer: {CallbackQueryButtons.Confirmation}"); // Task

        await _botService.SaveAsync(Session, botModel);
        await _postService.SendTextMessageAsync(Session,
            text: "The bot has been successfully registered! 🪄🥰"); // Task

        await _menuRepository.EnableMenuAsync(MenuPanelConstants.BotFeaturesMenuPanel);
        await _postService.SendTextMessageAsync(Session,
            text: "Now you can go to the menu: 🪄 Features of the bot.\nAnd customize your bot!"); // Task

        return CommandStepResult.CreateSuccessful();
    }

    return CommandStepResult.CreateRetry();
}
```

Рисунок 8 – Приклад реалізації SetProcessResponseAsync

```
◇ 0+1 usages  Maksym Fedorenko
protected override async Task<CommandStepResult> SetRetryActionRequestAsync()
{
    var botModel = JsonConvert.DeserializeObject<BotModel>(CommandContext.AdditionalData!)!;

    await _postService.DeleteMessageAsync(Session);
    await _postService.SendTextMessageAsync(Session,
        text: $"{BuildConfirmNameQuestion(botModel)}\nAnswer: {CallbackQueryButtons.Cancel}"); // Task
    return CommandStepResult.CreateSuccessful();
}
```

Рисунок 9 – Приклад реалізації SetRetryActionRequestAsync

```csharp
using ...

namespace Kyoto.Database.CommonRepositories.PreparedMessage;

public class PreparedMessagesRepository : IPreparedMessagesRepository
{
    private readonly IDatabaseContext _databaseContext;

    public PreparedMessagesRepository(IDatabaseContext databaseContext)
    {
        _databaseContext = databaseContext;
    }

    public async Task<Guid> GetPostEventIdAsync(string postEventName)
    {
        return (await _databaseContext.Set<PostEventDal>().FirstAsync(x => x.Name == postEventName)).Id;
    }

    public async Task AddNewsletterAsync(Domain.PreparedMessagesSystem.PreparedMessage preparedMessage)
    {
        await _databaseContext.SaveAsync(new CommonModels.PreparedMessage
        {
            Text = preparedMessage.Text,
            PostEvent = await _databaseContext.Set<PostEventDal>().FirstAsync(x=>x.Code == preparedMessage.PostEventCode),
            TimeToSend = preparedMessage.TimeToSend,
            KeyWords = preparedMessage.KeyWords
        });
    }

    public Task<List<PostEvent>> GetEventsAsync()
    {
        return _databaseContext.Set<PostEventDal>().Select(x => x.ToDomain()).ToListAsync();
    }
}
```

Рисунок 10 – Приклад репозиторію для обробки підготовлених повідомлень

```
using ...

namespace Kyoto.DI;

Maksym Fedorenko
public static class CommandExtensions
{
    1 usage  Maksym Fedorenko
    public static IServiceCollection AddFactoryCommands(this IServiceCollection services)
    {
        return services
            .AddTransient<ICommandSet, BotFactoryCommandSet>()
            .AddTransient<ICommandStepFactory, RegistrationCommandStepFactory>()
            .AddTransient<ICommandStepFactory, BotRegistrationCommandStepFactory>()
            .AddTransient<ICommandStepFactory, DeployBotCommandStepFactory>()
            .AddTransient<ICommandStepFactory, DisableBotCommandStepFactory>()
            .AddTransient<ICommandStepFactory, SetRegistrationCommandStepFactory>()
            .AddTransient<ICommandStepFactory, AddNewsletterCommandStepFactory>(); // IServiceCollection
    }

    1 usage  Maksym Fedorenko
    public static IServiceCollection AddClientCommands(this IServiceCollection services)
    {
        return services
            .AddTransient<ICommandSet, BotClientCommandSet>()
            .AddTransient<ICommandStepFactory, RegistrationCommandStepFactory>(); // IServiceCollection
    }

    2 usages  Maksym Fedorenko
    public static IServiceCollection AddCommandSystem(this IServiceCollection services)
    {
        return services
            .AddTransient<ICommandFactory, CommandFactory>()
            .AddTransient<ICommandService, CommandService>()
            .AddTransient<ICommandRepository, CommandRepository>(); // IServiceCollection
    }
}
```

Рисунок 11 – Приклад класу розширення для підключення залежностей до DI

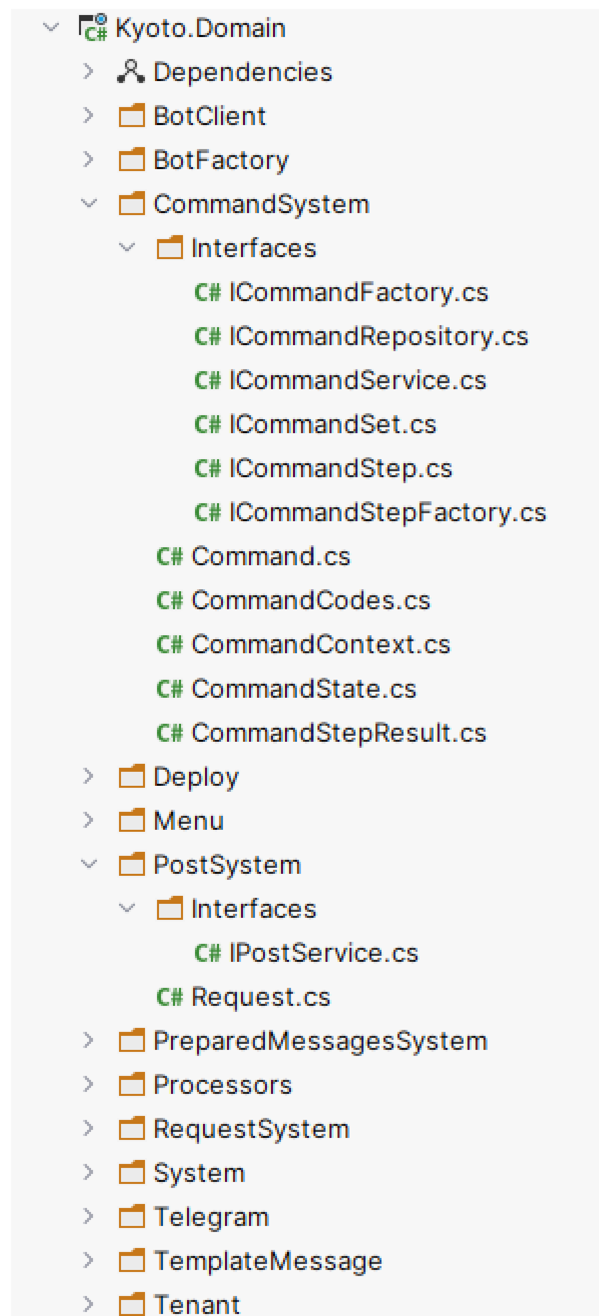


Рисунок 12 – Приклад структури проєкта Kyoto.Domain

```
using ...

namespace Kyoto.Services.Processors;

1 usage  Maksym Fedorenko
public class MessageService : IMessageService
{
    private readonly ICommandService _commandService;
    private readonly IMenuService _menuService;
    private readonly IPostService _postService;

    Maksym Fedorenko
    public MessageService(ICommandService commandService, IMenuService menuService, IPostService postService)
    {
        _commandService = commandService;
        _menuService = menuService;
        _postService = postService;
    }

    0+1 usages  Maksym Fedorenko
    public async Task ProcessAsync(Session session, Message message)
    {
        var text :string = message.Text!;
        var (isExist, command :string ) = await _menuService.TryGetMenuCommandCodeAsync(text);

        if (isExist) {
            await _commandService.ProcessCommandAsync(session, command, message);
        }
        else if(CommandCodes.Cancel == text)
        {
            var canceledCommand :string = await _commandService.CancelCommandAsync(session);
            await _postService.PostAsync(session, new SendMessageRequest(new SendMessageParameters
            {
                Text = $"🙀 Command {canceledCommand} was interrupted",
                ChatId = session.ChatId
            }).ToRequest()); // Task
        }
        else {
            await _commandService.ProcessCommandAsync(session, text, message);
        }

        await _menuService.SendMenuIfExistsAsync(session, message.Text!);
    }
}
```

Рисунок 13 – Приклад обробки повідомлень

```
using Kyoto.Domain.RequestSystem;
using Newtonsoft.Json;

namespace Kyoto.Services.RequestSystem;

1 usage   Maksym Fedorenko
public class RequestService : IRequestService
{
    private readonly HttpClient _httpClient;

    Maksym Fedorenko
    public RequestService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    Maksym Fedorenko
    public Task<HttpResponseMessage> SendAsync(HttpRequestMessage httpRequestMessage)
    {
        return _httpClient.SendAsync(httpRequestMessage);
    }

    0+2 usages   Maksym Fedorenko
    public async Task<bool> SendWithStatusCodeAsync(HttpRequestMessage httpRequestMessage)
    {
        var response = await _httpClient.SendAsync(httpRequestMessage);
        return response.IsSuccessStatusCode;
    }

    0+1 usages   Maksym Fedorenko
    public async Task<T> SendAsync<T>(HttpRequestMessage httpRequestMessage)
    {
        var response = await _httpClient.SendAsync(httpRequestMessage);
        var content :string = await response.Content.ReadAsStringAsync();
        return JsonConvert.DeserializeObject<T>(content)!;
    }
}
```

Рисунок 14 – Клас для відправки повідомлень RequestService

```
◇ 4 usages  Maksym Fedorenko *
public static void AddLogger(
    this ILoggingBuilder loggingBuilder,
    IConfiguration configuration,
    KafkaSettings kafkaSettings,
    string topic = "kafka_bot_factory")
{
    loggingBuilder.ClearProviders();
    loggingBuilder.AddConsole();
    loggingBuilder.AddSerilog(new LoggerConfiguration()
        .ReadFrom.Configuration(configuration)
        .WriteTo.Kafka(
            bootstrapServers: kafkaSettings.BootstrapServers,
            formatter: new GraylogFormatter(),
            securityProtocol: SecurityProtocol.Plaintext,
            saslMechanism: SaslMechanism.Plain,
            topic: $"{topic}.logs"
        )
        .Enrich.WithThreadId()
        .Enrich.FromLogContext()
        .Enrich.WithMachineName()
        .Enrich.WithProcessId()
        .Enrich.WithAssemblyName()
        .Enrich.WithAssemblyVersion()
        .Enrich.WithClientIp()
        .Enrich.WithClientAgent() // LoggerConfiguration
        .CreateLogger());
}
```

Рисунок 15 – Підключення логування

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc( name: "v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Kyoto Bot Client API"
    });

    options.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                },
                Name = "Bearer",
                In = ParameterLocation.Header
            },
            Array.Empty<string>()
        }
    });
});
```

Рисунок 16 – Підключення генерації Swagger

```
using Kyoto.Domain.Tenant;

namespace Kyoto.Bot.Client.Middlewares;

1 usage   Maksym Fedorenko *
public class TenantIdentifierMiddleware
{
    private const string TenantKey = "Tenant";
    private readonly RequestDelegate _next;

    Maksym Fedorenko
    public TenantIdentifierMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    Maksym Fedorenko *
    public async Task InvokeAsync(HttpContext context)
    {
        context.Request.Headers.TryGetValue(TenantKey, out var tenantKey :StringValues);
        if (string.IsNullOrEmpty(tenantKey))
        {
            await _next(context);
            return;
        }

        using (CurrentBotTenant.SetBotTenant(BotTenantModel.Create(tenantKey!)))
        {
            await _next(context);
        }
    }
}
```

Рисунок 17 – Міделвер для перехвату тенанта

```
namespace Kyoto.Domain.System;

73 usages  Maksym Fedorenko  8 exposing APIs
public class Session
{
    4 usages
    public Guid Id { get; private set; }
    24 usages
    public long ChatId { get; private set; }
    16 usages
    public long ExternalUserId { get; private set; }
    3 usages
    public int MessageId { get; private set; }
    7 usages
    public string TenantKey { get; set; }

    4 usages  Maksym Fedorenko
    private Session(Guid id, long chatId, long externalUserId, int messageId, string tenantKey)
    {
        Id = id;
        ChatId = chatId;
        ExternalUserId = externalUserId;
        MessageId = messageId;
        TenantKey = tenantKey;
    }

    1 usage  Maksym Fedorenko
    public static Session Create(Guid id, long chatId, long externalUserId, int messageId, string tenantKey)
    {
        return new Session(id, chatId, externalUserId, messageId, tenantKey);
    }

    2 usages  Maksym Fedorenko
    public static Session CreateNew(long chatId, long externalUserId, int messageId, string tenantKey)
    {
        return new Session( id: Guid.NewGuid(), chatId, externalUserId, messageId, tenantKey);
    }

    Maksym Fedorenko
    public static Session CreateNew(string tenantKey)
    {
        return new Session( id: Guid.NewGuid(),  chatId: default,  externalUserId: default,  messageId: default, tenantKey);
    }

    5 usages  Maksym Fedorenko
    public static Session CreatePersonalNew(string tenantKey, long chatId)
    {
        return new Session( id: Guid.NewGuid(), chatId,  externalUserId: default,  messageId: default, tenantKey);
    }
}
```

Рисунок 20 – Клас який зберігає інформацію про сесію виконання взаємодії з клієнтом

```
using ...

namespace Kyoto.Telegram.Sender.Services;

1 usage  Maksym Fedorenko
public class RequestService : IRequestService
{
    private readonly ITBot _tBot;
    private readonly KyotoBotSenderSettings _kyotoBotSenderSettings;

    Maksym Fedorenko
    public RequestService(ITBot tBot, KyotoBotSenderSettings kyotoBotSenderSettings)
    {
        _tBot = tBot;
        _kyotoBotSenderSettings = kyotoBotSenderSettings;
    }

    0+1 usages  Maksym Fedorenko
    public async Task<HttpResponseMessage> SendAsync(Session session, RequestModel requestModel)
    {
        var botTenantModel = BotTenantFactory.Store.Get(session.TenantKey);
        requestModel.Parameters.TryGetValue("chat_id", out var chatId :string? );

        _tBot.Init( botSettings: new TBotOptions(botTenantModel.Token));

        var request = new BaseRequest(
            requestModel.Endpoint,
            requestModel.HttpMethod,
            requestModel.Parameters.Select(x :KeyValuePair<string,string> => new Parameter(x.Key, x.Value)).ToList());

        if (_kyotoBotSenderSettings.IsLimiterEnable)
        {
            return await _tBot.PostWithLimiterAsync(request, limiterKey: $"{botTenantModel.TenantKey}:{chatId!}");
        }

        return await _tBot.PostAsync(request);
    }
}
```

Рисунок 21 – Клас, який відповідає за відправку повідомлень до Telegram

```
using ...

namespace Kyoto.Kafka.Handlers;

2 usages  Maksym Fedorenko
public class MessageHandler : IKafkaHandler<MessageEvent>
{
    private readonly IServiceProvider _serviceProvider;

    Maksym Fedorenko
    public MessageHandler(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    Maksym Fedorenko
    public async Task HandleAsync(MessageEvent messageEvent)
    {
        using (CurrentBotTenant.SetBotTenant(BotTenantModel.Create(messageEvent.TenantKey)))
        {
            using var scope = _serviceProvider.CreateScope();
            var messageService = scope.ServiceProvider.GetRequiredService<IMessageService>();
            await messageService.ProcessAsync(messageEvent.GetSession(), messageEvent.Message);
        }
    }
}
```

Рисунок 22 – Приклад обробника повідомлень від Kafka. Обробка текстових повідомлень від Telegram

```
using ...

namespace Kyoto.Kafka.Event;

6 usages  6 inheritors  Maksym Fedorenko
public class BaseSessionEvent : BaseEvent
{
    5 usages
    public string TenantKey { get; set; } = null!;
    2 usages
    public long ChatId { get; set; }
    2 usages
    public long ExternalUserId { get; set; }
    2 usages
    public int MessageId { get; set; }

    5 usages  Maksym Fedorenko
    public Session GetSession()
    {
        return Session.Create(SessionId, ChatId, ExternalUserId, MessageId, TenantKey);
    }

    Maksym Fedorenko
    public BaseSessionEvent()
    {

    }

    6 usages  Maksym Fedorenko
    public BaseSessionEvent(Session session)
    {
        SessionId = session.Id;
        TenantKey = session.TenantKey;
        ChatId = session.ChatId;
        MessageId = session.MessageId;
        ExternalUserId = session.ExternalUserId;
    }
}
```

Рисунок 23 – Приклад базового класу для повідомлень Kafka

```
[Fact]
◇ Maksym Fedorenko
public async Task SetActionRequest_WhenPostServiceSuccess()
{
    //Arrange
    const string text = "TestMessage";
    var successResult = CommandStepResult.CreateSuccessful();
    _postService.SendTextMessageAsync( session: Arg.Any<Session>(), text).Returns(Task.CompletedTask);
    var botRegistrationTokenStep = new BotRegistrationTokenStep(_postService);

    //Act
    var result = await botRegistrationTokenStep.SendActionRequestAsync();

    //Assert
    Assert.Equal( expected: successResult.IsRetry,  actual: result.IsRetry);
    Assert.Equal( expected: successResult.IsInterrupt,  actual: result.IsInterrupt);
    Assert.Equal( expected: successResult.ToRetryStep,  actual: result.ToRetryStep);
}

[Fact]
◇ Maksym Fedorenko
public async Task ProcessResponse_WhenPostServiceSuccess()
{
    //Arrange
    const string token = "TOKEN";
    var commandContext = CommandContext.Create(new Message { Text = token },  callbackQuery: null);
    var botRegistrationTokenStep = new BotRegistrationTokenStep(_postService);
    botRegistrationTokenStep.SetCommandContext(commandContext);

    //Act
    await botRegistrationTokenStep.ProcessResponseAsync();

    //Assert
    Assert.NotNull(botRegistrationTokenStep.CommandContext.AdditionalData);
    Assert.NotEmpty(botRegistrationTokenStep.CommandContext.AdditionalData);
}
```

Рисунок 24 – Приклад unit-тестів

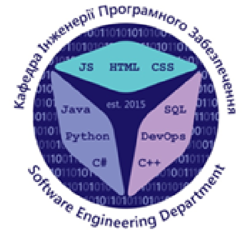
```
namespace Kyoto.Database;

2 usages   2 inheritors   Maksym Fedorenko
public class BaseDatabaseContext : DbContext, IDatabaseContext
{
    private string _connectionString;

    public DbSet<Command>? Commands { get; set; }
    public DbSet<ExternalUser>? ExternalUsers { get; set; }
    public DbSet<TemplateMessage>? TemplateMessages { get; set; }
    public DbSet<TemplateMessageType>? TemplateMessageTypes { get; set; }
    public DbSet<MenuPanel>? MenuPanels { get; set; }
    public DbSet<MenuButton>? MenuButtons { get; set; }
    public DbSet<SystemStatus>? SystemStatuses { get; set; }
    public DbSet<PostEvent>? EventMessages { get; set; }
    public DbSet<PreparedMessage>? PreparedMessages { get; set; }
    public DbSet<Feedback>? Feedbacks { get; set; }
    public DbSet<PreOrderService>? PreOrderServices { get; set; }
    public DbSet<Service>? Services { get; set; }

    2 usages   Maksym Fedorenko
    public BaseDatabaseContext(string connectionString)
    {
        _connectionString = connectionString;
    }

    Maksym Fedorenko
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseNpgsql(_connectionString);
    }

    0+1 usages   Maksym Fedorenko
    public async Task MigrateAsync(string connectionString, CancellationToken cancellationToken = default)
    {
        _connectionString = connectionString;
        await Database.MigrateAsync(cancellationToken);
    }
```

Рисунок 25 – Приклад базового класу контексту бази даних

ДОДАТОК Б

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



РОЗРОБКА ЧАТ-БОТА-ФАБРИКИ ДЛЯ ДОПОМОГИ МАЛОМУ БІЗНЕСУ В ТЕЛЕГРАМ МОВОЮ C# НА ПЛАТФОРМІ ASP.NET

Виконав студент 4 курсу
групи ПД-41
Федоренко Максим Леонідович
Керівник роботи
К.т.н, доц, доцент кафедри ІПЗ Негоденко Олена Василівна

Київ - 2023

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Об’єкт дослідження - просування малого бізнесу за допомогою чат-бота в Telegram.

Предмет дослідження - Telegram чат-бот фабрика на базі мікросервісів з мультитенантною архітектурою.

Мета роботи - підвищити ефективність просування малого бізнесу за допомогою бот фабрики на платформі ASP.NET мовою програмування C# із використанням мікросервісної архітектури.

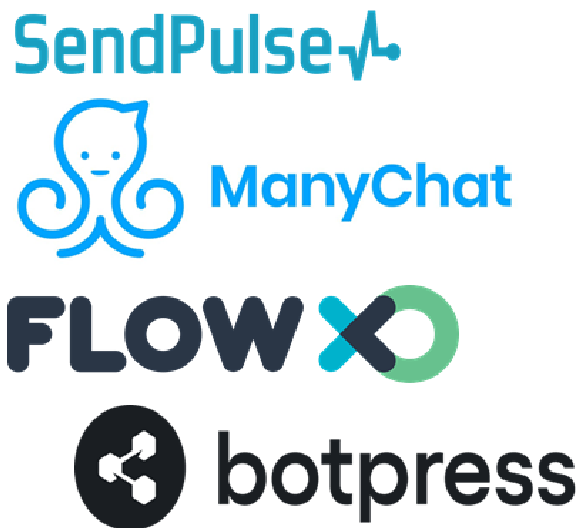
2

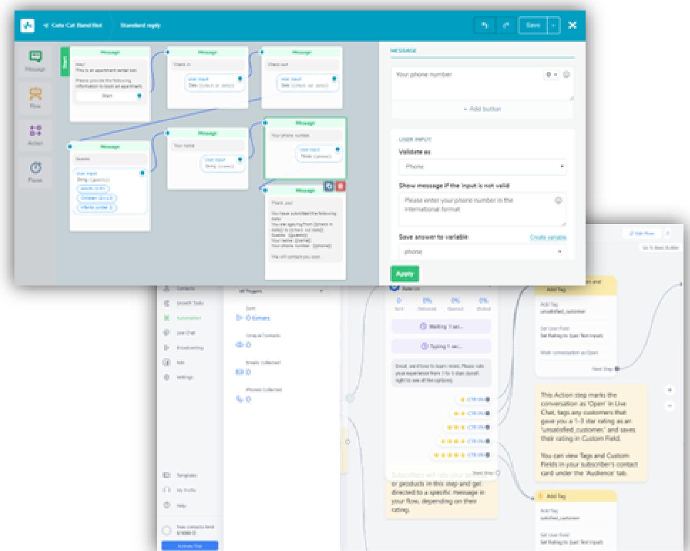
ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проаналізувати існуючі сучасні аналогічні програмні забезпечення та виявити їх переваги та недоліки. І як результат створити порівняльну таблицю.
2. Розробити функціональні та нефункціональні вимоги до програмного забезпечення.
3. Дослідити технології, інструменти та програмні засоби для вирішення поставленої задачі.
4. Розробити програмне забезпечення для спрощення створення Telegram чат бота з урахуванням прийнятих раніше технологій та поставлених заздалегідь вимог.
5. Провести тестування програмного забезпечення.
6. Пройти апробацію на Науково-технічних конференціях.

3

АНАЛІЗ АНАЛОГІВ





4

АНАЛІЗ АНАЛОГІВ

Показник	SendPulse	ManyChat	Botpress	Flow XO	**Kyoto Bot Factory**
Налаштування через зовнішні ресурси	+	+	+	+	-
Налаштування через чат Telegram	-	-	-	-	+
Налаштування за допомогою API	-	-	-	-	+
Life Chat	+	+	-	+	-
Збір відгуків	-	-	-	-	+
Генерування QR-Code - запрошення	-	-	-	-	+
Статистика використання	+	+	+	+	+
OpenSource	-	-	+	-	+
Інтеграція з іншими сервісами	+	+	+	+	-
Шаблонний функціонал	-	-	-	-	+
Навантаженість інтерфейсу	**Середня**	**Середня**	**Висока**	**Середня**	**Низька**
Розгортання на свої серверах	+	+	-	+	+

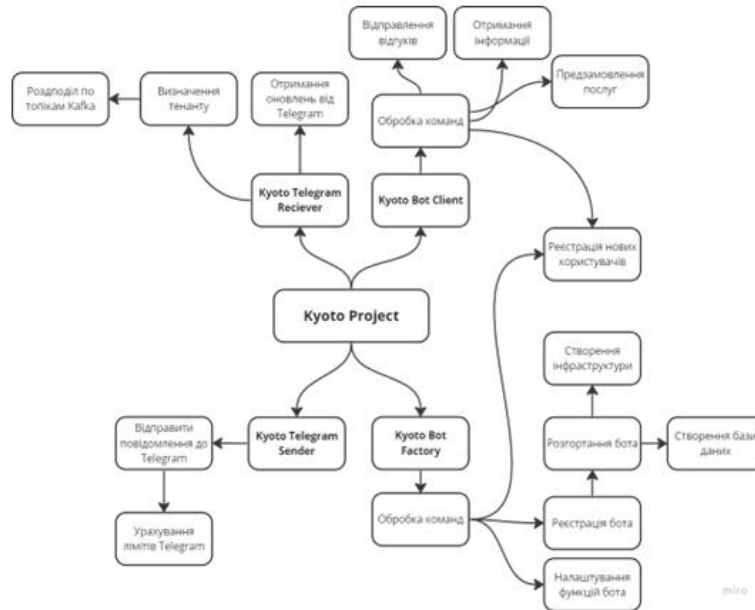
5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1. Реєстрація у системі
2. Реєстрація та розгортання нового бота.
3. Додавання та видалення повідомлень для розсилки
4. Включення та виключення збору відгуків від клієнтів
5. Отримання статистики використання бота клієнтами
6. Відключення чат-бота
7. Видалення чат-бота
8. Можливість робити передзамовлення послуг
9. Налаштування бота клієнта за допомогою API

6

Асоціативна мапа функціональності

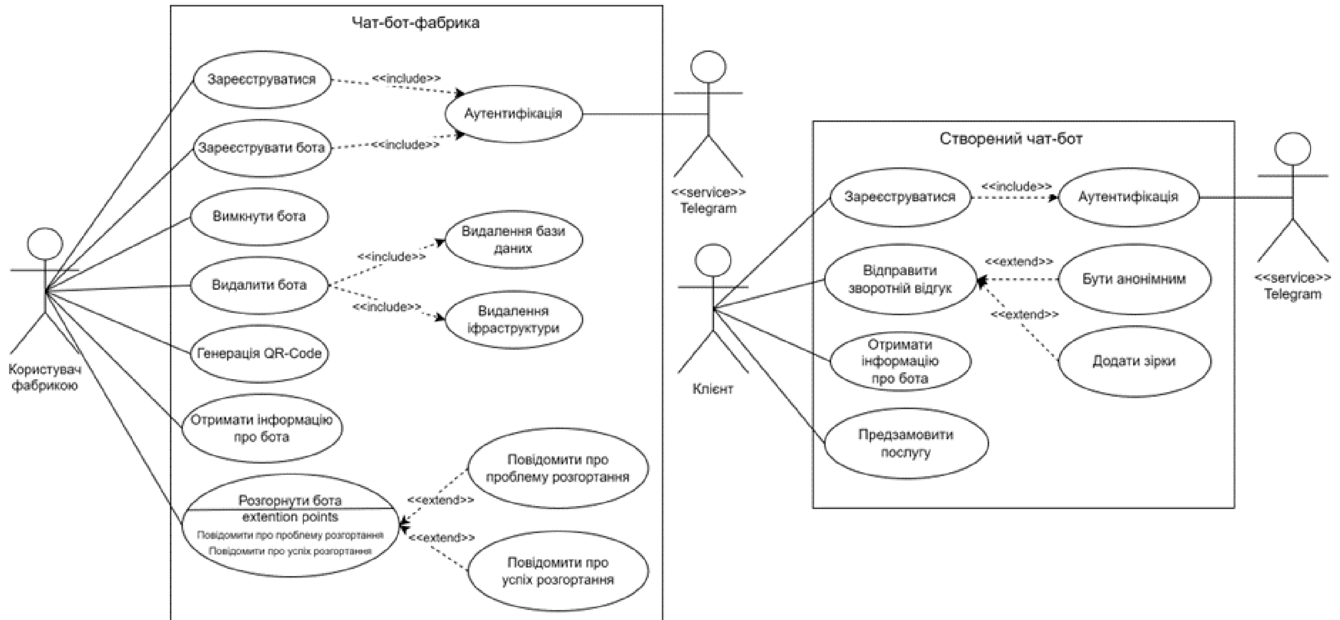


7

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

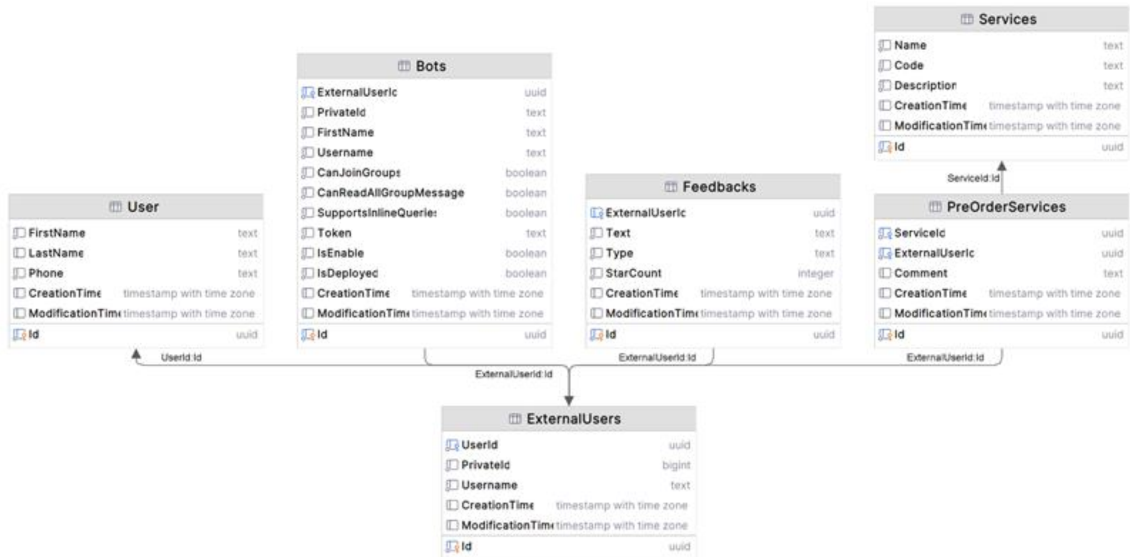
8

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



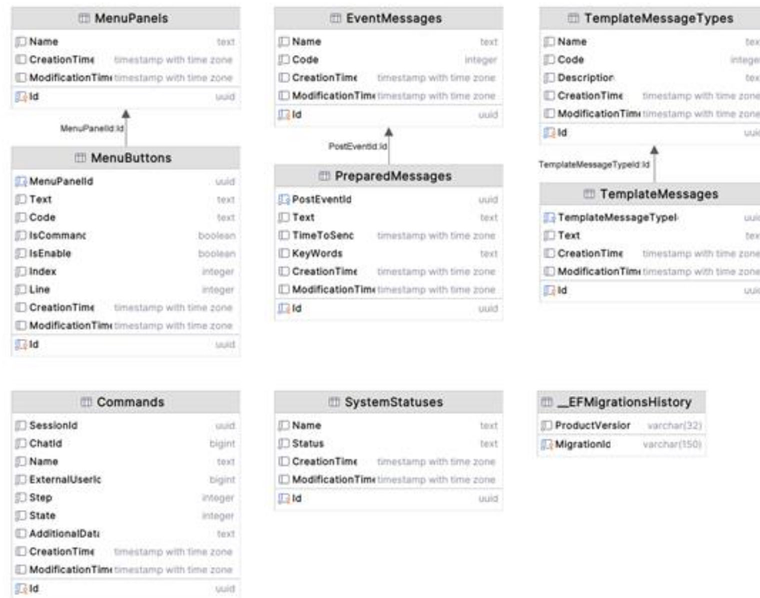
9

ДІАГРАМА ЗВ'ЯЗКІВ СУТНОСТЕЙ



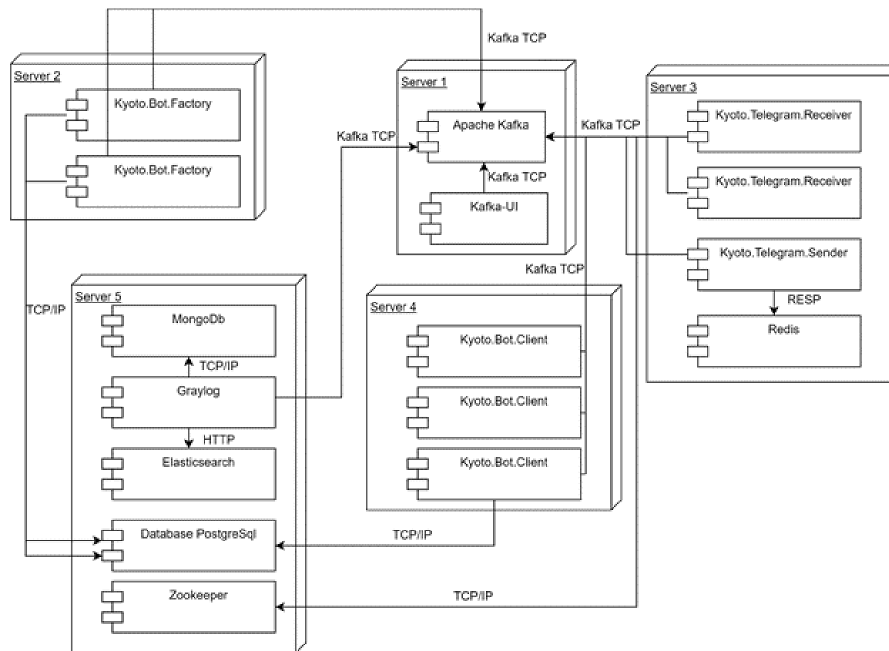
10

ДІАГРАМА ЗВ'ЯЗКІВ СУТНОСТЕЙ



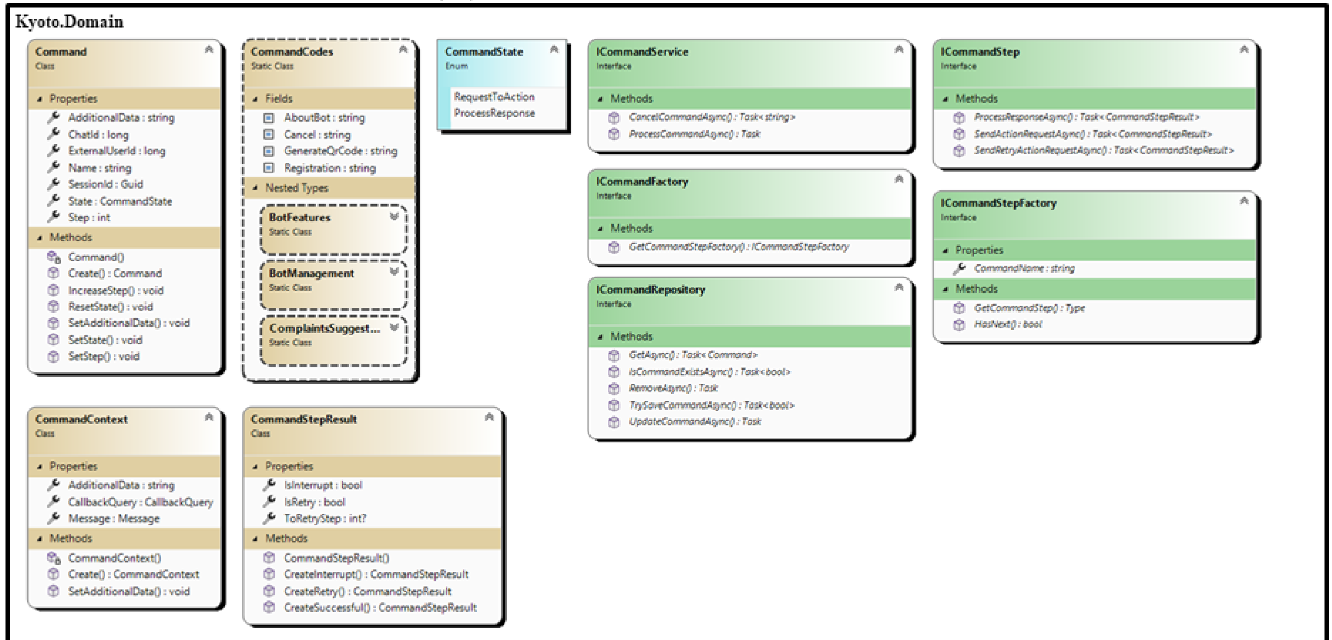
11

ДІАГРАМА РОЗГОРТАННЯ



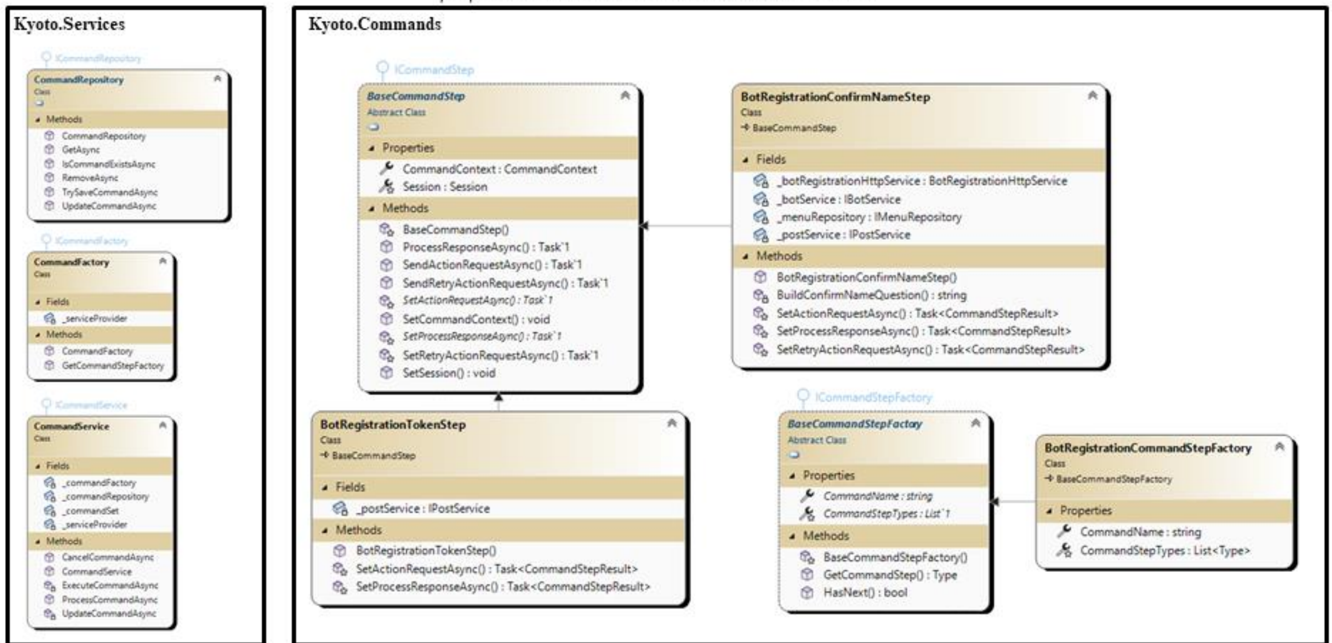
12

ДІАГРАМА КЛАСІВ



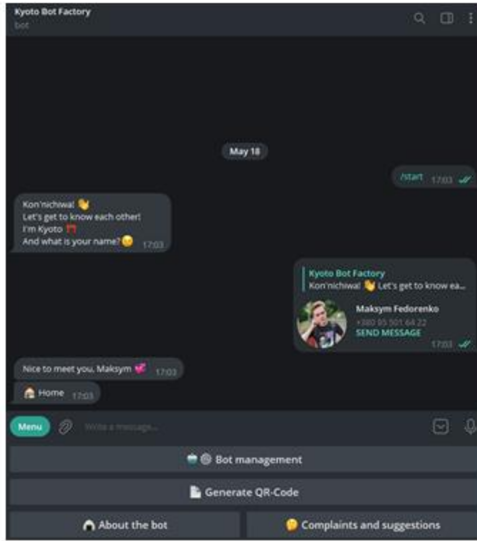
13

ДІАГРАМА КЛАСІВ

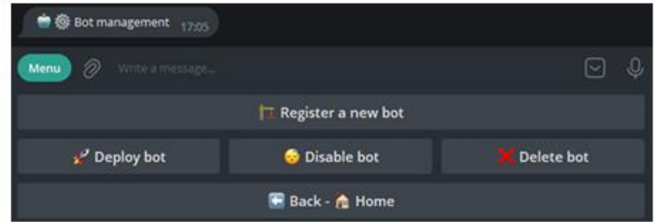


14

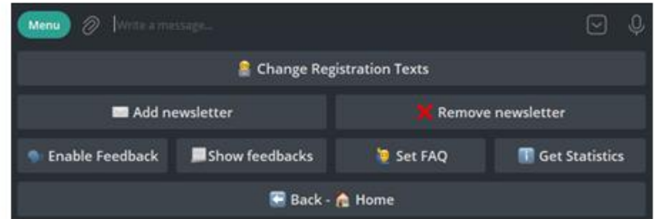
ЕКРАНІ ФОРМИ ЧАТ-БОТА-ФАБРИКИ



Початок роботи з Kyoto Bot Factory



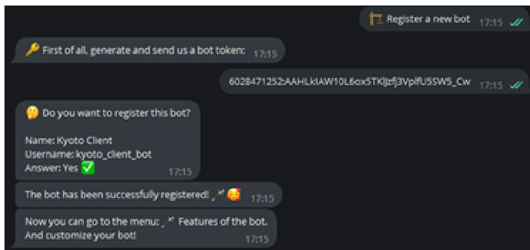
Меню управління телеграм ботами у Kyoto Bot Factory



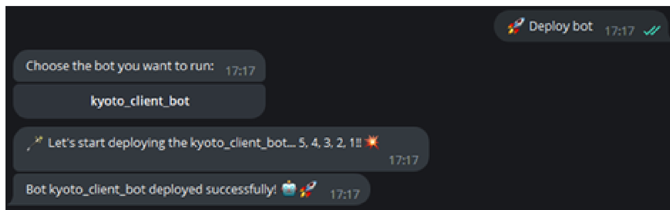
Меню управління функціоналом створеного бота

15

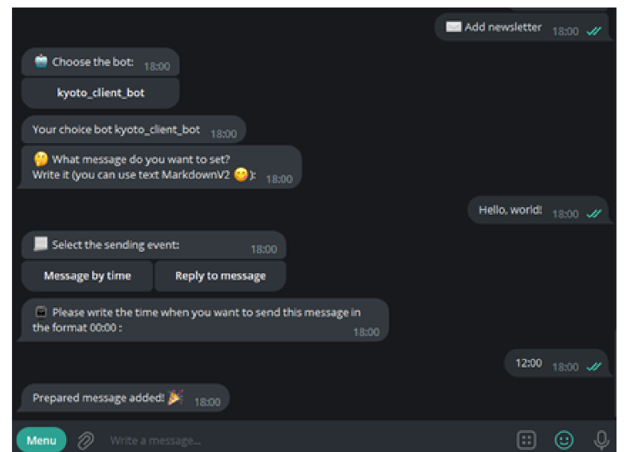
ЕКРАНІ ФОРМИ ЧАТ-БОТА-ФАБРИКИ



Реєстрація нового телеграм бота



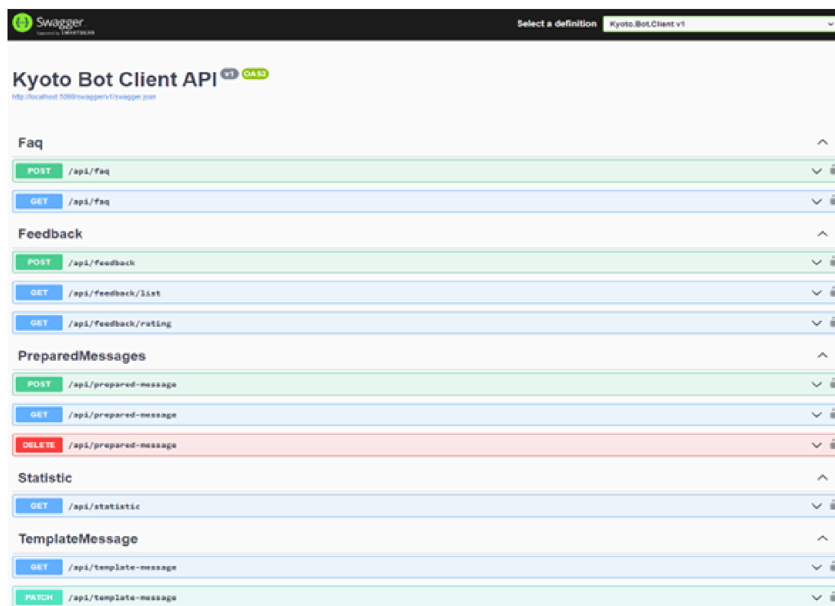
Меню управління телеграм ботами у Kyoto Bot Factory



Процес реєстрації нового повідомлення для розсилки

16

ЕКРАНІ ФОРМИ ЧАТ-БОТА-ФАБРИКИ



Swagger Kyoto Bot Client

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Федоренко М.Л. Використання чат боту для взаємодії з клієнтом через соціальні мережі: Матеріали II Всеукраїнської наукової конференції студентів та молодих вчених "Наукові досягнення та відкриття сучасної молоді". Збірник тез.- м. Покровськ/м. Луцьк, Донецький національний технічний університет, 31 травня 2023
2. Федоренко М.Л. Інтеграція штучного інтелекту з месенджером телеграм для покращення взаємодії користувачів з чат-ботами компанії: Матеріали III Всеукраїнської науково-практичної конференції "Сучасні інтелектуальні інформаційні технології в науці та освіті". Збірник тез. 16.05.2023, ДУТ, м. Київ — К.: ДУТ, 2023.
3. Федоренко М.Л. Розробка чат-бота-фабрики для допомоги малому бізнесу в телеграм: Матеріали Всеукраїнської науково-технічної конференції «Сучасний стан та перспективи розвитку IoT». Збірник тез. 07.04.2023, ДУТ, м. Київ — К.: ДУТ, 2023. — С. 226.

ВИСНОВКИ

1. Проведено аналіз предметної області та виявлено чотири веб-сервіси зі схожим набором функціоналу, а саме: SendPulse, ManyChat, Botpress, Flow XO. Визначено переваги і недоліки даних веб-сервісів.
2. Розроблено технічне завдання, визначено функціональні та нефункціональні вимоги щодо розробки програмного забезпечення.
3. Досліджено найсучасніші засоби та інструменти розробки програмного забезпечення з урахуванням особливостей предметної області.
4. Розроблено програмне забезпечення для спрощення створення особистого Telegram чат-бота з використанням мікросервісної архітектури та підвищення ефективності просування малого бізнесу.
5. Протестовано програмний додаток на стабільність роботи.

ДЯКУЮ ЗА УВАГУ!