Міністерство освіти і науки України
Кривий Ріг національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти магістра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Дослідження та застосування API різних веб-сервісів для створення інтегрованої платформи комунікації

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.

Студент гр. ІПЗ-23-1м
_________ /В. М. Д’яконов/

Керівник
кваліфікаційної роботи _________________ /А. М. Стрюк/

Завідувач кафедри _________________ /А. М. Стрюк/

Кривий Ріг
2024

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: магістр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри

__________ А. М. Стрюк

«___» _________ 20__ р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-23-1м Д’яконову Владиславу Михайловичу

1. Тема: Дослідження та застосування API різних веб-сервісів для створення інтегрованої платформи комунікації затверджено наказом по КНУ № 6 від «29» січня 2024 р.
2. Термін подання студентом закінченої роботи: «15» грудня 2024 р.
3. Вихідні дані по роботі: програмне забезпечення повинно включати інтеграцію API різних веб-сервісів, забезпечення комунікації між платформами, управління обміном даних.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): аналіз предметної області та особливостей використання API веб-сервісів, огляд сучасних рішень для інтеграції веб-сервісів, визначення вимог до інтегрованої платформи комунікації, проектування архітектури платформи та її компонентів, розробка схеми взаємодії.
5. Перелік ілюстративного матеріалу: функціональна схема, структурна схема, UML-діаграма бази даних, UML-діаграми класів, діаграма варіантів використання, рисунок прототипу інтерфейсу.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Формулювання актуальності та наукового апарату дослідження	15.07.2024 - 31.07.2024
2	Дослідження типів веб-сервісів та API	01.08.2024 - 15.08.2024
3	Аналіз існуючих рішень та досліджень	16.08.2024 - 31.08.2024
4	Визначення функціональних та нефункціональних вимог	01.09.2024 - 15.09.2024
5	Розробка функціональної та структурної схем	16.09.2024 - 30.09.2024
6	Проектування бази даних та UML-діаграм	01.10.2024 - 15.10.2024
7	Розробка візуального інтерфейсу платформи	16.10.2024 - 31.10.2024
8	Програмна реалізація основних класів проекту	01.11.2024 - 15.11.2024
9	Розробка посібника користувача	16.11.2024 - 18.11.2024
10.	Тестування системи	18.11.2024 - 19.11.2024
11.	Оформлення пояснювальної записки	20.11.2024 - 29.11.2024

Дата видачі завдання: «29» січня 2024 р.

Студент ________________ /В. М. Д’яконов/

Керівник роботи ________________ /А. М. Стрюк/

РЕФЕРАТ

ЦИФРОВА ТРАНСФОРМАЦІЯ, API, ІНТЕГРАЦІЯ, КОМУНІКАЦІЙНА ПЛАТФОРМА, БЕЗПЕКА ДАНИХ, ВЕБ-СЕРВІСИ, ПЕРСОНАЛІЗАЦІЯ, МАСШТАБОВАНІСТЬ, КОНКУРЕНТНА ПЕРЕВАГА, ПРОТОТИПУВАННЯ.

Пояснювальна записка: 73 с., 17 рис., 1 дод., 10 джерел.

Метою даної кваліфікаційної роботи є розробка моделі інтегрованої комунікаційної платформи з використанням API різних веб-сервісів. У роботі проаналізовано існуючі веб-сервіси та можливості їх інтеграції, а також визначено основні вимоги до побудови інтеграційної платформи, яка гарантує ефективну взаємодію між різними сервісами. Розглянуто аспекти, пов'язані з безпекою, персоналізацією та масштабованістю платформи, а також проаналізовано ключові фактори для успішної інтеграції API.

В результаті аналізу було обрано методи інтеграції для поєднання різних веб-сервісів, що дозволяє зручніше та ефективніше управляти даними та комунікаціями. Розроблено концептуальну модель платформи, яка описує архітектуру взаємодії компонентів і забезпечує стандартизовану інтеграцію.

Розроблена платформа була протестована та перевірена на функціональність, безпеку та продуктивність. Тестування дозволило виявити недоліки, які потребують подальшої роботи для покращення ефективності та надійності інтегрованого рішення.

ABSTRACT

DIGITAL TRANSFORMATION, API, INTEGRATION, COMMUNICATION PLATFORM, DATA SECURITY, WEB SERVICES, PERSONALIZATION, SCALABILITY, COMPETITIVE ADVANTAGE, PROTOTYPING.

Explanatory note: 73 p., 17 imgs., 1 appendix, 10 sources.

The aim of this thesis is to develop a model of an integrated communication platform using APIs of various web services. The paper analyzes existing web services and their integration possibilities, as well as defines the main requirements for building an integration platform that ensures effective interaction between different services. Aspects related to security, personalization, and scalability of the platform are considered, and key factors for the successful integration of APIs are analyzed.

As a result of the analysis, integration methods were chosen to combine various web services, allowing for more convenient and efficient management of data and communications. A conceptual model of the platform was developed, describing the architecture of component interaction and ensuring standardized integration.

The developed platform was tested and evaluated for functionality, security, and performance. The testing revealed shortcomings that require further work to improve the effectiveness and reliability of the integrated solution.

ЗМІСТ

ВСТУП 8

РОЗДІЛ 1 АКТУАЛЬНІСТЬ І НАУКОВИЙ АПАРАТ ДОСЛІДЖЕННЯ 10

1.1 Актуальність теми кваліфікаційної роботи 10

1.2 Науковий апарат дослідження 11

1.2.1 Мета дослідження 12

1.2.2 Об'єкт дослідження 12

1.2.3 Предмет дослідження 12

1.2.4 Завдання дослідження 12

1.2.5 Методи дослідження 13

РОЗДІЛ 2 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ НАЯВНИХ РІШЕНЬ 14

2.1 Особливості предметної області 14

2.1.1 Типи веб-сервісів та їхня функціональність 14

2.1.2 Типи API та їхні особливості 15

2.1.3 Виклики та обмеження інтеграції 15

2.1.4 Переваги інтеграції 15

2.2 Огляд та аналіз досліджень на тему інтеграції веб-сервісів 16

2.2.1 Сучасні тенденції та технології 16

2.2.2 Переваги та обмеження наявних рішень 17

2.3 Визначення вимог до інтегрованої платформи 19

2.3.1 Функціональні вимоги 20

2.3.2 Нефункціональні вимоги 21

2.3.3 Вимоги безпеки 22

РОЗДІЛ 3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПЛАТФОРМИ 24

3.1 Розробка функціональної схеми 24

3.2 Розробка структурної схеми 24

3.3 Розробка UML–діаграми структури бази даних 25

3.4 Розробка діаграми варіантів використання 26

3.5 Розробка структур таблиць бази даних та вибір типів даних полів... 26

3.6 Розробка моделі візуального інтерфейсу 27

3.7 Розробка UML-діаграм класів платформи 29

РОЗДІЛ 4 РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ 36

4.1 Розробка основних класів проекту 36

4.2 Розробка візуального інтерфейсу платформи 52

4.3 Розробка посібника користувача 53

ВИСНОВОК 56

ПЕРЕЛІК ПОСИЛАНЬ 57

Додаток А – Код програми 58

ВСТУП

У сучасному світі, який характеризується швидкою діджиталізацією та інтеграцією інформаційних технологій, здатність до взаємодії між різними веб-сервісами стає вирішальним фактором успіху компаній, соціальних ініціатив та індивідуальних проектів. Розробка інтегрованих рішень, які поєднують функції різних платформ, має вирішальне значення для підвищення ефективності роботи користувачів та організацій.

API (Application Programming Interfaces – інтерфейси прикладного програмування) є найважливішим засобом реалізації таких рішень. API надають доступ до функцій сторонніх сервісів, таких як соціальні мережі, месенджери, системи управління контентом, хмарні сховища тощо. Таким чином можна створювати програмні платформи, які інтегрують дані та функції цих сервісів в єдину екосистему, що відповідає сучасним вимогам зручності та продуктивності.

Актуальність даної роботи обумовлена зростаючою залежністю від інтеграційних рішень, які спрощують процеси комунікації, оптимізують взаємодію між сервісами та дозволяють автоматизувати рутинні завдання. У цьому контексті розробка інтегрованої комунікаційної платформи, що використовує можливості API поширених веб-сервісів, відкриває нові горизонти для користувачів у сфері бізнесу, освіти, культури та інших галузях.

Метою дипломної роботи є створення моделі інтегрованої комунікаційної платформи, яка надає зручний інструмент для обміну інформацією між користувачами, полегшуючи доступ до функцій декількох сервісів в єдиному інтерфейсі. Для досягнення поставленої мети в роботі поставлені наступні завдання:

- проаналізувати існуючі API популярних веб-сервісів, їх можливості та обмеження;
- спроектувати та розробити інтегровану архітектуру;

- забезпечити безпеку даних, масштабованість та зручність використання.

У дослідженні будуть використані сучасні технології розробки, а також підходи до інтеграції та оптимізації API для забезпечення надійності та ефективності запропонованого рішення.

Дипломна робота сприятиме розумінню принципів побудови інтегрованих систем, що сприяють підвищенню ефективності та зручності цифрових комунікацій, а також закладе фундамент для майбутніх досліджень та вдосконалення таких платформ.

РОЗДІЛ 1 АКТУАЛЬНІСТЬ І НАУКОВИЙ АПАРАТ ДОСЛІДЖЕННЯ

1.1 Актуальність теми кваліфікаційної роботи

Сучасне суспільство живе в епоху цифрової трансформації, коли технології відіграють ключову роль у повсякденному житті, бізнесі та науці. У зв'язку зі зростаючим обсягом даних і швидкістю їхнього обміну, що постійно зростає, інтеграція різних веб-сервісів стає не тільки бажаною, а й необхідною. Це особливо актуально в умовах глобалізації та цифровізації, коли ефективна комунікація та управління інформацією стають найважливішими факторами успіху.

Одним із найважливіших інструментів, що забезпечують взаємодію між різними додатками та сервісами, є інтерфейс прикладного програмування (API). API забезпечують стандартизовані способи взаємодії та обміну даними, даючи змогу різним системам інтегруватися одна з одною. Це відкриває нові можливості для створення інтегрованих платформ, які можуть об'єднувати функціональність різних веб-сервісів, надаючи користувачам єдине і зручне рішення для управління інформацією та комунікаціями.

Актуальність цієї роботи зумовлена кількома ключовими факторами:

1. зростаюча кількість веб-сервісів і додатків. Кількість веб-сервісів, що пропонують різні функції, зростає з кожним роком. Управління кількома платформами стає дедалі складнішим і потребує інтеграції для підвищення ефективності;
2. потреба в централізованому управлінні. Для компаній та індивідуальних користувачів важливо мати можливість централізованого управління кількома сервісами, що дає змогу заощаджувати час і ресурси, покращувати робочі процеси та підвищувати продуктивність;
3. тенденція до персоналізації. Сучасні користувачі очікують персоналізованого досвіду під час взаємодії з цифровими сервісами.

Інтеграція різних API дає змогу створювати більш персоналізовані та адаптовані рішення, що відповідають індивідуальним потребам кожного користувача;

4. безпека і надійність. У зв'язку зі зростанням кількості кібератак і загроз безпеці інтеграція веб-сервісів відповідно до сучасних стандартів безпеки стає важливим аспектом. Використання перевірених API дає змогу підвищити загальний рівень безпеки та надійності платформи;
5. конкурентна перевага. Компанії, які успішно інтегрують кілька веб-сервісів на одній платформі, отримують конкурентну перевагу, оскільки можуть запропонувати своїм клієнтам більш зручні та функціональні рішення. Це, своєю чергою, допомагає утримувати клієнтів і залучати нових.

Таким чином, вивчення і впровадження API різних веб-сервісів для створення інтегрованої комунікаційної платформи є актуальним завданням, що відповідає сучасним вимогам до управління інформацією та взаємодії в цифровому просторі. Це дослідження дасть змогу не тільки поглибити розуміння поточних можливостей і обмежень API, а й запропонувати нові підходи до їхньої інтеграції, що буде корисним як для академічного співтовариства, так і для IT-фахівців.

1.2 Науковий апарат дослідження

Науковий апарат дослідження містить формулювання мети, визначення об'єкта і предмета дослідження, постановку завдань і вибір методів, які будуть використовуватися для досягнення поставлених цілей. Цей розділ відіграє ключову роль у структуруванні та цілеспрямованості дослідження, забезпечує логічну зв'язність і повноту роботи.

1.2.1 Мета дослідження

Метою цього дослідження є розробка моделі інтегрованої комунікаційної платформи, заснованої на використанні API різних веб-сервісів. Дана модель повинна забезпечувати ефективну взаємодію між різними сервісами, задовольняти функціональні та нефункціональні вимоги користувачів, а також відповідати сучасним стандартам безпеки та масштабованості.

1.2.2 Об'єкт дослідження

Об'єктом дослідження є сучасні веб-сервіси та їхні API, що надають можливості для інтеграції та обміну даними. Це можуть бути соціальні мережі, системи управління контентом, месенджери, хмарні сховища та інші популярні платформи, які активно використовуються в особистій та професійній діяльності користувачів.

1.2.3 Предмет дослідження

Предметом дослідження є методи і технології інтеграції веб-сервісів за допомогою API для створення єдиної платформи комунікації. У цьому контексті особлива увага приділяється особливостям різних API, їхній сумісності, функціональним можливостям і обмеженням, а також методам забезпечення безпеки та захисту даних під час інтеграції.

1.2.4 Завдання дослідження

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. провести огляд і аналіз наявних веб-сервісів та їхніх API, вивчити їхні можливості та обмеження;
2. дослідити особливості предметної області, визначити ключові фактори, що впливають на інтеграцію веб-сервісів;
3. проаналізувати наявні дослідження та розробки за темою інтеграції веб-сервісів, виявити найкращі практики та підходи;

4. визначити вимоги до інтегрованої платформи комунікації, включно з функціональними та нефункціональними аспектами;
5. розробити концептуальну модель інтегрованої платформи, що описує архітектуру і взаємодію компонентів;
6. спроєктувати та прототипувати інтегровану платформу на основі визначених вимог, провести тестування та оцінку результатів.

1.2.5 Методи дослідження

Для виконання поставлених завдань будуть використані такі методи дослідження:

- аналітичний метод: вивчення та аналіз навчальної літератури, документації та наявних досліджень щодо інтеграції веб-сервісів і використання API;
- порівняльний метод: порівняння можливостей та обмежень різних API, що використовуються в сучасних веб-сервісах;
- метод системного аналізу: визначення вимог і розробка концептуальної моделі інтегрованої платформи, аналіз взаємодії між компонентами системи;
- експериментальний метод: створення прототипу та тестування розробленої інтегрованої платформи, проведення експериментів для оцінювання функціональності та продуктивності;
- метод моделювання: створення та аналіз моделей архітектури та взаємодії компонентів інтегрованої платформи.

Застосування цих методів дасть змогу всебічно і глибоко вивчити предмет, розробити ефективне рішення для інтеграції веб-сервісів і створити прототип інтегрованої комунікаційної платформи, що відповідає сучасним вимогам і стандартам.

РОЗДІЛ 2 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ НАЯВНИХ РІШЕНЬ

2.1 Особливості предметної області

Інтеграція веб-сервісів за допомогою API є складним і багатогранним процесом, який вимагає врахування безлічі факторів і особливостей. Для повнішого розуміння цієї предметної області необхідно розглянути ключові аспекти, що впливають на інтеграцію.

2.1.1 Типи веб-сервісів та їхня функціональність

Веб-сервіси пропонують безліч функціональних можливостей, які можуть бути інтегровані в єдину платформу. Найбільш важливими типами веб-сервісів є:

- соціальні мережі (наприклад, Facebook, Twitter, LinkedIn): пропонують API для управління постами, користувачами, коментарями та аналізу даних;
- месенджери (наприклад, Telegram, WhatsApp, Slack): пропонують API для надсилання та отримання повідомлень, управління чатами та інтеграції ботів;
- системи управління контентом (CMS) (наприклад, WordPress, Joomla): API дають змогу керувати контентом, користувачами та налаштуваннями сайту;
- хмарні сховища (наприклад, Google Drive, Dropbox, OneDrive): API забезпечують доступ до файлів, управління ними та обмін даними;
- електронна комерція (наприклад, Shopify, WooCommerce): API допомагають керувати товарами, замовленнями, клієнтами і транзакціями.

2.1.2 Типи API та їхні особливості

Існують різні типи API, кожен з яких має свої особливості:

- REST API: найпоширеніший тип, який використовує принципи REST і протокол HTTP для взаємодії. REST API прості у використанні та широко підтримуються;
- SOAP API: використовує XML для обміну повідомленнями, дуже надійний і безпечний, але вимагає великих зусиль під час розробки;
- GraphQL: дає змогу клієнту запитувати тільки необхідні дані, що підвищує продуктивність і знижує навантаження на мережу.

2.1.3 Виклики та обмеження інтеграції

Інтеграція веб-сервісів за допомогою API стикається з кількома основними проблемами:

- сумісність: різні веб-служби можуть використовувати різні стандарти і протоколи, що ускладнює інтеграцію;
- безпека: зростаюче число точок доступу і потенційні вразливості вимагають ретельного розгляду питань безпеки та конфіденційності;
- масштабованість: системи мають бути масштабованими, щоб забезпечити стабільну роботу в міру збільшення обсягу даних і кількості користувачів;
- залежність від сторонніх сервісів: зміна умов використання або відмова від сторонніх сервісів може згубно позначитися на інтегрованих системах.

2.1.4 Переваги інтеграції

Інтеграція веб-сервісів через API дає безліч переваг:

- розширена функціональність: дає змогу створювати більш потужні та багатофункціональні додатки;
- підвищення ефективності: покращує організацію робочих процесів і скорочує кількість рутинних операцій;
- централізоване управління даними: дає змогу збирати й обробляти дані з різних джерел у єдиному інтерфейсі;
- підвищення зручності використання: забезпечує більш зручні та інтуїтивно зрозумілі інтерфейси для користувачів.

Специфіка інтеграції веб-сервісів містить у собі безліч чинників, які необхідно враховувати для розробки успішних інтегрованих рішень. Розуміння цих особливостей дає змогу ефективно планувати та реалізовувати інтеграцію, що забезпечує високу продуктивність, безпеку та зручність використання.

2.2 Огляд та аналіз досліджень на тему інтеграції веб-сервісів

2.2.1 Сучасні тенденції та технології

Інтеграція веб-сервісів - це галузь, що швидко розвивається, з кількома важливими тенденціями і технологіями:

- Архітектура мікросервісів:
  - опис: реалізація архітектури мікросервісів, за якої додаток ділиться на невеликі незалежні сервіси, кожен з яких виконує одну функцію;
  - переваги: підвищена гнучкість, спрощене масштабування та можливість незалежного оновлення компонентів;
  - приклад: Netflix використовує архітектуру мікросервісів для забезпечення масштабованості та надійності своєї платформи потокового мовлення.
- Управління API:

- опис: важливість використання платформ управління API, таких як Apigee, AWS API Gateway та інших, для управління і контролю доступу до API;
- переваги: підвищена безпека, моніторинг, контроль версій і аналіз використання API;
- приклад: багато компаній, включно з Google і Amazon, пропонують потужні інструменти управління API, які розробники можуть використовувати для створення, захисту та масштабування API.

- GraphQL:
  - опис: представлення GraphQL як альтернативи REST API, що дозволяє клієнтам запитувати саме ті дані, які їм потрібні;
  - переваги: зниження навантаження на мережу, підвищення продуктивності та більш гнучкий підхід до обробки даних;
  - приклад: компанія Facebook розробила GraphQL і активно використовує його для своїх внутрішніх і публічних API.
- Безсерверні та FaaS (Functions as a Service) сервіси:
  - опис: використання безсерверних архітектур і функцій як сервісу, як-от AWS Lambda або Azure Functions, для виконання коду у відповідь на події без необхідності керування серверами;
  - переваги: спрощене масштабування, зниження операційних витрат і прискорення розробки;
  - приклад: стартапи та великі підприємства використовують функції на стороні сервера для виконання тригерних завдань, як-от обробка даних або надсилання повідомлень.

2.2.2 Переваги та обмеження наявних рішень

Дослідження та практичний досвід показують, що інтеграція веб-сервісів має як переваги, так і виклики.

Переваги існуючих рішень:

- Підвищення продуктивності:
    - інтеграція дозволяє автоматизувати багато процесів, скоротити час і трудовитрати;
    - приклади: автоматичне оновлення контенту на різних платформах, інтеграція CRM-систем з маркетинговими інструментами.
- Кращий аналіз та звітність:
    - дані, зібрані з різних джерел, можна об'єднувати та аналізувати в режимі реального часу;
    - приклади: інтеграція систем бізнес-аналітики (BI) з ERP і CRM для всебічного аналізу бізнесу.
- Підвищена гнучкість і масштабованість:
    - архітектура мікросервісів та управління API полегшують масштабування систем та адаптацію до змін;
    - приклади: такі компанії, як Uber та Airbnb, використовують мікросервіси для забезпечення масштабованості та надійності своїх платформ.

Обмеження існуючих рішень:

- Складність інтеграції:
    - різні стандарти та протоколи, що використовуються різними веб-сервісами, можуть ускладнювати інтеграцію;
    - приклади: інтеграція застарілих систем із сучасними хмарними сервісами часто вимагає значних зусиль і додаткових інструментів.
- Безпека:
    - інтеграція збільшує кількість точок входу, що може підвищити ризик кібератак і витоку даних;

- приклади: необхідно впроваджувати додаткові заходи безпеки, такі як шифрування даних та контроль доступу.
- Залежність від сторонніх сервісів:
  - використання сторонніх API може призвести до залежності від зовнішніх постачальників, які можуть змінити умови використання або припинити надання своїх послуг;
  - приклади: зміни в політиці API, як це сталося з API Facebook після скандалу з Cambridge Analytica.
- Продуктивність і затримки:
  - інтеграція різних сервісів може призвести до збільшення затримок і погіршення продуктивності через затримку в мережі та складність координації між сервісами;
  - приклади: неефективна обробка запитів до API може призвести до затримок у роботі додатків, особливо за великого навантаження.

Загалом, огляд та аналіз досліджень у сфері інтеграції веб-сервісів показує, що, незважаючи на існуючі труднощі, сучасні технології та підходи відкривають широкі можливості для ефективної інтеграції різних сервісів в єдину платформу. Це дозволяє створювати потужні, гнучкі та масштабовані рішення, які відповідають потребам як організацій, так і кінцевих користувачів.

2.3 Визначення вимог до інтегрованої платформи

Розробка інтегрованої комунікаційної платформи вимагає ретельного розгляду вимог щодо забезпечення її функціональності, безпеки та надійності. У цьому розділі описані функціональні, нефункціональні та безпекові вимоги, які необхідно враховувати при розробці платформи.

2.3.1 Функціональні вимоги

Функціональні вимоги визначають, що платформа повинна вміти робити для виконання своїх основних завдань. У цьому контексті можна виділити такі основні функціональні вимоги:

- Аутентифікація та авторизація користувачів:
  - платформа повинна підтримувати багатофакторну автентифікацію та інтегруватися з різними системами контролю доступу (OAuth, OpenID Connect тощо);
  - повинна бути реалізована гнучка система ролей і прав доступу для управління доступом до різних функцій і даних.
- Управління повідомленнями та комунікаціями:
  - інтегрований обмін повідомленнями та можливість надсилати та отримувати повідомлення через соціальні мережі;
  - підтримка групових дискусій, каналів і приватних повідомлень;
  - інтеграція з електронною поштою для надсилання та отримання листів.
- Інтеграція із зовнішніми сервісами:
  - платформа повинна підтримувати API підключення та управління різними веб-сервісами, такими як CRM, ERP, соціальні мережі, обмін миттєвими повідомленнями, хмарні сховища тощо;
  - автоматична синхронізація даних між різними сервісами.
- Управління контентом:
  - створення, редагування та видалення контенту (наприклад, постів у соціальних мережах, файлів у хмарних сховищах);
  - планування публікацій і підтримка автоматизованої публікації контенту.
- Звітність та аналіз:

    - платформа повинна надавати інструменти для збору та аналізу даних про взаємодію з користувачами, ефективність комунікації та інші показники;
    - візуалізація даних за допомогою графіків, діаграм ,та інформаційних панелей.
- Сповіщення:
    - підтримка кастомізованих сповіщень для різних подій, таких як нові повідомлення або зміни даних;
    - надсилати сповіщення різними каналами (месенджер, електронна пошта, SMS).

2.3.2 Нефункціональні вимоги

Нефункціональні вимоги описують якості та характеристики системи, які є важливими для правильної роботи системи:

- Продуктивність та масштабованість:
    - продуктивність і масштабованість: платформа повинна мати можливість обробляти велику кількість запитів і даних;
    - система повинна мати можливість легко адаптуватися до збільшення кількості користувачів та обсягу даних.
- Надійність та відмовостійкість:
    - платформа повинна бути відмовостійкою і забезпечувати високий рівень доступності (наприклад, 99,9% часу безвідмовної роботи);
    - підтримка механізмів резервного копіювання та відновлення даних.
- Простота використання та доступність:
    - інтуїтивно зрозумілий, зручний інтерфейс робить його простим у використанні та швидким у вивченні;

   - підтримка адаптивного дизайну, який працює на різних пристроях (ПК, планшетах і смартфонах).
- Сумісність та інтеграція:
   - платформа повинна бути сумісною з різними операційними системами та браузерами;
   - підтримувати інтеграцію з поширеними інструментами та сервісами, такими як Jira, Slack, Google Workspace тощо.
- Захист даних:
   - відповідність стандартам і правилам захисту даних (наприклад, GDPR);
   - налаштовувані політики конфіденційності та управління даними користувачів.

2.3.3 Вимоги безпеки

Безпека є важливим аспектом при розробці інтеграційних платформ. Необхідно враховувати наступні вимоги до безпеки:

- Захист даних:
   - захист даних: шифрування даних у стані спокою та під час передачі з використанням сучасних алгоритмів (наприклад, AES-256, TLS);
   - регулярне оновлення та управління ключами шифрування.
- Контроль доступу:
   - реалізація багатофакторної аутентифікації для всіх користувачів;
   - гнучке управління ролями та правами доступу для обмеження доступу до різних функцій і даних.
- Моніторинг та аудит:
   - безперервний моніторинг активності користувачів і системних подій для виявлення підозрілих дій;

  - журнали аудиту зберігаються для аналізу та запитів.
- Захист від зовнішніх загроз:
  - системи виявлення та запобігання вторгнень (IDS/IPS);
  - регулярне тестування на проникнення та сканування вразливостей.
- План реагування на інциденти:
  - розробка та впровадження плану реагування на інциденти безпеки, включаючи процедури сповіщення, розслідування та вирішення інцидентів.
- Навчання та інформування
  - регулярне навчання користувачів і персонал з питань безпеки та підвищення обізнаності про сучасні загрози.

Отже, для успішної побудови інтегрованої комунікаційної платформи необхідно враховувати різні функціональні, нефункціональні та безпекові вимоги. Це гарантує якісну, надійну та безпечну систему, яка відповідає потребам користувачів і найсучаснішим стандартам.

РОЗДІЛ 3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПЛАТФОРМИ

3.1 Розробка функціональної схеми

На рисунку 3.1 показано контекстну діаграму IDEF0.

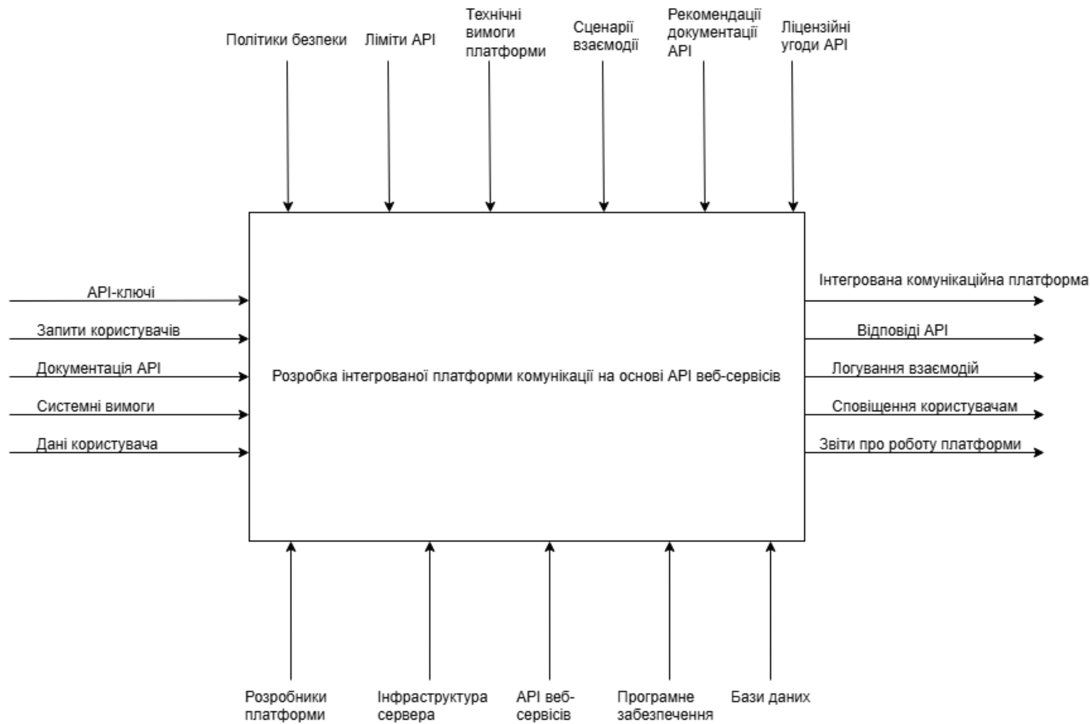

Рисунок 3.1 – Контекстна діаграма IDEF0

Діаграма IDEF0 A-0 дозволяє визначити ключові завдання та дії, необхідні для виконання основної функції або досягнення поставленої мети в рамках певної системи.

3.2 Розробка структурної схеми

На рисунку 3.2 представлена структурна діаграма, що демонструє, які частини слід реалізувати у вигляді підпрограм, а також містить деталізований опис алгоритмів цих підпрограм.

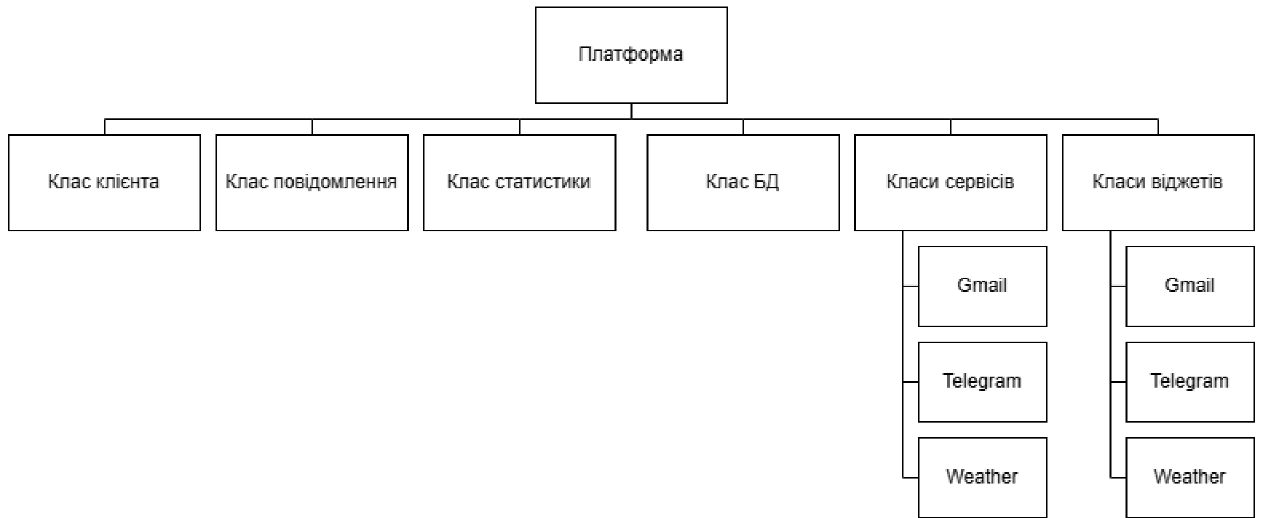

Рисунок 3.2 – Структурна діаграма

Ця діаграма дозволяє чітко зрозуміти, який функціонал реалізовано в кожному з класів, без необхідності занурюватися в його складну деталізацію.

3.3 Розробка UML–діаграми структури бази даних

Схема даних у базі даних зображено на рис. 3.3.

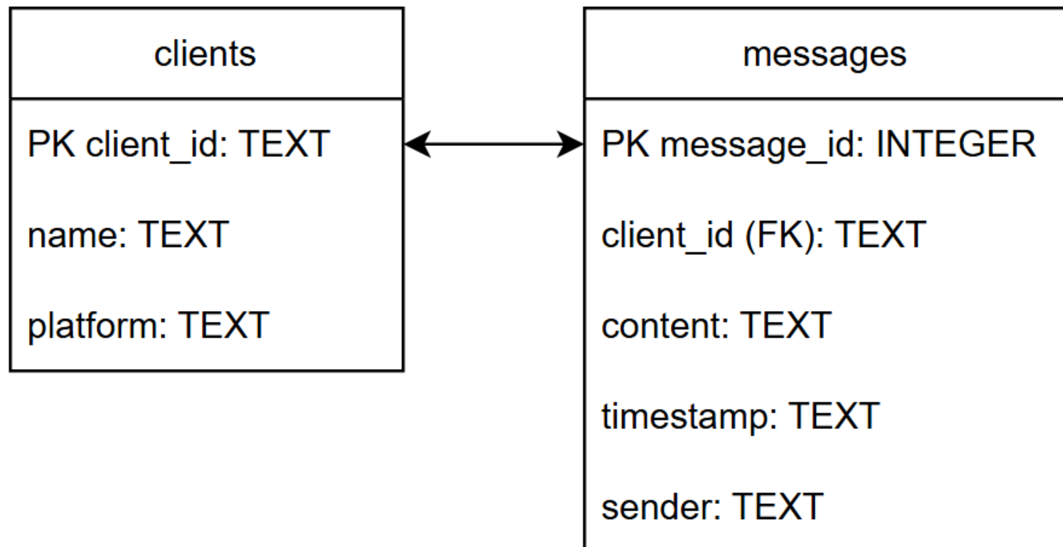

Рисунок 3.3 – UML-діаграма бази даних

Ця діаграма показує структуру бази даних, що складається з двох таблиць: clients (клієнти) та messages (повідомлення). Таблиця clients зберігає інформацію про клієнтів, а таблиця messages містить повідомлення, пов'язані з клієнтами через зовнішній ключ client_id.

3.4 Розробка діаграми варіантів використання

На рисунку 3.4 показано діаграму варіантів використання.

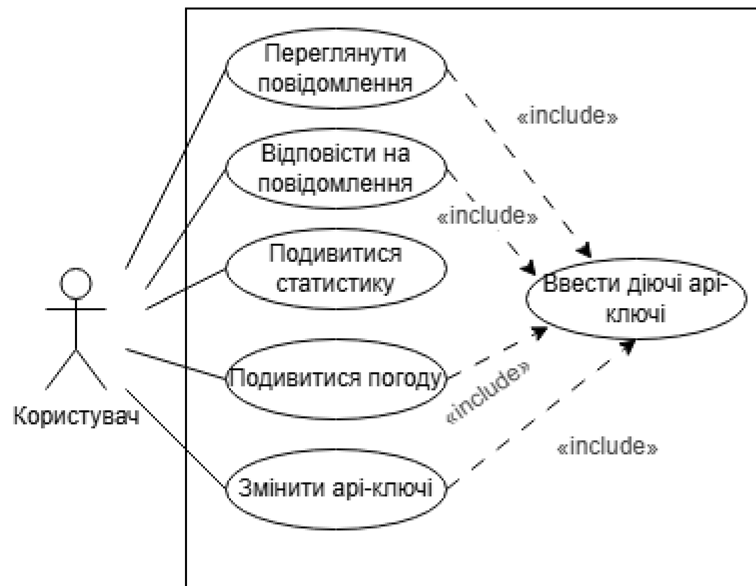

Рисунок 3.4 – Діаграма варіантів використання

Ця діаграма відображає основні сценарії використання системи з точки зору користувача. Діаграма варіантів використання дозволяє описати, які дії можуть виконувати користувачі та як вони взаємодіють з платформою.

3.5 Розробка структур таблиць бази даних та вибір типів даних полів

Всього у базі даних мають бути присутніми 2 таблиці. Нижче наведено опис структури бази даних для програми:

- таблиця "Клієнти" зберігає інформацію про клієнтів і містить три поля:

- "client_id" (первинний ключ, тип даних — текст): унікальний ідентифікатор клієнта;
- "name" (тип даних — текст): ім'я клієнта;
- "platform" (тип даних — текст): платформа, з якою працює клієнт.

- таблиця "Повідомлення" зберігає дані про повідомлення та включає п'ять полів:
  - "message_id" (первинний ключ, тип даних — ціле число): унікальний ідентифікатор повідомлення;
  - "client_id" (тип даних — текст, зовнішній ключ): ідентифікатор клієнта, на якого посилається повідомлення;
  - "content" (тип даних — текст): вміст повідомлення;
  - "timestamp" (тип даних — текст): час відправки повідомлення;
  - "sender" (тип даних — текст): відправник повідомлення (наприклад, ім'я чи роль користувача).

Ці таблиці пов'язані між собою через зовнішній ключ client_id в таблиці "Повідомлення", що вказує на відповідний "client_id" в таблиці "Клієнти".

3.6 Розробка моделі візуального інтерфейсу

При відкритті програми користувач бачить головне вікно, яке організоване наступним чином:

Основна структура головного вікна:

- меню навігації:
  - у верхній частині вікна розташоване головне меню з пунктами:
    - "Файл" — для виконання системних операцій (наприклад, вихід з програми чи відкриття нового вікна);
    - "Справка" — для отримання інформації про програму чи доступ до інструкцій.
- панель з чатами:

  - ліворуч розміщена панель із переліком доступних чатів. Користувач може вибрати чат із цього списку;
  - у списку чатів відображаються назви чатів та платформи, через які вони додані (наприклад, Telegram чи Gmail).
- область обміну повідомленнями:
  - у центральній частині вікна знаходиться велике поле, де відображаються повідомлення обраного чату. Якщо чат не вибрано, відображається повідомлення "Виберіть чат".
- панель введення повідомлень:
  - у нижній частині розташоване текстове поле для введення тексту повідомлення;
  - праворуч від текстового поля знаходиться кнопка " Надіслати", яка дозволяє надіслати введене повідомлення.
- швидкі відповіді:
  - під текстовим полем знаходиться панель із запрограмованими кнопками для швидких відповідей, таких як:
    - " Здрастуйте! Як можу вам допомогти?";
    - " Дякую за звернення! Ми зв'яжемося з вами найближчим часом.";
    - Та інші повідомлення для швидкого реагування.

Інтерактивність:

- вибір чату: користувач може обрати чат із лівої панелі, що оновлює вміст центрального поля обміну повідомленнями;
- відправка повідомлення: користувач вводить текст у відповідне поле та натискає кнопку "Надіслати", після чого текст відображається в обраному чаті;
- використання швидких відповідей: натискання на одну з кнопок швидких відповідей автоматично вставляє текст у поле введення повідомлень.

Додаткові можливості:

- програма може оновлювати список чатів у реальному часі, якщо додаються нові;
- в разі помилки з'єднання із сервером або неможливості доступу до чатів, користувач отримує відповідне повідомлення, і функціональність програми обмежується.

На рисунку 3.5 зображено центральний хаб для роботи користувача з чатами, забезпечуючи простий і зрозумілий інтерфейс.

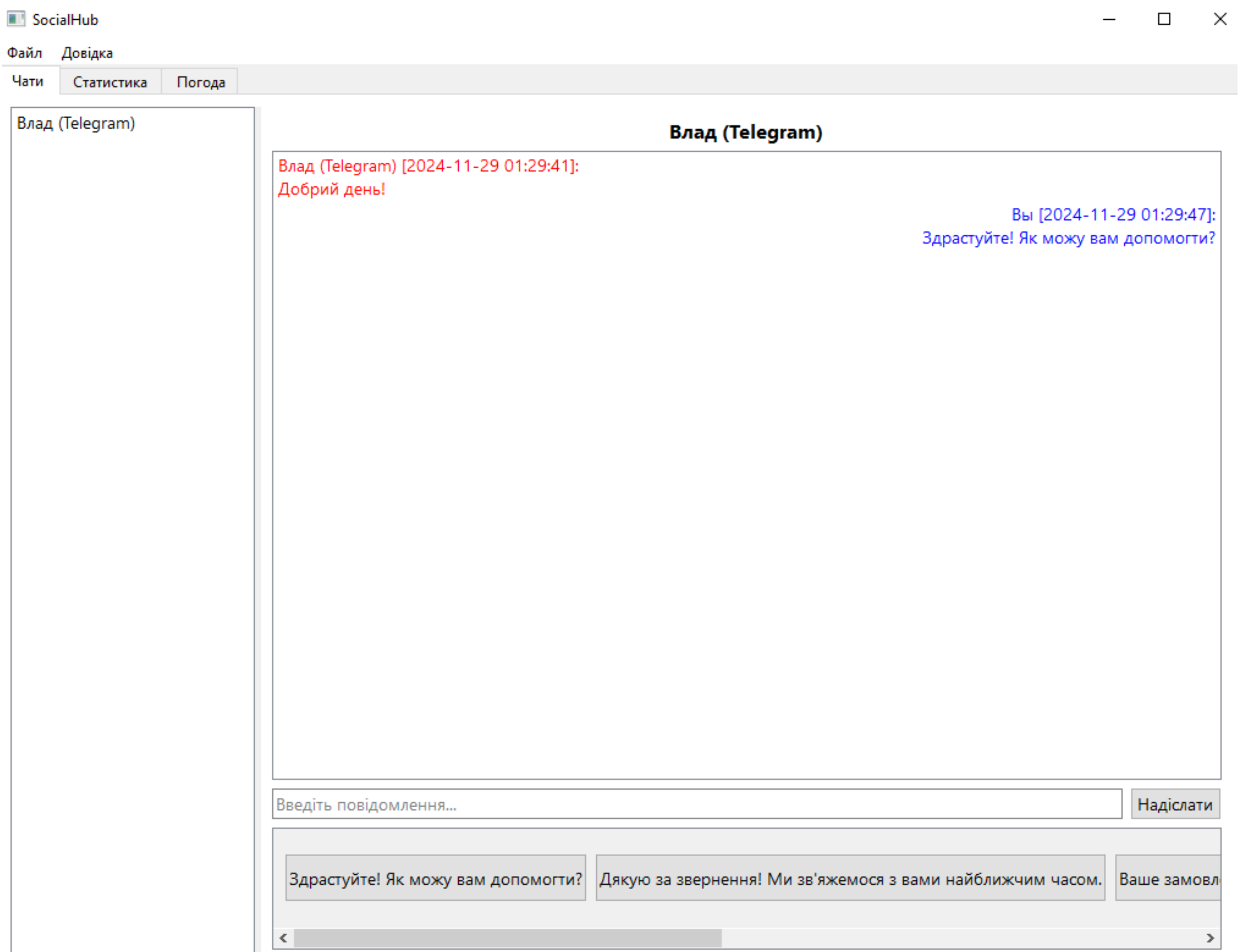

Рисунок 3.5 – Вигляд головної форми

3.7 Розробка UML-діаграм класів платформи

На рисунках 3.6 - 3.18 зображено UML-дірграми класів програмного комплексу.

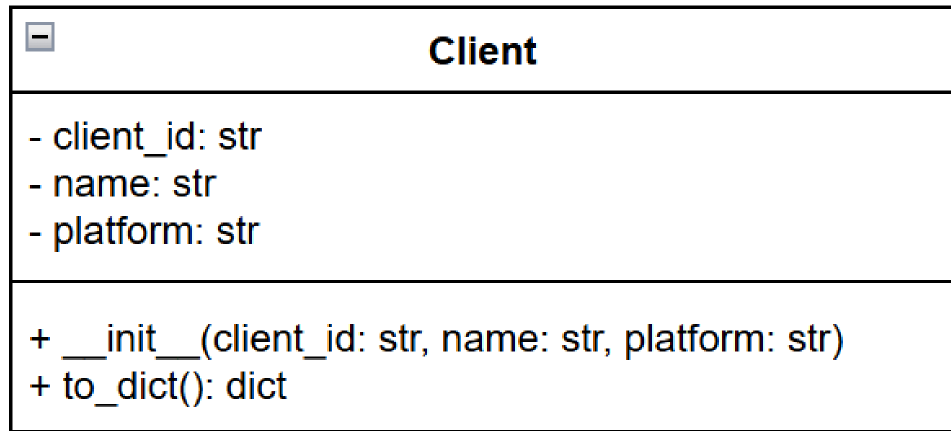

Рисунок 3.6 – UML-діаграма класу Client

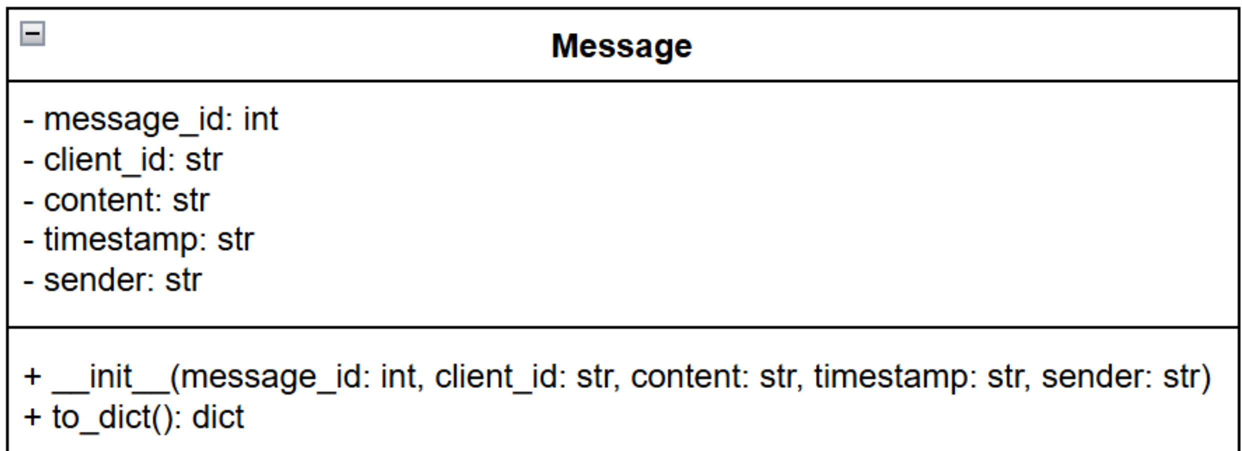

Рисунок 3.7 – UML-діаграма класу Message

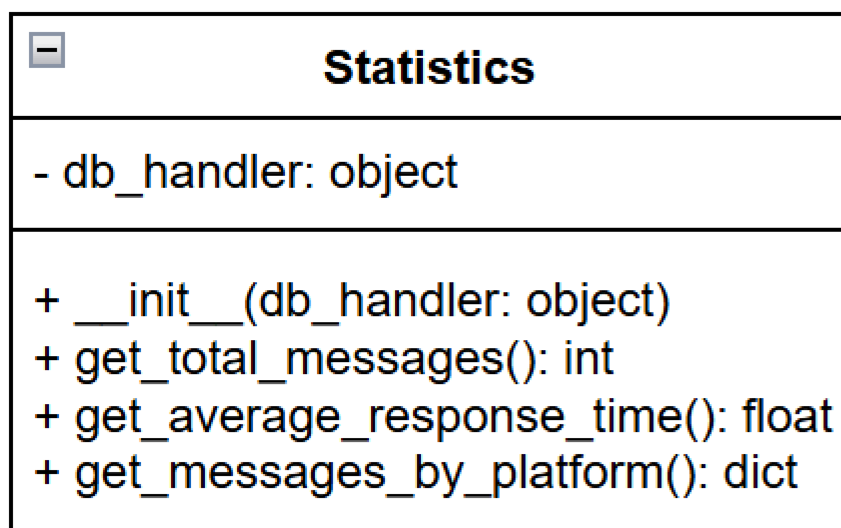

Рисунок 3.8 – UML-діаграма класу Statistics

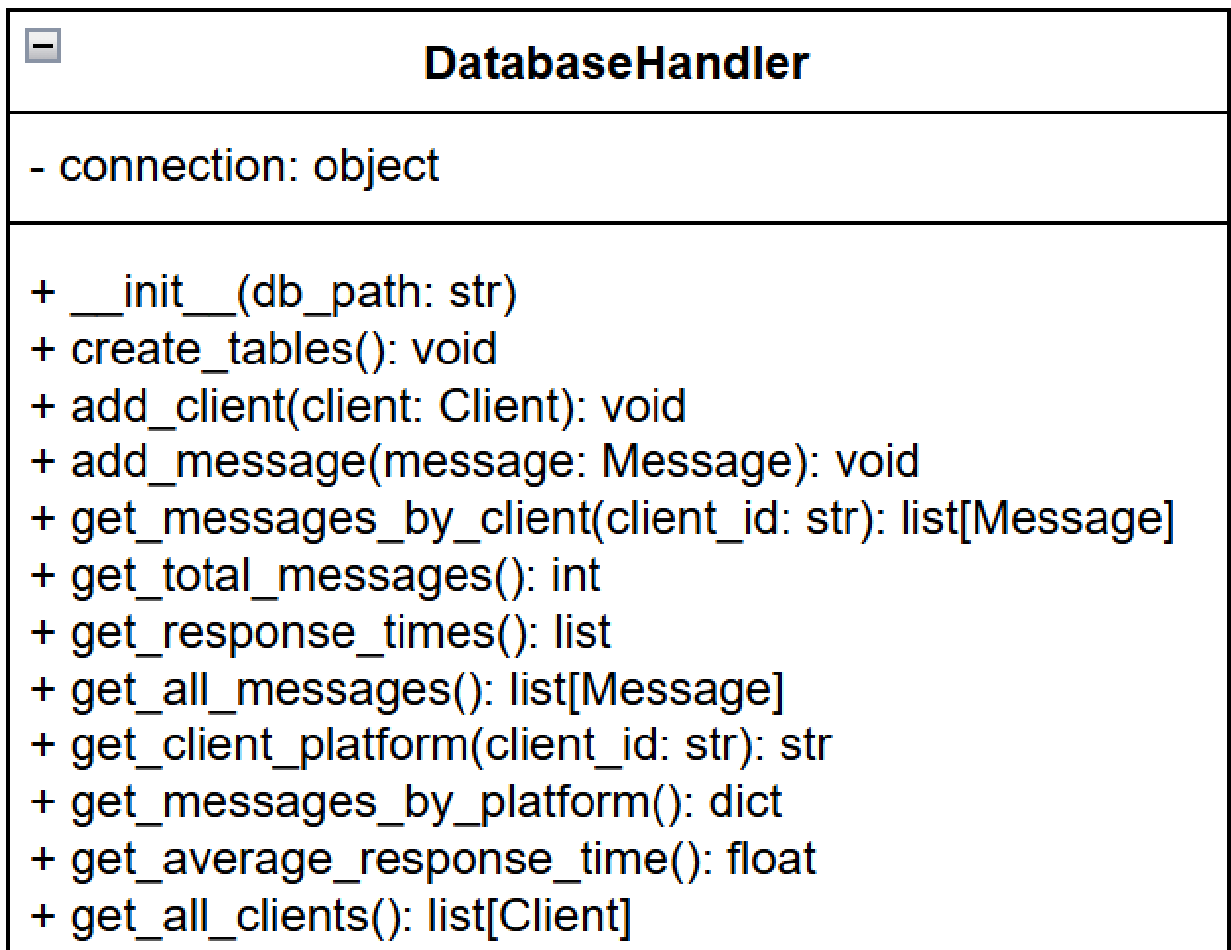

Рисунок 3.9 – UML-діаграма класу DatabaseHandler

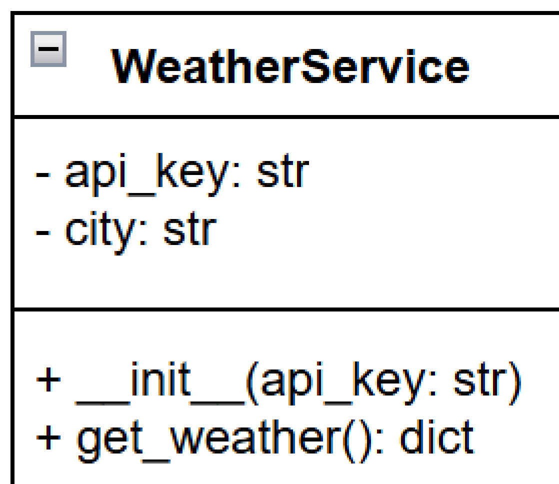

Рисунок 3.10 – UML-діаграма класу WeatherService

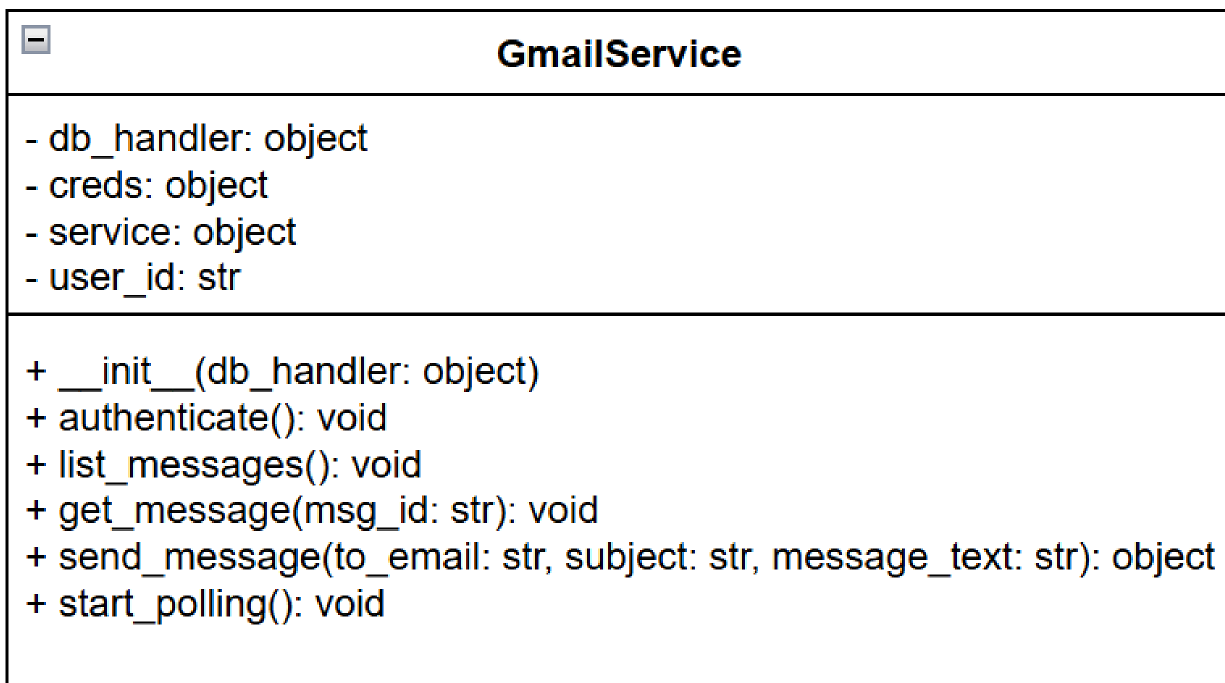

Рисунок 3.11 – UML-діаграма класу GmailService

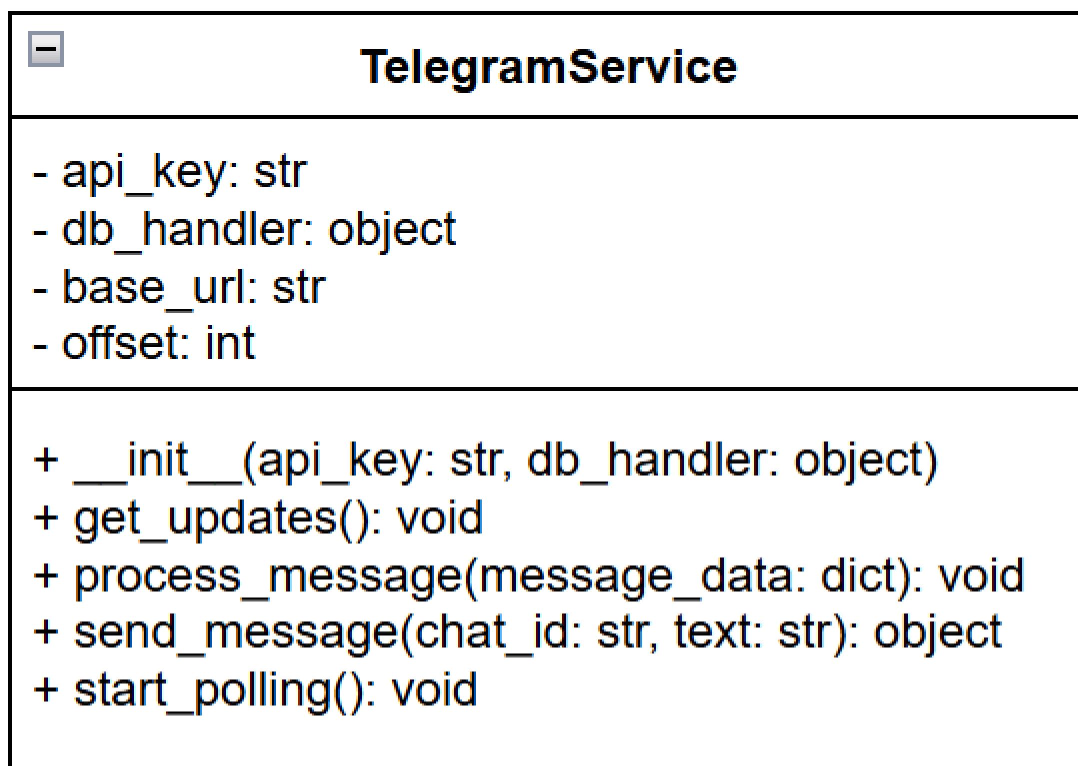

Рисунок 3.12 – UML-діаграма класу TelegramService

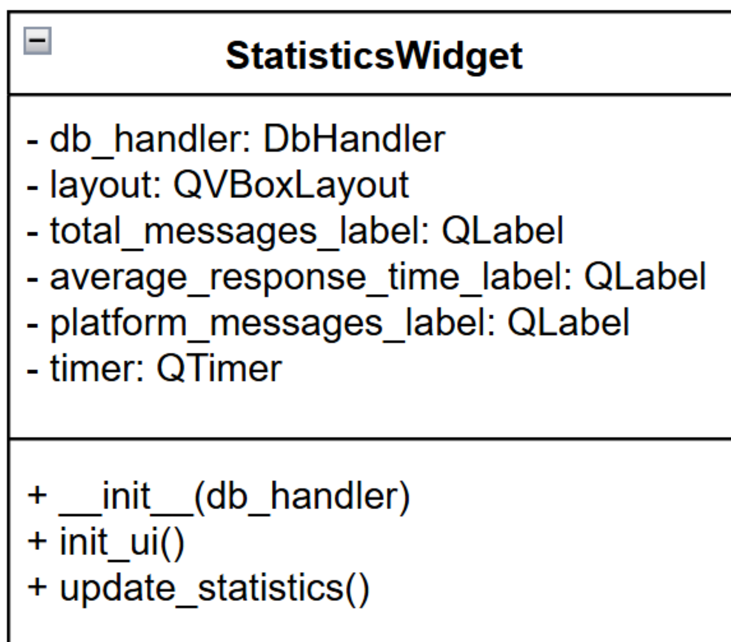

Рисунок 3.13 – UML-діаграма класу StatisticsWidget

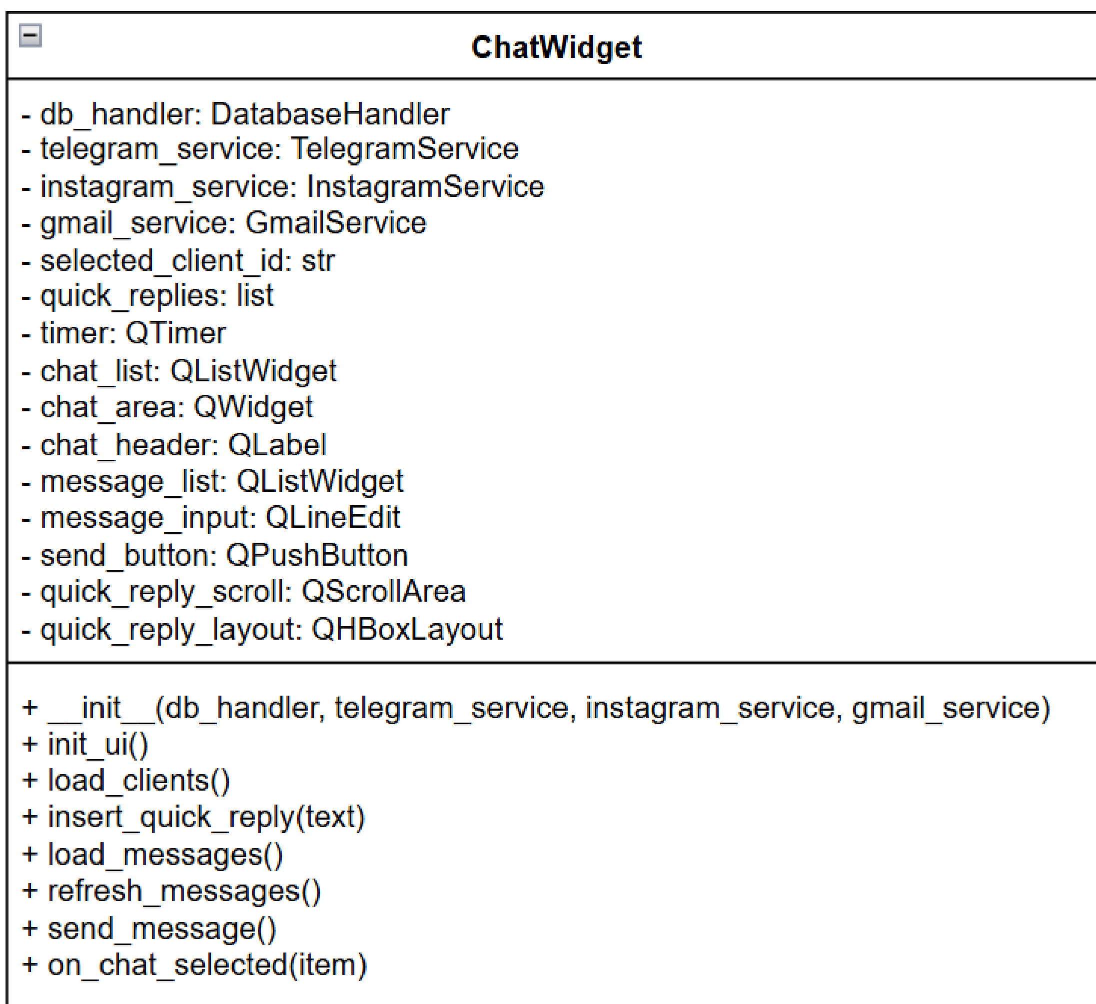

Рисунок 3.14 – UML-діаграма класу ChatWidget

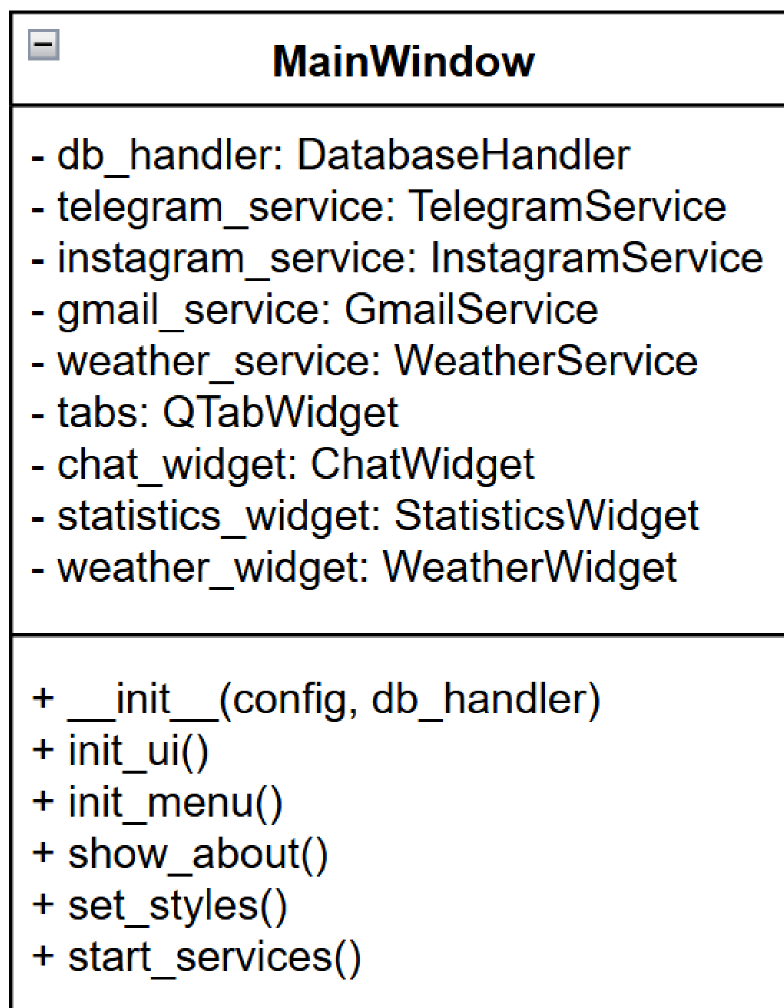

Рисунок 3.15 – UML-діаграма класу MainWindow

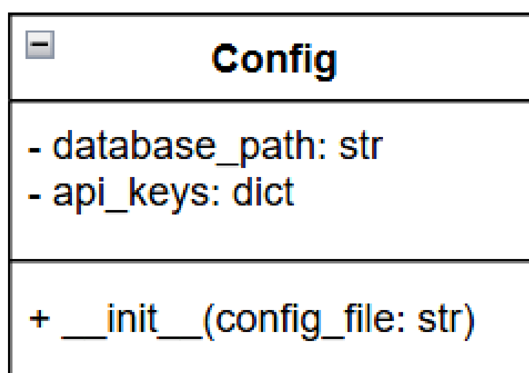

Рисунок 3.16 – UML-діаграма класу Config

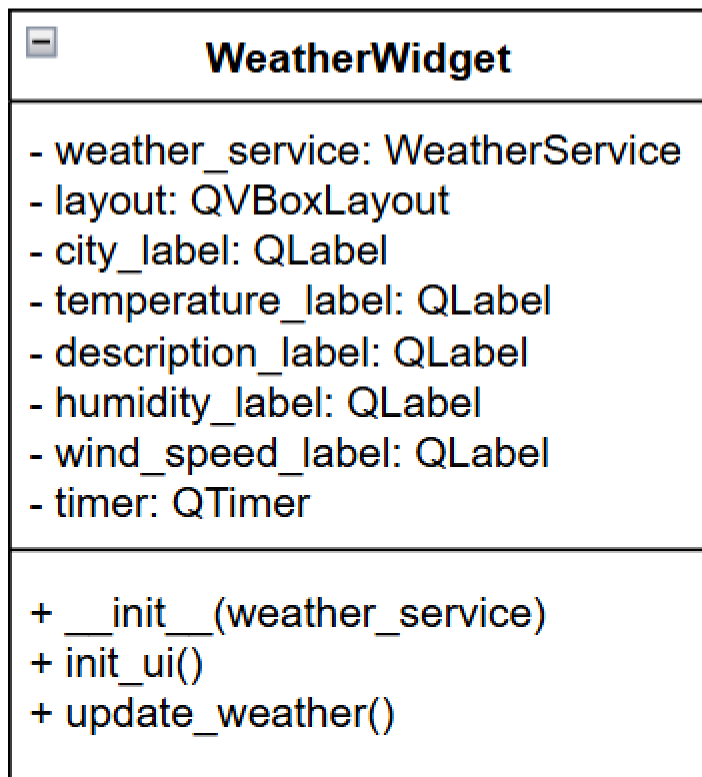

Рисунок 3.17 – UML-діаграма класу WeatherWidget

Ці діаграми допомагають ілюструвати класи, їх атрибути та методи, спрощуючи розуміння компонентів системи та їх взаємодії.

РОЗДІЛ 4 РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ

4.1 Розробка основних класів проекту

Клас Client містить такі властивості: client_id, name, platform. Та методи: __init__, to_dict. Цей клас відповідає за зберігання інформації про клієнта.

Детальний опис властивостей:

- client_id – ця властивість представляє унікальний ідентифікатор клієнта. Використовується для його однозначної ідентифікації. Тип даних: int або інший тип (залежно від використання);
- name – ця властивість зберігає ім'я клієнта. Тип даних: str;
- platform – ця властивість вказує на платформу, яку використовує клієнт (наприклад, Telegram, Gmail тощо). Тип даних: str.

Детальний опис методів:

- __init__(self, client_id, name, platform) – конструктор класу, який ініціалізує об'єкт Client з переданими значеннями для client_id, name та platform. Приймає три аргументи: client_id, name, platform. Нічого не повертає;
- to_dict(self) – конвертує об'єкт Client у словник (формат dict), де ключами є назви властивостей (client_id, name, platform), а значеннями – їх відповідні значення. Нічого не приймає. Повертає об'єкт типу dict.

Клас Message містить такі властивості: message_id, client_id, content, timestamp, sender. Та методи: __init__, to_dict. Цей клас відповідає за представлення повідомлення, яке надсилається клієнтом.

Детальний опис властивостей:

- message_id – ця властивість представляє унікальний ідентифікатор повідомлення. Використовується для його однозначної ідентифікації. Тип даних: int або інший тип (залежно від використання);

- client_id – ця властивість містить ідентифікатор клієнта, який надіслав повідомлення. Тип даних: int або інший тип (залежно від використання);
- content – ця властивість зберігає текст або вміст повідомлення. Тип даних: str;
- timestamp – ця властивість вказує час створення або відправлення повідомлення. Тип даних: str або datetime (залежно від реалізації);
- sender – ця властивість представляє відправника повідомлення (наприклад, ім'я або роль). Тип даних: str.

Детальний опис методів:

- __init__(self, message_id, client_id, content, timestamp, sender) – конструктор класу, який ініціалізує об'єкт Message з переданими значеннями для message_id, client_id, content, timestamp та sender. Приймає п'ять аргументів: message_id, client_id, content, timestamp, sender. Нічого не повертає;
- to_dict(self) – конвертує об'єкт Message у словник (формат dict), де ключами є назви властивостей (message_id, client_id, content, timestamp, sender), а значеннями – їх відповідні значення. Нічого не приймає. Повертає об'єкт типу dict.

Клас Statistics містить такі властивості: db_handler. Та методи: __init__, get_total_messages, get_average_response_time, get_messages_by_platform. Цей клас відповідає за обробку та отримання статистики із бази даних.

Детальний опис властивостей:

- db_handler – ця властивість представляє об'єкт, який відповідає за взаємодію з базою даних. Використовується для виконання запитів та отримання статистичних даних. Тип даних: об'єкт класу, що реалізує методи роботи з базою даних (наприклад, DBHandler).

Детальний опис методів:

- __init__(self, db_handler) – конструктор класу, який ініціалізує об'єкт Statistics з переданим об'єктом db_handler. Приймає один аргумент: db_handler. Нічого не повертає;
- get_total_messages(self) – отримує загальну кількість повідомлень із бази даних. Викликає відповідний метод get_total_messages у db_handler. Нічого не приймає. Повертає кількість повідомлень (int);
- get_average_response_time(self) – отримує середній час відповіді із бази даних. Викликає відповідний метод get_average_response_time у db_handler. Нічого не приймає. Повертає середній час відповіді;
- get_messages_by_platform(self) – отримує статистику кількості повідомлень для кожної платформи із бази даних. Викликає відповідний метод get_messages_by_platform у db_handler. Нічого не приймає. Повертає словник (dict), де ключами є назви платформ, а значеннями – кількість повідомлень.

Клас DatabaseHandler містить такі властивості: connection. Та методи: __init__, create_tables, add_client, add_message, get_messages_by_client, get_total_messages, get_response_times, get_all_messages, get_client_platform, get_messages_by_platform, get_average_response_time, get_all_clients. Цей клас відповідає за взаємодію з базою даних, зокрема за збереження, отримання та обробку даних клієнтів і повідомлень.

Детальний опис властивостей:

- connection – ця властивість представляє з'єднання з базою даних SQLite. Вона використовується для виконання SQL-запитів до бази даних. Тип даних: sqlite3.Connection.

Детальний опис методів:

- __init__(self, db_path) – конструктор класу, який відкриває з'єднання з базою даних за вказаним шляхом (db_path) і створює необхідні таблиці, викликаючи метод create_tables. Приймає один аргумент: db_path (шлях до файлу бази даних). Нічого не повертає;

- create_tables(self) – створює таблиці clients і messages, якщо вони ще не існують, з використанням SQL-запитів. Нічого не приймає і не повертає;
- add_client(self, client: Client) – додає нового клієнта до таблиці clients. Якщо клієнт з таким client_id вже існує, запит ігнорується. Приймає один аргумент: об'єкт типу Client. Нічого не повертає;
- add_message(self, message: Message) – додає нове повідомлення до таблиці messages. Приймає один аргумент: об'єкт типу Message. Нічого не повертає;
- get_messages_by_client(self, client_id) – отримує всі повідомлення, надіслані клієнтом із вказаним client_id. Повертає список об'єктів Message;
- get_total_messages(self) – отримує загальну кількість повідомлень у таблиці messages. Нічого не приймає. Повертає кількість повідомлень (int);
- get_response_times(self) – отримує список часів відповідей (логіка ще не реалізована). Нічого не приймає. Повертає список (list) або пустий список як заглушку;
- get_all_messages(self) – отримує всі повідомлення з таблиці messages. Нічого не приймає. Повертає список об'єктів Message;
- get_client_platform(self, client_id) – отримує платформу клієнта за його client_id. Приймає один аргумент: client_id. Повертає назву платформи (str) або None, якщо клієнт не знайдений;
- get_messages_by_platform(self) – отримує статистику кількості повідомлень для кожної платформи. Використовує об'єднання таблиць clients і messages. Нічого не приймає. Повертає словник (dict), де ключами є назви платформ, а значеннями – кількість повідомлень;

- get_average_response_time(self) – отримує середній час відповіді. Нічого не приймає. Повертає час;
- get_all_clients(self) – отримує всіх клієнтів із таблиці clients. Нічого не приймає. Повертає список об'єктів Client.

Клас GmailService містить такі властивості: db_handler, creds, service, user_id. Та методи: __init__, authenticate, list_messages, get_message, send_message, start_polling. Цей клас відповідає за інтеграцію з Gmail API, обробку вхідних повідомлень та їх збереження у базі даних, а також за відправку повідомлень через Gmail.

Детальний опис властивостей:

- db_handler – ця властивість представляє об'єкт для роботи з базою даних. Використовується для додавання клієнтів і повідомлень у базу. Тип даних: об'єкт класу DatabaseHandler;
- creds – містить авторизаційні дані для доступу до Gmail API. Тип даних: google.oauth2.credentials.Credentials;
- service – використовується для взаємодії з Gmail API після авторизації. Тип даних: сервісний об'єкт, створений за допомогою googleapiclient.discovery.build;
- user_id – ідентифікатор користувача Gmail API. Значення me використовується для посилання на авторизованого користувача. Тип даних: str.

Детальний опис методів:

- __init__(self, db_handler) – конструктор класу, який ініціалізує об'єкт GmailService з переданим об'єктом db_handler. Викликає метод authenticate для авторизації. Приймає один аргумент: db_handler. Нічого не повертає;
- authenticate(self) – відповідає за авторизацію користувача через Gmail API. Якщо файл token.json існує, використовує його для отримання токена доступу. Якщо токен недійсний або відсутній, запускає процес

авторизації через браузер. Після авторизації зберігає токен у файл token.json. Нічого не приймає і не повертає;

- list_messages(self) – отримує список непрочитаних повідомлень у папці INBOX за допомогою Gmail API, викликаючи метод get_message для кожного з них. Нічого не приймає і не повертає;
- get_message(self, msg_id) – отримує повну інформацію про повідомлення за його msg_id і обробляє його. Зчитує відправника, тему, текст повідомлення, додає клієнта до бази даних і зберігає повідомлення у базу. Після обробки повідомлення позначає його як прочитане. Приймає один аргумент: msg_id (ідентифікатор повідомлення). Нічого не повертає;
- send_message(self, to_email, subject, message_text) – відправляє повідомлення на вказану електронну адресу через Gmail API. Приймає три аргументи: to_email (адреса отримувача), subject (тема повідомлення) та message_text (текст повідомлення). Повертає відповідь API про відправлене повідомлення або друкує помилку у разі невдачі;
- start_polling(self) – запускає процес опитування вхідних повідомлень у фоновому потоці. Використовує метод list_messages для перевірки нових повідомлень кожні 10 секунд. Нічого не приймає і не повертає.

Клас TelegramService містить такі властивості: api_key, db_handler, base_url, offset. Та методи: __init__, get_updates, process_message, send_message, start_polling. Цей клас відповідає за інтеграцію з Telegram Bot API, обробку вхідних повідомлень, їх збереження у базі даних, а також за надсилання повідомлень через Telegram.

Детальний опис властивостей:

- api_key – містить API-ключ для автентифікації бота у Telegram Bot API. Використовується для побудови запитів до Telegram API. Тип даних: str;

- db_handler – представляє об'єкт для роботи з базою даних. Використовується для збереження інформації про клієнтів і повідомлення. Тип даних: об'єкт класу DatabaseHandler;
- base_url – базова URL-адреса для запитів до Telegram Bot API. Формується на основі api_key. Тип даних: str;
- offset – зберігає зсув для отримання нових оновлень (update_id). Використовується для уникнення обробки однакових повідомлень кілька разів. Тип даних: int або None.

Детальний опис методів:

- __init__(self, api_key, db_handler) – конструктор класу, який ініціалізує об'єкт TelegramService. Приймає два аргументи: api_key (API-ключ Telegram бота) і db_handler (об'єкт для роботи з базою даних). Нічого не повертає;
- get_updates(self) – отримує нові оновлення (повідомлення) від Telegram Bot API за допомогою методу getUpdates. Обробляє кожне повідомлення, викликаючи метод process_message, і оновлює значення offset. Нічого не приймає і не повертає;
- process_message(self, message_data) – обробляє отримане повідомлення. Зчитує ID чату, ім'я користувача, текст повідомлення та час надсилання. Зберігає інформацію про клієнта та повідомлення у базу даних через db_handler. Приймає один аргумент: message_data (дані повідомлення у форматі словника). Нічого не повертає;
- send_message(self, chat_id, text) – надсилає повідомлення користувачу через Telegram Bot API за допомогою методу sendMessage. Приймає два аргументи: chat_id (ідентифікатор чату, куди надсилається повідомлення) та text (текст повідомлення). Повертає відповідь API у форматі JSON або друкує повідомлення про помилку у разі невдачі;
- start_polling(self) – запускає процес опитування Telegram Bot API для отримання нових повідомлень у фоновому потоці. Викликає get_updates у циклі кожну секунду. Нічого не приймає і не повертає.

Клас WeatherService містить такі властивості: api_key, city. Та методи: __init__, get_weather. Цей клас відповідає за отримання погодних даних з сервісу OpenWeatherMap.

Детальний опис властивостей:

- api_key - ця властивість зберігає ключ API, необхідний для доступу до сервісу OpenWeatherMap. Вона представлена типом даних str і використовується для аутентифікації запитів до API;
- city - ця властивість зберігає назву міста, для якого потрібні погодні дані. Вона представлена типом даних str і за замовчуванням встановлена на "Kryvyi Rih";

Детальний опис методів:

- __init__(self, api_key) – конструктор класу, ініціалізує властивості api_key та city. Приймає один аргумент: api_key. Нічого не повертає;
- get_weather(self) – виконує запит до API OpenWeatherMap для отримання погодних даних. Формує URL та параметри запиту, включаючи назву міста, ключ API, одиниці вимірювання (Цельсій) та мову. Якщо запит успішний і код відповіді дорівнює 200, повертає словник з інформацією про місто, температуру, опис погоди, вологість та швидкість вітру. Якщо виникає помилка або код відповіді не 200, повертає None. Приймає лише параметр self. Повертає словник з погодними даними або None.

Клас ChatWidget містить такі властивості: quick_replies, db_handler, telegram_service, instagram_service, gmail_service, selected_client_id, timer, chat_list, chat_area, chat_header, message_list, message_input, send_button, quick_reply_scroll, quick_reply_layout. Та методи: __init__, init_ui, load_clients, insert_quick_reply, load_messages, refresh_messages, send_message, on_chat_selected. Цей клас відповідає за інтерфейс чату, управління клієнтами та повідомленнями, а також за інтеграцію з різними службами комунікації.

Детальний опис властивостей:

- quick_replies - список швидких відповідей, представлений типом даних list. Використовується для надання користувачеві попередньо визначених відповідей, які можна швидко вставити в поле вводу повідомлення;
- db_handler - обробник бази даних, представлений об'єктом класу, який відповідає за доступ до даних клієнтів та повідомлень. Використовується для отримання та збереження інформації про клієнтів і їхні повідомлення;
- telegram_service - сервіс Telegram, представлений об'єктом відповідного класу. Використовується для надсилання повідомлень клієнтам через платформу Telegram;
- instagram_service - сервіс Instagram, представлений об'єктом відповідного класу. Використовується для надсилання повідомлень клієнтам через платформу Instagram;
- gmail_service - сервіс Gmail, представлений об'єктом відповідного класу. Використовується для надсилання електронних листів клієнтам через сервіс Gmail;
- selected_client_id - ідентифікатор вибраного клієнта, представлений типом даних None або int. Зберігає ID клієнта, з яким наразі ведеться чат;
- timer - таймер для оновлення повідомлень, представлений об'єктом QTimer. Використовується для періодичного оновлення списку повідомлень та клієнтів;
- chat_list - список чатів (клієнтів), представлений об'єктом QListWidget. Відображає список всіх клієнтів для вибору чату;
- chat_area - область чату, представлена об'єктом QWidget. Містить елементи інтерфейсу для відображення та взаємодії з повідомленнями одного клієнта;

- chat_header - заголовок чату, представлений об'єктом QLabel. Відображає назву вибраного клієнта або повідомлення заклику до дії;
- message_list - список повідомлень, представлений об'єктом QListWidget. Відображає історію обміну повідомленнями між користувачем та клієнтом;
- message_input - поле вводу повідомлення, представлений об'єктом QLineEdit. Дозволяє користувачу вводити текст повідомлення для надсилання клієнту;
- send_button - кнопка надсилання повідомлення, представлена об'єктом QPushButton. Викликає функцію send_message для відправки введеного повідомлення;
- quick_reply_scroll - область прокрутки для швидких відповідей, представлена об'єктом QScrollArea. Містить кнопки зі швидкими відповідями для швидкого вставлення тексту в поле вводу;
- quick_reply_layout - макет для розміщення кнопок швидких відповідей, представлений об'єктом QHBoxLayout. Організовує кнопки швидких відповідей у горизонтальний ряд.

Детальний опис методів:

- __init__(self, db_handler, telegram_service, instagram_service, gmail_service) – конструктор класу, ініціалізує властивості quick_replies, db_handler, telegram_service, instagram_service, gmail_service, selected_client_id, та налаштовує інтерфейс користувача через метод init_ui. Також завантажує список клієнтів за допомогою load_clients та налаштовує таймер для оновлення повідомлень кожні 2 секунди. Приймає чотири аргументи: db_handler, telegram_service, instagram_service, gmail_service. Нічого не повертає;
- init_ui(self) – ініціалізує інтерфейс користувача, створює основні елементи макету, такі як список чатів, область чату, заголовок, список повідомлень, поле вводу та кнопки. Налаштовує макети та зв'язки між

елементами інтерфейсу. Приймає лише параметр self. Нічого не повертає;

- load_clients(self) – завантажує список клієнтів з бази даних за допомогою db_handler.get_all_clients та відображає їх у chat_list. Для кожного клієнта створюється елемент списку з відображенням імені або ID клієнта та платформи. Приймає лише параметр self. Нічого не повертає;
- insert_quick_reply(self, text) – вставляє текст швидкої відповіді у поле вводу повідомлення message_input. Приймає один аргумент: text. Нічого не повертає;
- load_messages(self) – завантажує повідомлення для вибраного клієнта з бази даних за допомогою db_handler.get_messages_by_client та відображає їх у message_list. Форматує час повідомлень, визначає відправника та встановлює відповідні вирівнювання та кольори. Приймає лише параметр self. Нічого не повертає;
- refresh_messages(self) – оновлює список клієнтів та, якщо вибраний клієнт, оновлює також список повідомлень, викликаючи методи load_clients та load_messages. Приймає лише параметр self. Нічого не повертає;
- send_message(self) – надсилає повідомлення клієнту через відповідний сервіс (telegram_service, instagram_service або gmail_service) залежно від платформи клієнта. Зберігає повідомлення у базі даних за допомогою db_handler.add_message та оновлює список повідомлень. Приймає лише параметр self. Нічого не повертає;
- on_chat_selected(self, item) – обробляє подію вибору клієнта з списку чатів. Встановлює ідентифікатор вибраного клієнта, оновлює заголовок чату та завантажує відповідні повідомлення, викликаючи методи load_messages. Приймає один аргумент: item. Нічого не повертає.

Клас MainWindow містить такі властивості: db_handler, telegram_service, instagram_service, gmail_service, weather_service, tabs, chat_widget, statistics_widget, weather_widget. Та методи: __init__, init_ui, init_menu, show_about, set_styles, start_services. Цей клас відповідає за основне вікно програми, керування вкладками та інтеграцію з різними сервісами.

Детальний опис властивостей:

- db_handler - обробник бази даних, представлений об'єктом, який відповідає за доступ до даних програми. Використовується для взаємодії з базою даних, зберігання та отримання інформації про користувачів, чати та інші дані;
- telegram_service - сервіс Telegram, представлений об'єктом класу TelegramService. Використовується для надсилання та отримання повідомлень через платформу Telegram;
- gmail_service - сервіс Gmail, представлений об'єктом класу GmailService. Використовується для надсилання електронних листів через сервіс Gmail.
- weather_service - сервіс погоди, представлений об'єктом класу WeatherService. Використовується для отримання інформації про погоду з відповідного API;
- tabs - віджет вкладок, представлений об'єктом QTabWidget. Використовується для організації різних секцій програми у вигляді вкладок;
- chat_widget - віджет чату, представлений об'єктом класу ChatWidget. Відповідає за інтерфейс чату та взаємодію з користувачем у розділі "Чати";
- statistics_widget - віджет статистики, представлений об'єктом класу StatisticsWidget. Відповідає за відображення статистичних даних у розділі "Статистика";

- weather_widget - віджет погоди, представлений об'єктом класу WeatherWidget. Відповідає за відображення інформації про погоду у розділі "Погода".

Детальний опис методів:

- __init__(self, config, db_handler) – конструктор класу, ініціалізує властивості db_handler, telegram_service, instagram_service, gmail_service, weather_service. Налаштовує основні параметри вікна, такі як заголовок та розміри. Викликає методи init_ui для налаштування інтерфейсу та start_services для запуску сервісів. Приймає два аргументи: config (конфігураційні налаштування) та db_handler (обробник бази даних). Нічого не повертає;
- init_ui(self) – ініціалізує інтерфейс користувача, створює основні елементи макету, такі як вкладки, віджети для чатів, статистики та погоди. Додає відповідні вкладки до віджета QTabWidget, встановлює центральний віджет вікна, ініціалізує меню та застосовує стилі. Приймає лише параметр self. Нічого не повертає;
- init_menu(self) – налаштовує меню програми, створює меню "Файл" з дією "Exit" для виходу з програми та меню "Довідка" з дією "Про програму", яке відображає інформаційне повідомлення. Приймає лише параметр self. Нічого не повертає;
- show_about(self) – відображає діалогове вікно з інформацією про програму, включаючи назву та версію. Використовується як обробник дії "Про програму" в меню "Довідка". Приймає лише параметр self. Нічого не повертає;
- set_styles(self) – встановлює стилі для різних елементів інтерфейсу програми за допомогою CSS. Налаштовує фон вікна, розміри шрифтів для списків, міток, кнопок та полів вводу. Приймає лише параметр self. Нічого не повертає;
- start_services(self) – запускає опитування для сервісів комунікації, таких як Telegram та Gmail, за допомогою методів start_polling

відповідних сервісів. Приймає лише параметр self. Нічого не повертає.

Клас StatisticsWidget містить такі властивості: db_handler, timer, layout, total_messages_label, average_response_time_label, platform_messages_label. Та методи: __init__, init_ui, update_statistics. Цей клас відповідає за відображення статистики вікна і оновлення статистичних даних про повідомлення.

Детальний опис властивостей:

- db_handler – ця властивість зберігає об'єкт класу, який відповідає за обробку даних з бази даних. Вона використовується для отримання статистики, наприклад, загальної кількості повідомлень або середнього часу відповіді;
- timer – ця властивість є об'єктом класу QTimer і використовується для періодичного оновлення статистики кожні 5 секунд;
- layout – ця властивість є об'єктом класу QVBoxLayout і відповідає за вертикальне розташування елементів на формі;
- total_messages_label – ця властивість є об'єктом класу QLabel і використовується для відображення загальної кількості повідомлень;
- average_response_time_label – ця властивість є об'єктом класу QLabel і використовується для відображення середнього часу відповіді на повідомлення;
- platform_messages_label – ця властивість є об'єктом класу QLabel і використовується для відображення статистики повідомлень по платформам.

Детальний опис методів:

- __init__(self, db_handler) – конструктор класу. Ініціалізує властивості класу, зокрема підключення до бази даних через db_handler. Викликає методи init_ui для налаштування інтерфейсу та update_statistics для початкового завантаження статистики. Створює таймер, який кожні 5 секунд викликає метод update_statistics;

- init_ui(self) – ініціалізує інтерфейс користувача. Створює елементи інтерфейсу, такі як лейбли для відображення статистики, та додає їх до макету;
- update_statistics(self) – оновлює статистику. Отримує нові дані з бази даних за допомогою db_handler та оновлює текстові значення на лейблах: кількість повідомлень, середній час відповіді та статистику по платформам.

Клас WeatherWidget містить такі властивості: weather_service, timer, layout, city_label, temperature_label, description_label, humidity_label, wind_speed_label. Та методи: __init__, init_ui, update_weather. Цей клас відповідає за відображення інформації про погоду та її оновлення.

Детальний опис властивостей:

- weather_service – ця властивість зберігає об'єкт класу WeatherService, який відповідає за отримання актуальних даних про погоду;
- timer – ця властивість є об'єктом класу QTimer і використовується для періодичного оновлення інформації про погоду кожні 10 хвилин;
- layout – ця властивість є об'єктом класу QVBoxLayout і відповідає за вертикальне розташування елементів інтерфейсу на формі;
- city_label – ця властивість є об'єктом класу QLabel і використовується для відображення назви міста;
- temperature_label – ця властивість є об'єктом класу QLabel і використовується для відображення температури в місті;
- description_label – ця властивість є об'єктом класу QLabel і використовується для відображення опису погодних умов;
- humidity_label – ця властивість є об'єктом класу QLabel і використовується для відображення рівня вологості;
- wind_speed_label – ця властивість є об'єктом класу QLabel і використовується для відображення швидкості вітру.

Детальний опис методів:

- __init__(self, weather_service) – конструктор класу. Ініціалізує властивості класу, зокрема підключення до служби погоди через weather_service. Викликає методи init_ui для налаштування інтерфейсу та update_weather для початкового завантаження даних про погоду. Створює таймер, який кожні 10 хвилин викликає метод update_weather;
- init_ui(self) – ініціалізує інтерфейс користувача. Створює елементи інтерфейсу, такі як лейбли для відображення даних про погоду, та додає їх до макету. Також задає стилі для відображення тексту;
- update_weather(self) – отримує поточні дані про погоду через метод get_weather класу WeatherService. Якщо дані успішно отримано, оновлює текстові значення на лейблах (місто, температура, опис, вологість, швидкість вітру). Якщо дані не вдалося отримати, виводить повідомлення про помилку.

Клас Config містить такі властивості: database_path, api_keys. Та методи: __init__. Цей клас відповідає за зчитування конфігураційних даних з файлу і надання доступу до них.

Детальний опис властивостей:

- database_path – ця властивість містить шлях до бази даних. Якщо значення не вказано в конфігураційному файлі, використовується значення за замовчуванням 'database/socialhub.db';
- api_keys – ця властивість містить словник з API-ключами. Якщо в конфігураційному файлі немає відповідних даних, властивість ініціалізується як порожній словник.

Детальний опис методів:

- __init__(self, config_file) – конструктор класу. При ініціалізації класу відкривається конфігураційний файл (формат JSON), і з нього завантажуються дані. Дані про шлях до бази даних та API-ключі

зберігаються в відповідних властивостях класу. Якщо у файлі відсутні ці параметри, використовуються значення за замовчуванням.

4.2 Розробка візуального інтерфейсу платформи

PyQt – це потужний інструмент для створення графічних інтерфейсів у Python, заснований на бібліотеці Qt, написаній на C++. Він дозволяє розробляти програми з професійним зовнішнім виглядом та багатим функціоналом.

PyQt надає велику кількість готових елементів керування (віджетів), які можна використовувати для створення інтерфейсів. Це можуть бути кнопки, текстові поля, зображення, таблиці, списки, прапорці, перемикачі, повзунки, індикатори прогресу та інші елементи. Кожен віджет має різні параметри, що дозволяють тонко регулювати розміри, кольори, шрифти, стилі та інші властивості.

Фреймворк пропонує гнучку систему компонування, де елементи можна розташовувати вертикально, горизонтально, як таблиць чи форм. Це дозволяє створювати складні інтерфейси з автоматичним настроюванням розмірів елементів при зміні розмірів вікна.

Однією з ключових особливостей PyQt є система сигналів та слотів, яка дозволяє обробляти події, що виникають під час взаємодії користувача з інтерфейсом. Сигнали повідомляють про зміни, а слоти виконують дії у відповідь. Це дозволяє створювати інтерактивні програми, які реагують на дії користувача.

За допомогою стильових таблиць, схожих на CSS, PyQt дозволяє створювати стильний та одноманітний інтерфейс, змінюючи зовнішній вигляд окремих елементів або всієї програми.

Фреймворк підтримує роботу з панелями інструментів, контекстними меню, діалоговими вікнами, анімаціями та багатьма іншими функціями, забезпечуючи зручний та ефективний досвід користувача. Ви також можете створювати власні віджети, розширюючи стандартний набір елементів.

PyQt дозволяє організувати програмний код з використанням модульної архітектури, що спрощує обслуговування та розширення проекту. Завдяки цьому фреймворк стає зручним інструментом для створення складних програм з розширеними можливостями налаштування та взаємодії з користувачем.

Використовуючи PyQt, можна створювати програми з високою функціональністю та сучасним дизайном. Хоча процес створення інтерфейсу виконується через код, а не візуальні редактори, PyQt надає всі необхідні інструменти для ефективної розробки. Активна спільнота та наявність документації сприяють його популярності серед розробників на Python.

4.3 Розробка посібника користувача

Посібник користувача є важливою складовою будь-якого програмного забезпечення, оскільки він забезпечує кінцевого користувача необхідною інформацією для правильного та ефективного використання платформи. У цьому розділі буде представлено детальні інструкції щодо налаштування та використання розробленого програмного забезпечення.

Для забезпечення коректної роботи системи необхідно виконати попередні налаштування та отримати доступ до відповідних сервісів. Нижче наведено покрокові інструкції для кожного етапу налаштування.

Детальні інструкції для отримання API ключів:

- Gmail API:
  1. перейдіть на Google Cloud Console (https://console.cloud.google.com/);
  2. створіть новий проект або виберіть існуючий;
  3. увімкніть Gmail API:
     a. у меню зліва знайдіть "API та сервіси" > "Бібліотека";
     b. знайдіть "Gmail API" та увімкніть його.
  4. створіть облікові дані:
     a. перейдіть до "API та сервіси" > "Облікові дані";
     b. натисніть "Створити облікові дані" > "OAuth 2.0 Client ID";

   c. виберіть тип додатку;
   d. заповніть необхідні поля.
5. налаштуйте екран OAuth:
   a. додайте дозволені домени;
   b. вкажіть області доступу (scopes).
6. завантажте JSON файл з обліковими даними;
7. додайте дозволені redirect URI.

- Telegram Bot API:
  1. відкрийте Telegram і знайдіть @BotFather;
  2. надішліть команду /newbot;
  3. дотримуйтесь інструкцій:
     a. введіть ім'я бота;
     b. введіть username бота (має закінчуватися на 'bot').
  4. BotFather надасть токен API - збережіть його;
  5. додаткові налаштування через команди:
     a. /setdescription - опис бота;
     b. /setabouttext - інформація про бота;
     c. /setuserpic - встановлення аватара;
     d. /setcommands - встановлення команд.
- OpenWeather API
  1. перейдіть на сайт OpenWeatherMap (https://openweathermap.org/);
  2. зареєструйте акаунт;
  3. після входу перейдіть до розділу "API keys":
     a. у головному меню виберіть "API keys" ;
     b. або перейдіть за прямим посиланням: https://home.openweathermap.org/api_keys.
  4. створіть новий API ключ:
     a. натисніть "Generate";
     b. введіть ім'я для ключа.
  5. дочекайтеся активації ключа (може зайняти до 2 годин);

6. виберіть відповідний тарифний план:
    a. безкоштовний (60 викликів/хвилину);
    b. різні платні плани.

Після отримання всіх необхідних API ключів та токенів, необхідно правильно їх інтегрувати у систему:

- файл облікових даних Google:
    - знайдіть завантажений JSON-файл з обліковими даними Google;
    - перемістіть його в кореневу папку проекту;
    - перейменуйте файл на "credentials.json".
- налаштування config.json:
    - відкрийте файл config.json в текстовому редакторі;
    - знайдіть поле "telegram" та вставте отриманий токен Telegram бота у лапки;
    - знайдіть поле "openweather" та вставте ключ API від OpenWeather у лапки;
    - збережіть внесені зміни.

Дотримання цих інструкцій забезпечить правильне налаштування та безперебійну роботу платформи. Рекомендується зберігати всі облікові дані та ключі доступу в надійному місці та регулярно перевіряти їх актуальність.

ВИСНОВОК

У ході дослідження було розглянуто інтеграцію веб-сервісів для створення єдиної комунікаційної платформи. Аналіз підтвердив актуальність та важливість цієї теми в умовах сучасного інформаційного суспільства. Науковий апарат дослідження охоплював визначення мети, об'єкта, предмета, формулювання проблеми та вибір методологічного підходу.

Предмет дослідження проаналізовано з урахуванням актуальних тенденцій та технологій у галузі інтеграції веб-сервісів. Проведено огляд і аналіз наявних рішень для виявлення їхніх сильних сторін і недоліків.

На основі аналізу сформульовано вимоги до платформи, зокрема функціональні, нефункціональні та вимоги безпеки. Розроблено концептуальну модель системи й архітектурне рішення, що включає детальний опис архітектури, дизайну платформи, інтерфейсів і взаємодії між компонентами.

Було створено, протестовано та оцінено прототип інтегрованої комунікаційної платформи. Тестування виявило низку недоліків, які потребують подальшого вдосконалення.

У результаті прототип успішно продемонстрував базову функціональність системи. Подальший розвиток передбачає оптимізацію, поліпшення продуктивності, забезпечення високого рівня безпеки та зручності використання на основі результатів тестування і зворотного зв’язку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Telegram APIs [Електронний ресурс] – Режим доступу до ресурсу: https://core.telegram.org/api.
2. python-telegram-bot docs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python-telegram-bot.org/en.
3. Google Cloud Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/docs.
4. Al Sweigart. "Automate the Boring Stuff with Python: Practical Programming for Total Beginners" [Електронний ресурс] – Режим доступу до ресурсу: https://automatetheboringstuff.com/.
5. Luciano Ramalho. "Fluent Python: Clear, Concise, and Effective Programming" [Електронний ресурс] – Режим доступу до ресурсу: https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/.
6. Michał Jaworski, Tarek Ziadé. "Expert Python Programming" [Електронний ресурс] – Режим доступу до ресурсу: https://www.packtpub.com/en-us/product/expert-python-programming-fourth-edition-9781801071109.
7. Mark Summerfield. "Programming in Python 3: A Complete Introduction to the Python Language" [Електронний ресурс] – Режим доступу до ресурсу: https://www.pearson.com/store/p/programming-in-python-3-a-complete-introduction-to-the-python-language/P100000161965.
8. Alex Xu. "System Design Interview – An Insider's Guide" [Електронний ресурс] – Режим доступу до ресурсу: https://www.systemdesigninterview.com/.
9. Weather API docs [Електронний ресурс] – Режим доступу до ресурсу: https://openweathermap.org/api.
10. Stack Overflow [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/.

Додаток А – Код програми

Клас Client:

```
class Client:
    def __init__(self, client_id, name, platform):
        self.client_id = client_id
        self.name = name
        self.platform = platform

    def to_dict(self):
        return {
            'client_id': self.client_id,
            'name': self.name,
            'platform': self.platform
        }
```

Клас Message:

```
class Message:
    def __init__(self, message_id, client_id, content,
timestamp, sender):
        self.message_id = message_id
        self.client_id = client_id
        self.content = content
        self.timestamp = timestamp
        self.sender = sender  # Новое поле

    def to_dict(self):
        return {
            'message_id': self.message_id,
            'client_id': self.client_id,
            'content': self.content,
            'timestamp': self.timestamp,
            'sender': self.sender
        }
```

Клас Statistics:

```
class Statistics:
    def __init__(self, db_handler):
        self.db_handler = db_handler

    def get_total_messages(self):
        return self.db_handler.get_total_messages()

    def get_average_response_time(self):
```

```
        return self.db_handler.get_average_response_time()

    def get_messages_by_platform(self):
        return self.db_handler.get_messages_by_platform()
```

Клас DatabaseHandler:

```
import sqlite3
from core.client import Client
from core.message import Message

class DatabaseHandler:
    def __init__(self, db_path):
        self.connection = sqlite3.connect(db_path,
check_same_thread=False)
        self.create_tables()

    def create_tables(self):
        cursor = self.connection.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS clients (
                client_id TEXT PRIMARY KEY,
                name TEXT,
                platform TEXT
            )
        ''')
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS messages (
                message_id INTEGER PRIMARY KEY AUTOINCREMENT,
                client_id TEXT,
                content TEXT,
                timestamp TEXT,
                sender TEXT,  -- Новая колонка
                FOREIGN KEY(client_id) REFERENCES
clients(client_id)
            )
        ''')
        self.connection.commit()

    def add_client(self, client: Client):
        cursor = self.connection.cursor()
        cursor.execute('''
            INSERT OR IGNORE INTO clients (client_id, name,
platform) VALUES (?, ?, ?)
        ''', (client.client_id, client.name, client.platform))
        self.connection.commit()

    def add_message(self, message: Message):
        cursor = self.connection.cursor()
        cursor.execute('''
```

```
            INSERT INTO messages (client_id, content, timestamp,
sender) VALUES (?, ?, ?, ?)
        ''', (message.client_id, message.content,
message.timestamp, message.sender))
        self.connection.commit()

    def get_messages_by_client(self, client_id):
        cursor = self.connection.cursor()
        cursor.execute('''
            SELECT message_id, client_id, content, timestamp,
sender FROM messages WHERE client_id = ?
        ''', (client_id,))
        rows = cursor.fetchall()
        return [Message(*row) for row in rows]

    def get_total_messages(self):
        cursor = self.connection.cursor()
        cursor.execute('SELECT COUNT(*) FROM messages')
        count = cursor.fetchone()[0]
        return count

    def get_response_times(self):
        return []

    def get_all_messages(self):
        cursor = self.connection.cursor()
        cursor.execute('''
            SELECT message_id, client_id, content, timestamp
FROM messages
        ''')
        rows = cursor.fetchall()
        return [Message(*row) for row in rows]

    def get_client_platform(self, client_id):
        cursor = self.connection.cursor()
        cursor.execute('''
            SELECT platform FROM clients WHERE client_id = ?
        ''', (client_id,))
        result = cursor.fetchone()
        return result[0] if result else None

    def get_messages_by_platform(self):
        cursor = self.connection.cursor()
        cursor.execute('''
            SELECT clients.platform, COUNT(messages.message_id)
FROM messages
            JOIN clients ON messages.client_id =
clients.client_id
            GROUP BY clients.platform
        ''')
        result = cursor.fetchall()
        return dict(result)
```

```
    def get_average_response_time(self):
        return 0

    def get_all_clients(self):
        cursor = self.connection.cursor()
        cursor.execute('''
            SELECT client_id, name, platform FROM clients
        ''')
        rows = cursor.fetchall()
        return [Client(*row) for row in rows]
```

Клас GmailService:

```
from __future__ import print_function
import os.path
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from core.message import Message
from core.client import Client
from datetime import datetime
import base64
from email.mime.text import MIMEText

SCOPES = ['https://www.googleapis.com/auth/gmail.modify']

class GmailService:
    def __init__(self, db_handler):
        self.db_handler = db_handler
        self.creds = None
        self.service = None
        self.user_id = 'me'
        self.authenticate()

    def authenticate(self):
        if os.path.exists('token.json'):
            self.creds =
Credentials.from_authorized_user_file('token.json', SCOPES)

        if not self.creds or not self.creds.valid:
            if self.creds and self.creds.expired and
self.creds.refresh_token:
                self.creds.refresh(Request())
            else:
                flow =
InstalledAppFlow.from_client_secrets_file('credentials.json',
SCOPES)
                self.creds = flow.run_local_server(port=0)

            with open('token.json', 'w') as token:
```

```
                token.write(self.creds.to_json())
        self.service = build('gmail', 'v1',
credentials=self.creds)

    def list_messages(self):
        try:
            results =
self.service.users().messages().list(userId=self.user_id,
labelIds=['INBOX'], q='is:unread').execute()
            messages = results.get('messages', [])
            for msg in messages:
                self.get_message(msg['id'])
        except Exception as e:
            print(f'Error fetching messages: {e}')

    def get_message(self, msg_id):
        try:
            msg =
self.service.users().messages().get(userId=self.user_id,
id=msg_id, format='full').execute()
            payload = msg.get('payload', {})
            headers = payload.get('headers', [])

            email_from = ''
            email_subject = ''
            for header in headers:
                if header['name'] == 'From':
                    email_from = header['value']
                if header['name'] == 'Subject':
                    email_subject = header['value']

            data = payload.get('body', {}).get('data')
            if not data:
                parts = payload.get('parts', [])
                for part in parts:
                    if part.get('mimeType') == 'text/plain':
                        data = part.get('body', {}).get('data')
                        break

            if data:
                text =
base64.urlsafe_b64decode(data).decode('utf-8')
            else:
                text = ''

            timestamp =
datetime.fromtimestamp(int(msg['internalDate']) / 1000)
            client_id = email_from
            name = email_from.split('@')[0]
            client = Client(client_id=client_id, name=name,
platform='Gmail')
            self.db_handler.add_client(client)
```

```
            message = Message(
                message_id=None,
                client_id=client_id,
                content=text,
                timestamp=str(timestamp),
                sender='client'
            )
            self.db_handler.add_message(message)
            self.service.users().messages().modify(
                userId=self.user_id,
                id=msg_id,
                body={'removeLabelIds': ['UNREAD']}
            ).execute()
        except Exception as e:
            print(f'Error processing message {msg_id}: {e}')

    def send_message(self, to_email, subject, message_text):
        message = MIMEText(message_text)
        message['to'] = to_email
        message['subject'] = subject
        raw_message =
base64.urlsafe_b64encode(message.as_bytes()).decode()
        try:
            message = self.service.users().messages().send(
                userId=self.user_id,
                body={'raw': raw_message}
            ).execute()
            print(f'Message sent to {to_email}')
            return message
        except Exception as e:
            print(f'Error sending message: {e}')

    def start_polling(self):
        import threading
        import time
        def poll():
            while True:
                self.list_messages()
                time.sleep(10)
        threading.Thread(target=poll, daemon=True).start()
```

Клас TelegramService:

```
import requests
from core.message import Message
from core.client import Client
from datetime import datetime
import threading
import time

class TelegramService:
```

```
    def __init__(self, api_key, db_handler):
        self.api_key = api_key
        self.db_handler = db_handler
        self.base_url =
f'https://api.telegram.org/bot{self.api_key}/'
        self.offset = None

    def get_updates(self):
        url = f'{self.base_url}getUpdates'
        params = {'timeout': 100, 'offset': self.offset}
        try:
            response = requests.get(url, params=params)
            result = response.json().get('result', [])

            for update in result:
                self.offset = update['update_id'] + 1
                message_data = update.get('message')
                if message_data:
                    self.process_message(message_data)
        except Exception as e:
            print(f"Error getting updates: {e}")

    def process_message(self, message_data):
        chat_id = message_data['chat']['id']
        name = message_data['chat'].get('first_name', '') + ' '
+ message_data['chat'].get('last_name', '')
        text = message_data.get('text', '')
        timestamp = datetime.fromtimestamp(message_data['date'])

        client = Client(client_id=str(chat_id),
name=name.strip(), platform='Telegram')
        self.db_handler.add_client(client)

        message = Message(
            message_id=None,
            client_id=str(chat_id),
            content=text,
            timestamp=str(timestamp),
            sender='client'
        )
        self.db_handler.add_message(message)

    def send_message(self, chat_id, text):
        url = f'{self.base_url}sendMessage'
        data = {'chat_id': chat_id, 'text': text}
        try:
            response = requests.post(url, data=data)
            return response.json()
        except Exception as e:
            print(f"Error sending message: {e}")

    def start_polling(self):
        def poll():
```

```
            while True:
                self.get_updates()
                time.sleep(1)

        threading.Thread(target=poll, daemon=True).start()
```

Клас WeatherService:

```
import requests

class WeatherService:
    def __init__(self, api_key):
        self.api_key = api_key
        self.city = "Kryvyi Rih"

    def get_weather(self):
        url = "http://api.openweathermap.org/data/2.5/weather"
        params = {
            'q': self.city,
            'appid': self.api_key,
            'units': 'metric',
            'lang': 'ru'
        }
        try:
            response = requests.get(url, params=params)
            data = response.json()
            if data.get('cod') != 200:
                return None

            weather_info = {
                'city': data['name'],
                'temperature': data['main']['temp'],
                'description':
data['weather'][0]['description'],
                'humidity': data['main']['humidity'],
                'wind_speed': data['wind']['speed']
            }
            return weather_info
        except Exception as e:
            print(f"Error fetching weather data: {e}")
            return None
```

Клас ChatWidget:

```
from PyQt6.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QListWidget, QTextEdit,
QPushButton,
```

```
    QListWidgetItem, QLabel, QSplitter, QLineEdit, QFrame,
QScrollArea, QWidget
)
from PyQt6.QtCore import Qt, QTimer, QSize
from core.message import Message
from datetime import datetime
from core.client import Client

class ChatWidget(QWidget):
    def __init__(self, db_handler, telegram_service,
instagram_service, gmail_service):
        super().__init__()

        self.quick_replies = [
            "Здрастуйте! Як можу вам допомогти?",
            "Дякую за звернення! Ми зв'яжемося з вами найближчим
часом.",
            "Ваше замовлення прийняте і обробляється.",
            "На жаль, даний товар тимчасово відсутній.",
            "Наш графік роботи з 9:00 до 18:00, понеділок-
п'ятниця.",
        ]
        self.db_handler = db_handler
        self.telegram_service = telegram_service
        self.instagram_service = instagram_service
        self.gmail_service = gmail_service

        self.selected_client_id = None

        self.init_ui()
        self.load_clients()

        self.timer = QTimer()
        self.timer.timeout.connect(self.refresh_messages)
        self.timer.start(2000)

    def init_ui(self):
        main_layout = QHBoxLayout()
        self.setLayout(main_layout)

        self.chat_list = QListWidget()
        self.chat_list.setFixedWidth(200)

self.chat_list.itemClicked.connect(self.on_chat_selected)

        self.chat_area = QWidget()
        chat_layout = QVBoxLayout()
        self.chat_area.setLayout(chat_layout)

        self.chat_header = QLabel("Виберіть чат")

self.chat_header.setAlignment(Qt.AlignmentFlag.AlignCenter)
```

```
        self.chat_header.setStyleSheet("font-weight: bold; font-
size: 16px;")
        chat_layout.addWidget(self.chat_header)

        self.message_list = QListWidget()
        chat_layout.addWidget(self.message_list)

        message_input_layout = QHBoxLayout()
        self.message_input = QLineEdit()
        self.message_input.setPlaceholderText("Введіть
повідомлення...")
        self.send_button = QPushButton("Надіслати")
        self.send_button.clicked.connect(self.send_message)
        message_input_layout.addWidget(self.message_input)
        message_input_layout.addWidget(self.send_button)
        chat_layout.addLayout(message_input_layout)

        self.quick_reply_scroll = QScrollArea()
        self.quick_reply_scroll.setFixedHeight(100)
        quick_reply_widget = QWidget()
        self.quick_reply_layout = QHBoxLayout()
        quick_reply_widget.setLayout(self.quick_reply_layout)

        for reply in self.quick_replies:
            button = QPushButton(reply)
            button.clicked.connect(lambda checked, text=reply:
self.insert_quick_reply(text))
            button.setFixedHeight(40)
            self.quick_reply_layout.addWidget(button)

        self.quick_reply_scroll.setWidget(quick_reply_widget)
        self.quick_reply_scroll.setWidgetResizable(True)

        chat_layout.addWidget(self.quick_reply_scroll)

        main_layout.addWidget(self.chat_list)
        main_layout.addWidget(self.chat_area)

        splitter = QSplitter(Qt.Orientation.Horizontal)
        splitter.addWidget(self.chat_list)
        splitter.addWidget(self.chat_area)
        main_layout.addWidget(splitter)

    def load_clients(self):
        self.chat_list.clear()
        clients = self.db_handler.get_all_clients()
        for client in clients:
            display_name = client.name if client.name else
client.client_id
            item = QListWidgetItem(f"{display_name}
({client.platform})")
            item.setData(int(Qt.ItemDataRole.UserRole),
client.client_id)
```

```
            self.chat_list.addItem(item)

    def insert_quick_reply(self, text):
        self.message_input.setText(text)

    def load_messages(self):
        self.message_list.clear()
        if self.selected_client_id:
            messages =
self.db_handler.get_messages_by_client(self.selected_client_id)
            for message in messages:
                timestamp = message.timestamp.split('.')[0]
                if message.sender == 'user':
                    sender_name = "Вы"
                    alignment = Qt.AlignmentFlag.AlignRight
                    color = Qt.GlobalColor.blue
                else:
                    sender_name = self.chat_header.text()
                    alignment = Qt.AlignmentFlag.AlignLeft
                    color = Qt.GlobalColor.red

                item_text = f"{sender_name}
[{timestamp}]:\n{message.content}"
                item = QListWidgetItem(item_text)
                item.setTextAlignment(alignment)
                item.setForeground(color)
                self.message_list.addItem(item)

    def refresh_messages(self):
        self.load_clients()
        if self.selected_client_id:
            self.load_messages()

    def send_message(self):
        content = self.message_input.text().strip()
        if content and self.selected_client_id:
            platform =
self.db_handler.get_client_platform(self.selected_client_id)
            if platform == 'Telegram':

self.telegram_service.send_message(self.selected_client_id,
content)
            elif platform == 'Instagram':

self.instagram_service.send_message(self.selected_client_id,
content)
            elif platform == 'Gmail':
                subject = "Відповідь"

self.gmail_service.send_message(self.selected_client_id,
subject, content)

            message = Message(
```

```
                message_id=None,
                client_id=self.selected_client_id,
                content=content,
                timestamp=str(datetime.now()),
                sender='user'
            )
            self.db_handler.add_message(message)

            self.message_input.clear()
            self.load_messages()

    def on_chat_selected(self, item):
        self.selected_client_id =
item.data(int(Qt.ItemDataRole.UserRole))
        client_name = item.text()
        self.chat_header.setText(client_name)
        self.load_messages()
```

Клас MainWindow:

```
from PyQt6.QtWidgets import QMainWindow, QTabWidget, QMenuBar,
QMenu
from PyQt6.QtGui import QAction
from ui.chat_widget import ChatWidget
from ui.statistics_widget import StatisticsWidget
from ui.weather_widget import WeatherWidget
from services import TelegramService, InstagramService,
GmailService, WeatherService

class MainWindow(QMainWindow):
    def __init__(self, config, db_handler):
        super().__init__()
        self.setWindowTitle('SocialHub')
        self.setGeometry(100, 100, 1024, 768)

        self.db_handler = db_handler
        self.telegram_service =
TelegramService(config.api_keys.get('telegram'), db_handler)
        self.instagram_service =
InstagramService(config.api_keys.get('instagram'), db_handler)
        self.gmail_service = GmailService(db_handler)
        self.weather_service =
WeatherService(config.api_keys.get('openweather'))

        self.init_ui()
        self.start_services()

    def init_ui(self):
        self.tabs = QTabWidget()
```

```
        self.chat_widget = ChatWidget(self.db_handler,
self.telegram_service, self.instagram_service,
self.gmail_service)
        self.statistics_widget =
StatisticsWidget(self.db_handler)
        self.weather_widget =
WeatherWidget(self.weather_service)

        self.tabs.addTab(self.chat_widget, "Чати")
        self.tabs.addTab(self.statistics_widget, "Статистика")
        self.tabs.addTab(self.weather_widget, "Погода")
        self.setCentralWidget(self.tabs)

        self.init_menu()
        self.set_styles()


    def init_menu(self):
        menubar = self.menuBar()

        file_menu = menubar.addMenu('Файл')
        exit_action = QAction('Exit', self)
        exit_action.triggered.connect(self.close)
        file_menu.addAction(exit_action)

        help_menu = menubar.addMenu('Довідка')
        about_action = QAction('Про програму', self)
        about_action.triggered.connect(self.show_about)
        help_menu.addAction(about_action)

    def show_about(self):
        from PyQt6.QtWidgets import QMessageBox
        QMessageBox.information(self, "Про програму",
"SocialHub\nВерсія 1.0")

    def set_styles(self):
        style = """
        QMainWindow {
            background-color: #F0F0F0;
        }
        QListWidget {
            font-size: 14px;
        }
        QLabel {
            font-size: 16px;
        }
        QPushButton {
            font-size: 14px;
        }
        QLineEdit {
            font-size: 14px;
        }
        """
```

```
        self.setStyleSheet(style)

    def start_services(self):
        self.telegram_service.start_polling()
        self.gmail_service.start_polling()
```

Клас StatisticsWidget:

```
from PyQt6.QtWidgets import QWidget, QVBoxLayout, QLabel
from PyQt6.QtCore import QTimer

class StatisticsWidget(QWidget):
    def __init__(self, db_handler):
        super().__init__()
        self.db_handler = db_handler
        self.init_ui()
        self.update_statistics()

        self.timer = QTimer()
        self.timer.timeout.connect(self.update_statistics)
        self.timer.start(5000)

    def init_ui(self):
        self.layout = QVBoxLayout()
        self.total_messages_label = QLabel()
        self.average_response_time_label = QLabel()
        self.platform_messages_label = QLabel()

        self.layout.addWidget(self.total_messages_label)
        self.layout.addWidget(self.average_response_time_label)
        self.layout.addWidget(self.platform_messages_label)

        self.setLayout(self.layout)

    def update_statistics(self):
        total_messages = self.db_handler.get_total_messages()
        average_response_time = 
self.db_handler.get_average_response_time()
        platform_messages = 
self.db_handler.get_messages_by_platform()

        self.total_messages_label.setText(f"Усього повідомлень: 
{total_messages}")
        self.average_response_time_label.setText(f"Середній час 
відповіді: {average_response_time} секунд")
        platform_stats = '\n'.join([f"{platform}: {count}" for 
platform, count in platform_messages.items()])
        self.platform_messages_label.setText(f"Повідомлення по 
платформам:\n{platform_stats}")
```

Клас WeatherWidget:

```
from PyQt6.QtWidgets import QWidget, QVBoxLayout, QLabel
from PyQt6.QtCore import QTimer
from services.weather_service import WeatherService

class WeatherWidget(QWidget):
    def __init__(self, weather_service):
        super().__init__()
        self.weather_service = weather_service

        self.init_ui()
        self.update_weather()

        self.timer = QTimer()
        self.timer.timeout.connect(self.update_weather)
        self.timer.start(600000)

    def init_ui(self):
        self.layout = QVBoxLayout()
        self.setLayout(self.layout)

        self.city_label = QLabel()
        self.temperature_label = QLabel()
        self.description_label = QLabel()
        self.humidity_label = QLabel()
        self.wind_speed_label = QLabel()

        self.layout.addWidget(self.city_label)
        self.layout.addWidget(self.temperature_label)
        self.layout.addWidget(self.description_label)
        self.layout.addWidget(self.humidity_label)
        self.layout.addWidget(self.wind_speed_label)

        style = """
        QLabel {
            font-size: 14px;
        }
        """
        self.setStyleSheet(style)

    def update_weather(self):
        weather = self.weather_service.get_weather()
        if weather:
            self.city_label.setText(f"Місто: {weather['city']}")
            self.temperature_label.setText(f"Температура:
{weather['temperature']} °C")
            self.description_label.setText(f"Опис:
{weather['description']}")
            self.humidity_label.setText(f"Вологість:
{weather['humidity']}%")
```

```
            self.wind_speed_label.setText(f"Швидкість вітру: 
{weather['wind_speed']} м/с")
        else:
            self.city_label.setText("Неможливо отримати дані про 
погоду.")
```

Клас Config:

```
import json

class Config:
    def __init__(self, config_file):
        with open(config_file, 'r') as f:
            data = json.load(f)
            self.database_path = data.get('database', 
'database/socialhub.db')
            self.api_keys = data.get('api_keys', {})
```

Файл main:

```
import sys
from PyQt6.QtWidgets import QApplication
from ui.main_window import MainWindow
from config import Config
import os

def main():
    app = QApplication(sys.argv)
    config = Config('config.json')

    if not os.path.exists('database'):
        os.makedirs('database')

    from database.db_handler import DatabaseHandler
    db_handler = DatabaseHandler(config.database_path)
    window = MainWindow(config, db_handler)
    window.show()
    sys.exit(app.exec())

if __name__ == '__main__':
    main()
```