МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«КОМПЛЕКСНА ПРОГРАМНА СИСТЕМА КОНТРОЛЮ НАЯВНОСТІ ЕНЕРГОПОСТАЧАННЯ»

Рівень бакалавр
Галузь знань 12 «Інформаційні технології»,
спеціальність 122 «Комп’ютерні науки»
Виконав здобувач 4 курсу
Геращенко Владислав Денисович

Керівник к.пед.н., доцент
Лобода Юлія Геннадіївна

Рецензент к.т.н., доцент
Манаков Сергій Юрійович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційних технологій
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп'ютерні науки»
Назва освітньої програми «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Наталія Логінова _____
«___»_____2024 року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Геращенко Владиславу Денисовичу

1. Тема роботи: «Комплексна програмна система контролю наявності енергопостачання»
   Керівник роботи: к.пед.н, доцент Юлія Геннадіївна Лобода
   затверджені наказом НУ «ОЮА» від «02» квітня 2024 року № 809-06.
2. Строк подання здобувачем роботи «20» травня 2024 року
3. Дата видачі завдання «15» вересня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Формування функціональних вимог	13.10.2023	
2	Аналіз та вибір засобів розробки	25.10.2023	
3	Розробка архітектури застосунку	17.11.2023	
4	Проєктування бази даних	29.11.2023	
5	Розробка серверної частини	14.01.2024	
6	Розробка клієнтської частини	15.03.2024	
7	Тестування на вразливості та аналіз ефективності запропонованого рішення	17.04.2024	
8	Оформлення звітної частини	27.04.2024	

Здобувач _____
(підпис) (прізвище та ініціали)
Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему “Комплексна програмна система контролю наявності енергопостачання” на здобуття першого (бакалаврського) рівня вищої освіти за спеціальністю 122 Комп’ютерні науки, освітньою програмою: Комп’ютерні науки, містить 12 рисунків, 1 таблицю, 2 додатки, 20 літературних джерел за переліком посилань. Робота виконана на 53 сторінках загального тексту і 37 сторінках основного тексту.

Мета дослідження є розробка комплексної системи виявлення наявності енергопостачання, що дозволяє надавати населенню надійну інформацію щодо статусу енергосистеми.

Об’єктом дослідження є комплексні програмні системи контролю наявності енергопостачання.

Предметом дослідження є процес розробки комплексної програмної системи контролю наявності енергопостачання.

У кваліфікаційній роботи розроблено систему наявності енергопостачання на основі API-серверу, що включає механізми, автентифікації користувачів, програмно реалізовано прототип системи, проведено експериментальні дослідження на модельних даних для перевірки ефективності та надійності системи. Результати роботи можуть бути використані для використання з ціллю підвищення інформованості населення.

Подальшими напрямками досліджень є розширення можливостей системи, кооперація з наявними системами.

Ключові слова: енергопостачання, комплексна програмна система, контроль, інформація, моніторинг, управління, API-сервер, Telegram-bot, асинхронність.

ABSTRACT

Qualification work on the topic “Comprehensive software system for controlling the availability of energy supply” for obtaining the first (bachelor's) level of higher education in the specialty 122 Computer Science, educational program: Computer Science, contains 12 figures, 1 table, 2 appendices, 20 literature sources listed in the references. The work is carried out on 53 pages of general text and 37 pages of main text.

The purpose of the study is to develop a comprehensive system for detecting the availability of power supply, which provides reliable information to the population regarding the status of the power system.

The object of the study is comprehensive software systems for controlling the availability of energy supply.

The subject of the study is the process of developing a comprehensive software system for controlling the availability of energy supply.

In the qualification work, a power supply availability system based on an API server has been developed, which includes user authentication mechanisms, a software-implemented system prototype, and experimental studies on model data to verify the system's effectiveness and reliability. The results of the work can be used to increase public awareness.

Further research directions include expanding the system's capabilities and cooperating with existing systems.

Keywords: power supply, comprehensive software system, monitoring, information, management, API server, Telegram bot, asynchronous.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API – application programming interface

СКБД – система керування базами даних

REST – representational state transfer

HTTP – hypertext transfer protocol

CRUD – create, read, update, delete

RDB – relational database

AOF – append-only file

SSL – secure sockets layer

ID – identifier

ВСТУП

Актуальність теми дослідження «Комплексна програмна система контролю наявності енергопостачання» має велике значення з огляду на сучасні глобальні виклики у сфері енергетики та стабільності інфраструктури. У світлі поточних тенденцій, пов'язаних з урбанізацією, цифровізацією та зростаючою потребою в електроенергії, забезпечення ефективного моніторингу та управління ресурсами стає критично важливим завданням. Комплексна програмна система дозволяє виявляти потенційні проблеми постачання електроенергії в режимі реального часу, що сприяє швидкому реагуванню та мінімізації ризиків відключення. Крім того, такі системи оптимізують процес розподілу електроенергії, знижують витрати та підвищують ефективність роботи енергосистем.

Важливо, що розробка та впровадження таких програмних рішень дозволяють виявляти слабкі місця в інфраструктурі та швидко їх усувати. Це підвищує надійність системи в цілому, забезпечує постійну доступність електроенергії для споживачів та підтримує економічний розвиток. Таким чином, дослідження та розробка комплексної програмної системи контролю наявності енергопостачання сприяють сталому розвитку енергетичної інфраструктури та підвищують енергетичну безпеку країни.

Кваліфікаційна робота пропонує розробку інноваційного рішення на основі API-сервера, що інтегрується з Telegram-ботом та мапою для забезпечення ефективного управління та моніторингу стану енергосистем. Використання Telegram-бота як інтерфейсу для кінцевих користувачів дозволяє швидко, надійно та зручно отримувати актуальну інформацію про відключення електроенергії, планові ремонтні роботи, а також надсилати повідомлення про аварійні ситуації на лініях. Інтерактивна мапа, яка взаємодіє з API-сервером, надає візуалізацію даних, дозволяючи користувачам візуально визначати масштаб проблем.

Проектування API-сервера передбачає створення надійної, масштабованої та безпечної платформи, здатну до інтеграції з різноманітними датчиками та

системами збору даних, які оперативно реагують на зміни в енергосистемі. Сервер буде обробляти вхідні дані, аналізувати їх і, за потреби, ініціювати відповідні команди.

Мета дослідження є розробка комплексної системи виявлення наявності енергопостачання, що дозволяє надавати населенню надійну інформацію щодо статусу енергосистеми.

Об'єктом дослідження є комплексні програмні системи контролю наявності енергопостачання.

Предметом дослідження є процес розробки комплексної програмної системи контролю наявності енергопостачання.

Для досягнення поставленої мети потрібно вирішити такі **завдання**:

- проаналізувати засоби розробки та їх варіації;
- спроектувати програмний продукт;
- розробити Telegram-bot;
- розгорнути комплексу систему;
- створити базу даних;
- провести тестування працездатності системи.

Практична значущість одержаних результатів. Використання Telegram-бота як інтерфейсу для кінцевих користувачів дозволяє швидко, надійно та зручно отримувати актуальну інформацію про відключення електроенергії та планові ремонтні роботи, а також надсилати повідомлення про аварійні ситуації на лініях. Це значно спрощує обмін інформацією між користувачами та операторами системи, підвищуючи готовність до аварійних ситуацій та оперативність реагування.

Візуалізація даних у реальному часі через мапу, що взаємодіє з API-сервером, дозволяє користувачам візуально оцінювати масштаби проблем та планувати альтернативні шляхи постачання ресурсів. Це сприяє більш ефективному плануванню та використанню наявних енергоресурсів.

Проектування API-сервера забезпечує надійну, масштабовану та безпечну платформу, здатну інтегруватися з різноманітними датчиками та системами збору

даних для оперативного реагування на зміни в енергосистемі. Сервер обробляє вхідні дані, аналізує їх і при необхідності ініціює відповідні команди, що дозволяє мінімізувати негативний вплив аварійних ситуацій та оптимізувати розподіл ресурсів. Таким чином, результати цього дослідження сприятимуть підвищенню ефективності та стійкості енергетичної інфраструктури.

1 АНАЛІЗ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

1.1 Аналіз предметної області

Комплексна програмна система виявлення наявності енергопостачання — це сучасне високотехнологічне рішення для безперервного моніторингу стану енергопостачання у різних мережах та установках. Вона побудована з використанням передових технологій, що дозволяють здійснювати автоматичне виявлення та аналіз стану електроенергетичних систем, а також контролювати наявність електроенергії в місцях споживання.

Основу системи становить мережа спеціалізованих API-клієнтів, які розташовані у стратегічно важливих точках енергетичної інфраструктури. Кожен із цих клієнтів виконує важливу функцію збору даних про різні параметри, зокрема наявність електроенергії, її напругу та інші електричні характеристики. Зібрані дані негайно передаються на центральний API-сервер для подальшої обробки та аналізу.

На центральному сервері розміщено програмне забезпечення, яке виконує ретельний аналіз отриманих даних, виявляючи і класифікуючи різні стани системи енергопостачання. Це може бути як нормальне функціонування, так і ситуації обмеженого постачання електроенергії або навіть повного відключення. Програмне забезпечення забезпечує точне визначення типу і характеру проблем, а також виявляє потенційні загрози, дозволяючи своєчасно реагувати на них [1].

Особливістю системи є її здатність автоматично надсилати сповіщення відповідальним особам або сервісним командам у разі виникнення проблем з енергопостачанням. Це робить можливим негайне реагування на інциденти, мінімізуючи час простою обладнання та знижуючи ризик значних втрат. Таким чином, система сприяє підтримці стабільної роботи критично важливої інфраструктури, забезпечуючи економічну ефективність і збереження ресурсів.

Ця комплексна програмна система демонструє глибоке розуміння сучасних вимог до моніторингу енергетичних ресурсів. Вона є важливим інструментом для будь-якої організації, яка прагне підтримувати безперебійну роботу своїх

електричних систем, ефективно реагуючи на інциденти та попереджаючи можливі проблеми.

Одним із ключових аспектів комплексної програмної системи виявлення наявності енергопостачання є її здатність адаптуватися до різноманітних конфігурацій мереж та інфраструктур. Це досягається за рахунок масштабованості архітектури, яка дозволяє ефективно розгортати систему як у невеликих підприємствах, так і на великих промислових об'єктах. Завдяки можливості інтеграції з існуючими системами керування та моніторингу, система надає комплексний огляд стану енергетичної мережі в реальному часі.

Програмне забезпечення дозволяє проводити глибокий аналіз історичних даних, виявляючи закономірності та тренди в енергоспоживанні. Це дає можливість не лише оперативно реагувати на поточні збої, але й прогнозувати потенційні проблеми на основі ранніх ознак, тим самим знижуючи ризик раптового відключення або обмеження постачання електроенергії.

Система також підтримує динамыічне налаштування порогів сповіщення, щоб відповідальні особи могли точно контролювати моменти, коли необхідно вживати заходів. Це запобігає надмірній чутливості системи та дозволяє уникнути надмірних сповіщень, водночас забезпечуючи швидке інформування про важливі інциденти.

Важливою особливістю є підтримка різних рівнів доступу для користувачів системи. Це гарантує, що критично важливі дані залишаються доступними лише для уповноважених співробітників, захищаючи їх від несанкціонованого доступу. Одночасно, система забезпечує зручний і надійний інтерфейс для всіх рівнів користувачів, дозволяючи швидко переглядати необхідну інформацію та реагувати на зміни в стані енергопостачання [2].

Завдяки потужному набору інструментів аналізу та гнучкій архітектурі, ця програмна система є невід'ємною частиною сучасних енергетичних інфраструктур. Вона забезпечує швидке реагування на проблеми, прозорість і контроль у керуванні енергоресурсами, що сприяє ефективній експлуатації обладнання, оптимізації витрат і підвищенню загальної надійності мереж.

1.2 Асинхронні операції

Асинхронні операції є фундаментальним аспектом сучасного програмування, дозволяючи комп'ютерам ефективно обробляти кілька задач одночасно, не чекаючи завершення кожної окремої операції. Це особливо корисно в мережевому програмуванні та будь-яких застосунках, де вхід або вихід даних може вимагати значних затримок, наприклад, при завантаженні файлів, взаємодії з базами даних або запитах до веб-сервісів [3].

Асинхронність дозволяє програмі продовжувати виконання інших задач, поки чекає на завершення відповідної операції, яка вимагає часу. В традиційному, синхронному програмуванні, програма б заблокувалась, чекаючи на відповідь від веб-сервера або завершення читання файлу. Асинхронні методи вирішують цю проблему, дозволяючи обробити затримку ефективніше. Різницю у часі під час виконання функцій можливо переглянути на рисунку 1.1 [4].

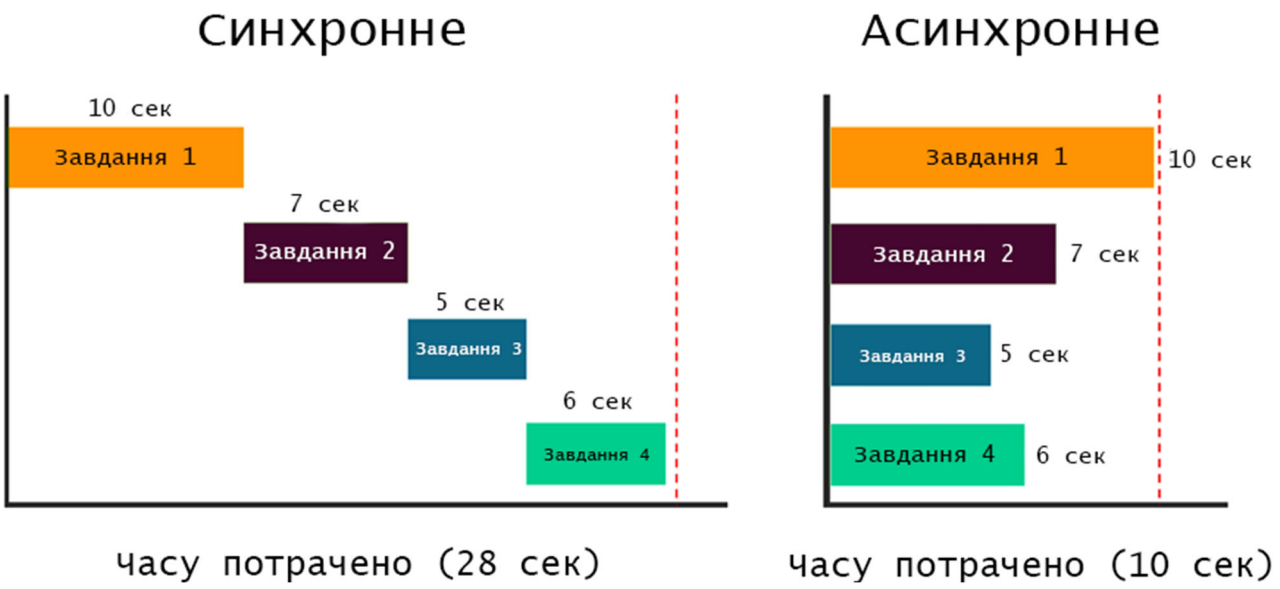


Рисунок 1.1 – Прискорення із використанням асинхронних функцій

Асинхронне програмування базується на використанні особливих мовних конструкцій, таких як «async» та «await» у Python, для виконання задач без блокування головного потоку програми. Це означає, що під час виконання асинхронної операції, яка потребує значного часу для завершення, основний потік

програми не стоїть на місці, а продовжує обробку інших завдань. Потім, коли результат асинхронної операції стає доступним, код повертається до цієї операції і обробляє її результати, забезпечуючи при цьому чистий та зрозумілий код.

Цей підхід допомагає відокремлювати запити, які потребують часу на виконання, від основного потоку програми. Наприклад, під час роботи з введенням та виведенням даних, асинхронність дозволяє уникнути блокування програми під час очікування відповіді від зовнішнього джерела, такого як база даних або веб-сервіс. Як результат, асинхронне програмування може значно зменшити затримки і підвищити продуктивність системи.

Асинхронність є особливо корисною для розробки API-серверів, де ефективна обробка запитів від великої кількості одночасних користувачів є критично важливою. Веб-сервери широко використовують асинхронне введення-виведення, щоб підтримувати велику кількість паралельних з’єднань на одному апаратному ресурсі. Це дозволяє їм оптимізувати використання серверних потужностей і знижує вартість обслуговування системи.

Асинхронне програмування також сприяє кращій масштабованості додатків. Розподіл завдань на кілька асинхронних процесів дає можливість гнучко керувати обробкою під час пікових навантажень. Завдяки цьому додатки можуть легко адаптуватися до збільшення кількості користувачів або зростання вимог до ресурсів.

У сучасних технологічних стеків асинхронність стала невід’ємною складовою, відіграючи ключову роль у розробці ефективних та високопродуктивних програмних рішень. Вона забезпечує швидку і плавну роботу додатків, мінімізує час простою системи та дозволяє обробляти великі обсяги даних, не втрачаючи продуктивності. У результаті розробники отримують можливість створювати гнучкі, масштабовані та ефективні системи, що відповідають вимогам сучасного світу технологій.

1.3 Специфіка розробки

Одним з основних аспектів розробки API є вибір архітектури та протоколів комунікації. Найпопулярнішим є REST (Representational State Transfer), який використовує стандартні HTTP методи такі як GET, POST, PUT, DELETE для взаємодії з ресурсами сервера. Така архітектура не тільки спрощує розробку, але й робить API більш зрозумілим та легким для інтеграції з іншими застосунками [5].

Перейдемо до іншої концепції – CRUD, яка є абревіатурою від Create, Read, Update, Delete – Створити, Прочитати, Оновити, Видалити. Ці дії становлять основу для більшості операцій з даними. CRUD використовується для того, щоб розподілити програмний код на окремі частини, кожна з яких відповідає за певні дії, які є фундаментальними для структурованого зберігання, обробки та відновлення даних у програмних системах [6].

Операція «Create» відповідає за створення нових записів у базі даних. У контексті API, це зазвичай здійснюється через POST запит (рисунок 1.2), який передає дані нового об’єкта (наприклад, користувача, продукта, або будь-якого іншого ресурсу) до API-сервера, який потім вставляє ці дані в базу даних.

```
POST /add_my_user/
{
 "user_name": "Oleksandr",
 "phone_number": "+380962311234"
}
```

Рисунок 1.2 – Приклад POST-запиту

READ відповідає за отримання інформації з бази даних. Ця операція в API часто використовує GET запити для запиту даних. Дані можуть бути запитані для одного конкретного об’єкта або у вигляді списку об’єктів, залежно від параметрів запиту. GET-запит можливо виконати за допомогою команди: «GET /users/135»

Операція UPDATE змінює існуючі дані. Вона зазвичай реалізується через PUT або PATCH запити (рисунок 1.3). PUT запити зазвичай оновлюють всі поля об’єкта, тоді як PATCH запити можуть змінити лише вказані поля.

```
PUT /update_my_users/
Content-Type: application/json

{
 "user_name": "Oleksiy",
 "phone_number": "+380991231234"
}
```

Рисунок 1.3 – Приклад UPDATE-запиту

DELETE видаляє існуючі дані. Операція реалізується через DELETE запит, який вказує на об’єкт, що потрібно видалити з бази даних. Приклад запиту «DELETE /users/135».

Важливою частиною розробки є також забезпечення безпеки. Це включає аутентифікацію і авторизацію користувачів, шифрування даних, а також контроль доступу до різних типів ресурсів. Аутентифікація зазвичай реалізується через API-ключи, за допомогою яких налаштовується лімітований доступ до певних ресурсів.

Крім того, необхідно буде звернути увагу на масштабування системи. API-сервер має бути здатний обробляти велику кількість запитів від багатьох користувачів одночасно. Використання розподілених систем може допомогти впоратися з великими обсягами трафіку та забезпечити високу доступність та надійність сервісу.

1.4 Асинхронність у розробці API

Асинхронність відіграє вирішальну роль у розробці API-серверів, особливо в контексті систем, що забезпечують високу продуктивність і спроможні обробляти значні обсяги даних або вимагають швидкої реакції на запити користувачів. Асинхронний сервер дозволяє ефективно обробляти запити в неблокуючому режимі, тобто такий сервер може приймати нові запити, поки очікує на відповідь від бази даних чи зовнішніх сервісів. Ця особливість критично важлива для забезпечення високої пропускної спроможності та ефективності системи [7].

Імплементація асинхронності в системах забезпечує значне зниження латентності відгуків сервера, оскільки процесорні потоки звільняються від необхідності простоювати в очікуванні завершення операцій вводу-виводу. Це дозволяє оптимізувати загальну продуктивність сервера, адже ресурси процесора можуть бути ефективно використані для інших завдань. Спеціалізовані бібліотеки, такі як asyncio в Python або Task Parallel Library (TPL) у C#, надають розробникам інструменти для ефективного управління асинхронними потоками. Вони забезпечують стабільну та надійну роботу програмного забезпечення, дозволяючи виконувати кілька завдань одночасно.

Завдяки асинхронним операціям веб-сервери можуть обслуговувати велику кількість паралельних з’єднань на одному апаратному ресурсі. Це значно підвищує ефективність використання серверних потужностей, що у свою чергу знижує витрати на обслуговування та підвищує якість надання послуг користувачам. Здатність обробляти численні з’єднання паралельно без істотних затримок є важливим аспектом у світі високих навантажень і динамічного веб-трафіку.

Однак проектування асинхронного API вимагає ретельно продуманої архітектури та компетентного управління потоками даних. Неправильне використання асинхронних викликів може спричинити проблеми зі стабільністю системи, наприклад, ситуації гонитви даних або невловимі баги, які складно діагностувати. Впровадження ефективних технічних рішень для асинхронної взаємодії вимагає ретельного планування та розуміння архітектури програми. Розробники повинні враховувати нюанси керування пам’яттю, синхронізації даних та обробки винятків, щоб гарантувати надійну та безперебійну роботу системи.

Вибір відповідних рішень для асинхронної взаємодії має стратегічне значення для успішної реалізації програмного коду. Це дозволяє досягти балансу між високою продуктивністю, масштабованістю та стабільністю системи, що особливо важливо у сучасному програмному середовищі, де вимоги до швидкості та надійності постійно зростають.

1.5 Аналіз наявних систем наявності енергопостачання

Через постійні перебої в енергетиці з’явилося декілька мап, на яких можливо переглянути де знайти світло. Одне із таких – «ЛУН Місто». Це багатофункціональний сайт із великою кількістю інтерактивних мап. Він надає інформацію щодо місць, де потенційно може бути світло [9].

Ще, можливо використовувати офіційний сайт ДТЕК-у, на якому за адресою можливо зрозуміти про стан енергетики [10].

Втім, варто визнати, що жодна з наявних систем не може гарантувати стовідсоткову достовірність інформації про наявність світла. Проблеми з енергетикою можуть виникати раптово і без попередження, що робить планування більш складним. Крім того, більшість існуючих платформ не має можливості надавати сповіщення у реальному часі про зміни в енергопостачанні, або робить це із затримкою. Це підкреслює потребу в розробці комплексної системи виявлення наявності енергопостачання.

Таблиця 1.1 – Аналіз переваг та недоліків наявних систем наявності енергопостачання

Назва продукту	Переваги	Недоліки
«ЛУН Місто»	Масштабність	Відсутність статусу у реальному часі
«ДТЕК»	Офіційне джерело	Не можливо оглянути увесь масштаб подій
«СвітлоБот»	Безпека під час повітряної тривоги, наявність застосунку	Відсутність можливості передачі інших показників живлення. Закритий API-сервер, затримка при вимкненні світла

Зіставляючи переваги та недоліки продуктів «ЛУН Місто», «ДТЕК» та «СвітлоБот», можна відзначити їх унікальні особливості й сфери застосування.

«ЛУН Місто» вирізняється масштабністю свого підходу, надаючи користувачам загальну картину розташування об'єктів у місті. Проте недоліком є відсутність статусу об'єктів у реальному часі, що може призвести до неточних даних і ускладнити прийняття рішень.

«ДТЕК» виступає офіційним джерелом інформації про енергопостачання, надаючи користувачам надійні дані. Однак складність у тому, що цей продукт не дозволяє оглянути повний масштаб подій та аналізувати ситуацію комплексно.

«СвітлоБот» забезпечує безпеку користувачів під час повітряної тривоги завдяки наявності застосунку. Проте обмеження включають відсутність можливості передачі інших показників живлення, а також використання закритого API-сервера. Додатково спостерігається затримка під час вимкнення світла, що може викликати труднощі у швидкому реагуванні на зміну ситуації.

Вибір між цими продуктами залежить від конкретних потреб користувача: «ЛУН Місто» підходить для загального планування, «ДТЕК» — для отримання офіційних даних, а «СвітлоБот» — для індивідуальної безпеки та отримання оперативної інформації в надзвичайних ситуаціях.

Після аналізу ринку було вирішено розробити систему, яка покриває усі недоліки наявних продуктів. А саме, необхідність розробки загальнодоступної системи із раптовою реакцією на зміни в енергетиці із зображенням загальної ситуації на мапі.

1.6 Висновки до розділу 1

У цьому розділі було проведено аналіз комплексної програмної системи виявлення наявності енергопостачання, що демонструє важливість та ефективність автоматизованого моніторингу стану електроенергетики в різних мережах та установках. За допомогою API-клієнтів, розміщених у критичних точках інфраструктури, система збирає важливі дані про електроенергію, які аналізуються для виявлення потенційних збоїв у енергопостачанні, забезпечуючи швидке реагування на аварійні ситуації.

Також у розділі було обговорено вагоме значення асинхронних операцій у програмуванні, що дозволяє системі продовжувати виконання інших задач під час очікування відповідей на запити, що багаторазово підвищує її продуктивність та ефективність. Асинхронність є ключовим елементом у розробці сучасних

високопродуктивних програмних систем, особливо в умовах, де швидкість відгуку та здатність обробляти великі обсяги запитів є критичними.

Розділ також включає огляд архітектурних підходів та методів взаємодії з даними, таких як CRUD операції та REST архітектура, які забезпечують структурованість, безпеку, та легкість управління ресурсами в межах API. Ці технології є фундаментом для розробки надійних і масштабованих систем, що можуть ефективно слугувати в різноманітних додатках.

Цей розділ підкреслює важливість інноваційних підходів в програмуванні та системній інтеграції для розробки ефективних рішень у галузі моніторингу енергопостачання, наголошуючи на здатності адаптуватися до викликів сучасних технологічних та енергетичних умов.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Огляд використовуваних технологій та інструментів

У даному підрозділі розглянуто основні технології та інструменти, які були використані для розробки веб-додатку. Сучасний підхід до розробки передбачає інтеграцію багатьох компонентів, що дозволяє створювати масштабовані, безпечні та високопродуктивні системи. У цьому контексті було обрано такі технології: Python з веб-фреймворком FastAPI, база даних PostgreSQL, Redis для кешування, бібліотеку Leaflet для інтерактивних карт, а також Docker та Docker Compose для розгортання усієї системи.

Python є високорівневою, інтерпретованою мовою програмування, що забезпечує широкі можливості для розробки веб-додатків. Він відомий своєю чистотою синтаксису та читабельністю коду. Через його гнучкість та простоту було обрано саме цю мову програмування для розробки основної частини системи.

FastAPI є сучасним веб-фреймворком для побудови API з використанням Python 3.7 та вище. Цей фреймворк відрізняється своєю високою продуктивністю, яку можна порівнювати із Node.js чи Go, завдяки асинхронній підтримці через Starlette (для веб-частини) та Pydantic (для валідації даних). FastAPI надає модернізований, швидкий та інтуїтивно зрозумілий спосіб для створення веб-додатків, що значно спрощує процес розробки і зменшує кількість потенційних помилок через строгу типізацію.

Для розробки комплексної програмної системи контролю наявності енергопостачання необхідно було обрати фреймворк, який б дозволяв швидко та зручно створювати API, таким чином, було обрано фреймворк FastAPI. Цей вибір було зумовлено кількома ключовими перевагами цього фреймворку порівняно із Django або Flask, які традиційно вважаються стандартом для веб-розробки в Python. По-перше, FastAPI вбудовано підтримує асинхронну обробку запитів, що є великою перевагою для високонавантажених систем, де потрібна висока пропускна спроможність та мінімізація часу відгуку. Використання асинхронного коду

дозволяє ефективніше використовувати ресурси сервера, тим самим знижуючи витрати.

По-друге, FastAPI автоматично генерує документацію за допомогою Swagger та ReDoc, що робить API більш доступними для розуміння та інтеграції з іншими розробниками та системами. Це особливо корисно в великих проектах, де час на розробку та впровадження є критичними факторами.

На відміну від Django, який більше підходить для повномасштабних проектів зі строго визначеною структурою, FastAPI надає більше гнучкості у виборі архітектури додатку, що дозволяє легко інтегрувати її з іншими сервісами та компонентами системи. Також, на відміну від Flask, який є мінімалістичним і зосереджується на простоті, FastAPI пропонує розширені функціональні можливості з самого початку, такі як залежності, авторизація, валідація даних і багато іншого, що робить його ідеальним вибором для розробки програмного коду, який потребує високої продуктивності, масштабованості та легкої інтеграції з іншими технологіями [11].

Для розробки було обрано базу даних PostgreSQL, тому що вона є однією з найпопулярніших та найбільш розширюваних СУБД у світі інформаційних технологій. Postgres підтримує як SQL (стандартний мову запитів), так і JSON для неструктурованих даних, що робить її дуже гнучкою.

Однією з ключових особливостей PostgreSQL є його висока масштабованість, яка дозволяє системі обробляти великі обсяги даних і складні операції. Крім того, система має розширену підтримку транзакцій, що включає в себе відмінну обробку паралелізму та забезпечення цілісність даних. Postgres підтримує різні варіанти реплікації та можливості автоматичного відновлення після збоїв, що робить її гарним рішенням для мого застосунку.

Для мови програмування Python існує бібліотека asyncpg, яка надає можливість асинхронного підключення до бази даних PostgreSQL. Цей інструмент має важливе значення в контексті розробки асинхронних API-серверів, оскільки API-сервер комплексної програмної системи контролю наявності енергопостачання також є асинхронним.

У контексті альтернативних рішень існують інші бібліотеки для роботи з PostgreSQL, такі як psycopg2 та її наступник psycopg3. Psycopg2 – це стандартний драйвер для PostgreSQL у Python, який працює синхронно. Однак, з часом виникла потреба у більш сучасних рішеннях, що призвело до створення psycopg3, який покращує обробку з'єднань та транзакцій, але все ще залишається в основному синхронним [12]. З іншого боку Asyncpg, була розроблена спеціально для асинхронного взаємодії із базами даних PostgreSQL і є значно швидшою за своїх синхронних попередників у використанні асинхронних патернів програмування. Це робить її ідеальним вибором для розробки комплексної програмної системи контролю наявності енергопостачання, де час відгуку та пропускна здатність мають критичне значення. Порівняти швидкість бібліотек можливо на рисунку 2.1.

Рисунок 2.1 – Швидкості різних бібліотек зв'язку із PostgreSQL

Redis використовується як інструмент для кешування та управління лімітами API. Це високопродуктивне рішення для зберігання даних в оперативній пам'яті, що підтримує різні типи даних, такі як рядки, списки, мапи, множини та сортовані множини. Основною перевагою Redis є його здатність забезпечувати дуже швидкий доступ до даних, що ідеально підходить для реалізації механізмів кешування. Приклад кешування зображено на рисунку 2.2 [13].

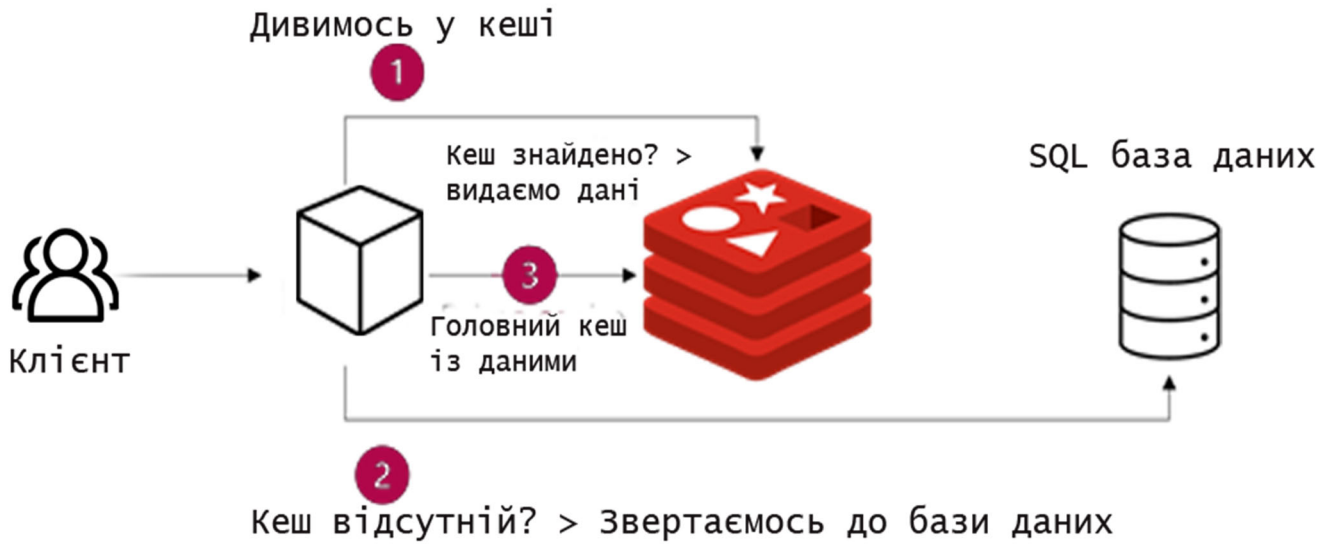


Рисунок 2.2 – Приклад використання Redis

В контексті кешування, Redis дозволяє тимчасово зберігати найчастіше запитувані дані, значно знижуючи навантаження на основну базу даних і покращуючи час відгуку системи. Такий підхід не тільки сприяє збільшенню продуктивності веб-додатку, але й зменшує затримку, що є критично важливим для користувацького досвіду.

Для управління лімітами API, Redis використовується для ведення лічильників, які контролюють кількість запитів, що користувач може відправити за певний проміжок часу. Це забезпечує ефективний контроль навантаження на сервер та запобігає зловживанням ресурсами. Використання Redis для цієї мети є особливо вигідним завдяки його можливості швидко обробляти великі обсяги транзакцій з мінімальними затримками.

Redis, хоч і є потужним інструментом для кешування та управління даними, має декілька обмежень та недоліків, які слід враховувати при виборі технологій для комплексної програмної системи:

1. Обмежена тривалість зберігання даних: оскільки Redis працює з даними в оперативній пам'яті, це може стати проблемою при зберіганні великих обсягів даних через високу вартість оперативної пам'яті порівняно з дисковим зберіганням. Також, дані можуть бути втрачені при відключенні або аварії системи, якщо не налаштовано постійне зберігання (persistence).

2. Модель постійного зберігання: хоча Redis пропонує декілька опцій для постійного зберігання даних, таких як RDB (snapshotting) та AOF (append-only file), жоден з цих методів не може гарантувати повну відмовостійкість без певних компромісів щодо продуктивності або цілісності даних при використанні окремо.

3. Безпека: Redis по замовчуванню не надає достатньо механізмів безпеки, таких як шифрування даних під час передачі (SSL) або автентифікація на основі більш складних правил. Це може стати проблемою, якщо Redis використовується в незахищеній мережі.

4. Відсутність вбудованих запитів і операцій зі складними даними: на відміну від традиційних баз даних, Redis не підтримує складні запити, такі як JOIN, або складну обробку даних. Це може ускладнити використання Redis для деяких типів додатків, де необхідні більш складні операції з даними.

5. Горизонтальне масштабування: хоча Redis підтримує кластеризацію для масштабування, налаштування і управління кластером Redis може бути складнішим порівняно з іншими базами даних, які нативно підтримують горизонтальне масштабування.

6. Залежність від одного вузла при реплікації: у стандартній конфігурації реплікації всі репліки синхронізуються з одним майстер-вузлом. Це створює ризик однієї точки відмови, якщо майстер-вузол вийде з ладу.

Для візуального відображення наявності енергопостачання, необхідно було обрати спосіб, який б наглядно та швидко показував інформацію по всьому місту. Таким чином було обрано бібліотеку Folium. Вона дозволяє створювати інтерактивні карти для веб-сторінок з використанням бібліотеки Leaflet.js. Ця бібліотека особливо корисна для аналітиків даних і розробників, які бажають візуалізувати географічні дані у зручний і ефективний спосіб. Приклад веб-інтерфейсу Folium можливо переглянути на рисунку 2.3.

Folium є максимально зручним інтерфейсом для Leaflet.js, дозволяючи використовувати Python для створення карти, додавання шарів, маркерів, кіл, полігонів та інших елементів. Ці об'єкти можна легко налаштувати з додатковими параметрами для кольорів, розмірів, підказок і модальних вікон.

Ключові особливості Folium включають:

1. Гнучкість: Можливість додавати різні типи візуалізацій і кастомізувати їх відповідно до потреб.

2. Інтеграція з Python екосистемою: Folium інтегрується з іншими бібліотеками аналізу даних у Python, такими як Pandas і Numpy.

3. Інтерактивність: Карти, створені за допомогою Folium, є інтерактивними, що дозволяє користувачам масштабувати, переміщуватись та взаємодіяти з картою.

Folium ідеально підходить для розробки комплексної програмної системи, де потрібно швидко візуалізувати великі обсяги геоданих та представляти їх у зручній і зрозумілій формі [14].

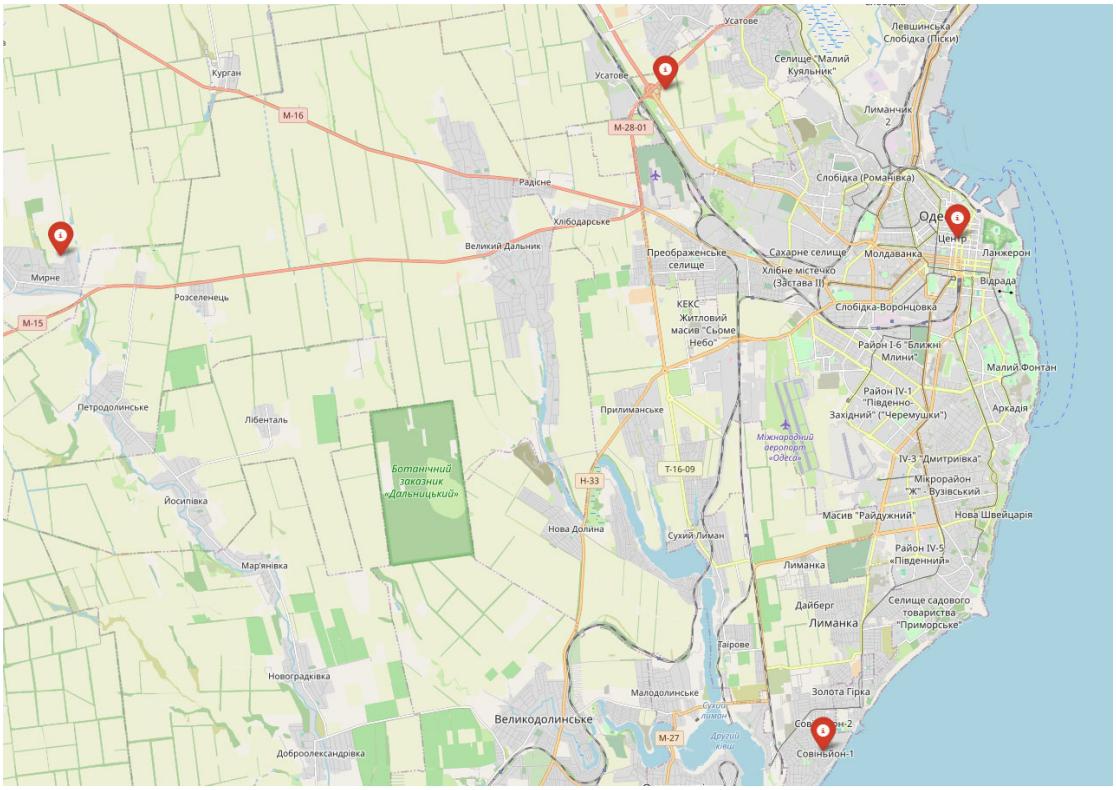


Рисунок 2.3 – Веб-інтерфейс Folium

Docker є потужною платформою, яка дозволяє розробникам пакувати свої додатки разом із їхніми залежностями у контейнери. Ці контейнери є легковісними, виконуються однаково на будь-якій системі та ізолюють додатки від одне одного та від основної системи. Це спрощує процес розгортання та запуску програмного забезпечення, оскільки контейнер містить усе необхідне для роботи додатка.

Docker Compose, в свою чергу, є інструментом для визначення та управління багатоконтейнерними додатками Docker. Він використовує YAML файл для конфігурації контейнерів, які становлять програму. Це дозволяє запускати всю програмну стеку однією командою, забезпечуючи цілісність середовища на різних машинах.

У контексті роботи було використано Docker для розгортання декількох служб: бази даних PostgreSQL, веб-фреймворку FastAPI, Redis та Telegram бота. Docker дозволяє ізолювати кожен компонент у своєму контейнері, що гарантує, що кожен компонент може працювати незалежно і мати свої специфічні залежності без конфліктів.

Docker Compose у цьому випадку допомагає легко керувати стартом, зупинкою та переконфігурацією цих контейнерів, що робить процес розробки більш простим і менш часозатратним. Можна описати всі необхідні служби в одному файлі docker-compose.yml, вказавши, як вони мають взаємодіяти між собою і які порти повинні бути відкритими. Це також спрощує процес внесення змін у систему, оскільки мені потрібно вносити зміни лише в одне місце [15].

Таким чином, використання Docker і Docker Compose у бакалаврській роботі є ключовим для створення ефективної, масштабованої та легко розгортаємої системи, яка може бути легко адаптована та розширена в майбутньому. Дозволяючи запускати усю систему на будь якому пристрої, що має Docker.

Telegram Bot-API є потужним інструментом, який дозволяє розробникам створювати користувальницьких ботів для месенджера Telegram. Цей API надає широкий спектр можливостей для взаємодії ботів із користувачами через стандартні чи спеціалізовані повідомлення, кнопки для швидких відповідей, інлайн-запити та багато іншого.

Також він дозволяє ботам виконувати багато різних функцій, наприклад відправляти повідомлення, фотографії, відео, стікери, аудіо, голосові повідомлення, а також управляти файлами і форматами даних, що використовуються для більш складних взаємодій. Боти можуть також

використовувати клавіатури для збору відповідей від користувачів, що робить інтерфейс більш інтерактивним і зручним.

Однією з ключових особливостей Telegram Bot-API є його висока швидкість та надійність, що дозволяє обробляти великі об'єми повідомлень із мінімальними затримками. Крім того, Telegram надає потужні засоби для захисту даних, включаючи end-to-end шифрування для гарантії конфіденційності спілкування [16].

У даному випадку, використання Telegram Bot-API дозволяє забезпечити зручний інтерфейс для кінцевих користувачів, де вони можуть взаємодіяти з API-сервером через знайомий месенджер. Бот буде слугувати засобом для автоматизації взаємодій, наприклад, для отримання API ключів, повідомлень про статус системи, управління задачами або навіть для інтеграції з іншими службами.

Використання цього API та месенджеру у бакалаврській роботі є прикладом того, як можна ефективно та надійно використовувати, вже наявний інтерфейс для взаємодії із API.

2.2 Аналіз архітектури комплексної системи

Архітектура комплексної програмної системи визначає її загальну структуру та взаємодію між компонентами. Ключові характеристики архітектури обумовлюються вимогами до функціональності, продуктивності та масштабованості. У межах розділу аналізується вибір архітектурного підходу та його переваги для конкретної системи.

Основні компоненти архітектури:

1. API-сервер:
– відповідає за обробку запитів від клієнтських додатків;
– забезпечує передачу даних між клієнтським додатком і базою даних;
– реалізує логіку бізнес-процесів та обробку асинхронних операцій.
2. Telegram-бот:
– служить інтерфейсом взаємодії користувача із системою;
– відправляє запити на API-сервер для отримання чи надсилання даних.

3. База даних:

– містить структуровану інформацію для функціонування системи;

– інтегрована з API-сервером для здійснення CRUD-операцій.

Архітектурний підхід охоплює клієнт-серверну структуру, яка розподіляє обробку даних між сервером та клієнтами, а також мікросервісну архітектуру, що дозволяє розподілити систему на окремі модулі для забезпечення гнучкості та масштабованості. До цього додається асинхронність, що мінімізує час простою через асинхронні операції. Запропонована архітектура має кілька переваг, зокрема масштабованість, яка дозволяє незалежно нарощувати окремі компоненти, високу надійність завдяки мікросервісному підходу, що запобігає загальним відмовам при проблемах в окремих модулях, та гнучкість, оскільки архітектура легко адаптується до змін у вимогах та додаванні нових модулів.

Такий підхід дає змогу створити надійну систему з високою продуктивністю та адаптивністю.

У межах аналізу архітектури комплексної системи приділяється особлива увага вибору архітектурного підходу та його відповідності вимогам функціональності, продуктивності й масштабованості. Архітектура системи складається з кількох ключових компонентів, які взаємодіють між собою для виконання визначених бізнес-процесів.

API-сервер виконує центральну роль у забезпеченні зв'язку між клієнтськими додатками та базою даних, обробляючи всі запити і реалізуючи бізнес-логіку. Він виступає як проміжна ланка, відповідаючи за управління потоками даних і асинхронними операціями.

Telegram-бот забезпечує зручний інтерфейс взаємодії користувача із системою, відправляючи запити на API-сервер для отримання або надсилання даних. Ця взаємодія є важливим аспектом архітектури, адже надає користувачам можливість доступу до функціоналу системи з будь-якого пристрою.

База даних служить сховищем структурованої інформації, необхідної для ефективного функціонування системи. Вона інтегрована з API-сервером для здійснення операцій створення, читання, оновлення та видалення даних.

Вибір архітектурного підходу здійснювався на користь клієнт-серверної моделі, що дозволяє ефективно розподілити обробку даних між клієнтами і сервером. У поєднанні з мікросервісною архітектурою це забезпечує гнучкість, оскільки різні модулі можуть розроблятися та масштабуватися незалежно один від одного. Застосування асинхронних операцій мінімізує затримки та забезпечує високу продуктивність.

Така архітектура гарантує не лише високу продуктивність і масштабованість системи, але й забезпечує її надійність, оскільки мікросервісний підхід дозволяє ізолювати відмови в окремих модулях, мінімізуючи вплив на інші частини системи. Завдяки цій структурі система залишається гнучкою, здатною адаптуватися до зміни вимог та швидкого додавання нових модулів.

2.3 Висновки до розділу 2

У розділі 2 було розглянуто проектування комплексної програмної системи контролю наявності енергопостачання, зокрема здійснено огляд використовуваних технологій та інструментів, які слугували основою для розробки комплексної системи. Проведений аналіз архітектури дозволив створити чітке бачення основних складових системи та принципів їх взаємодії.

Цей огляд включав ретельний підбір технологій, які відповідають завданням та заданим вимогам, а також враховують специфіку предметної області. Проектування архітектури комплексної системи дало можливість закласти міцний фундамент для її розробки та подальшого розгортання. Таким чином, цей розділ забезпечив критично важливу базу для реалізації всі\х необхідних компонентів програмного продукту, забезпечуючи їхню сумісність, ефективну взаємодію та оптимальну продуктивність.

Розробка архітектури також передбачала врахування масштабованості та гнучкості системи, що дозволить легко адаптуватися до зростаючих вимог та впроваджувати нові функції. Вибрані технології та інструменти були спрямовані на забезпечення високої надійності, швидкодії та безпеки системи. Враховуючи

сучасні тенденції у розробці програмних продуктів, особливу увагу приділено питанням взаємодії з користувачами, тому інтерфейс та функціональність системи були ретельно продумані.

Завдяки такому підходу проєктування програмного продукту в цьому розділі створило основу для подальшого розроблення компонентів системи, забезпечуючи ефективну інтеграцію API-сервера, Telegram-бота, мапи та бази даних в єдиний комплекс. Детальний аналіз архітектури вказує на можливість розширення системи та її оптимізації в майбутньому.

3 РОЗРОБКА ТА АНАЛІЗ ОТРИМАНИХ ДАНИХ

У даному розділі розглянуто створення API-серверу за допомогою FastAPI, включаючи елементи, такі як визначення схем даних, маршрутизація запитів, та асинхронна обробка запитів. Використання Pydantic для типізації даних та інтеграції з ORM бібліотеками. Візуальне представлення моделі та відповіді API-серверу викладено на рисунках. Додатково висвітлено розробку Telegram-бота з використанням бібліотеки Aiogram 3, описано основні команди для взаємодії з користувачем, такі як генерація API-ключа, видалення даних користувача, та перевірка статусу електропостачання. Окремий розділ присвячено розгортанню комплексної системи з використанням Docker Compose, включаючи деталі налаштування різних сервісів (контейнерів) і параметрів, необхідних для їх взаємодії та функціонування. Також охоплено створення та управління базою даних з використанням SQLAlchemy, деталізуючи структуру таблиць та зв'язки між ними.

3.1 Створення API-серверу

Для створення API-серверу за допомогою FastAPI, перш за все, необхідно визначити базові компоненти такі як схеми даних та маршрутизація запитів. Схеми даних дозволяють забезпечити строгу валідацію та типізацію вхідних і вихідних даних, а маршрутизація визначає, як API обробляє різні запити. Розглянемо детальніше процес розробки API-сервера, використовуючи Pydantic та FastAPI для визначення схем і асинхронний підхід до обробки запитів [17].

Основна модель, яку необхідно передавати користувачеві, це модель «Address». Її можливо переглянути на рисунку 3.1.

Розглянемо цю схему детальніше:

– id: це поле має тип int і слугує унікальним ідентифікатором адреси;
– latitude: поле типу float зберігає широту місцезнаходження адреси;
– longitude: зберігає довготу місцезнаходження;

– electricity_status: вказує на наявність або відсутність електропостачання;

– orm_mode: Цей параметр конфігурації дозволяє моделі Pydantic взаємодіяти з ORM бібліотеками, такими як SQLAlchemy [18]. Це означає, що об'єкти цієї моделі можуть бути легко перетворені з і в ORM моделі, що спрощує збереження і зчитування даних з бази даних.

```python
class Address(BaseModel):
    id: int = Field(None, description="Unique identifier of the address")
    latitude: float = Field(None, description="Latitude of the location")
    longitude: float = Field(None, description="Longitude of the location")
    electricity_status: bool = Field(
        None, description="Status of electricity supply at the location"
    )

    class Config:
        orm_mode = True
```

Рисунок 3.1 – Схема «Address»

Завдяки цій моделі, при запиті до API (рис. 3.3), наприклад, за допомогою «get_all_addresses», отримаємо структуровану відповідь, яку візуально зображено на рисунку 3.4. Цей запит може повертати лише адреси, що були позначені як неприватні. Таким чином, утворилася схема запиту до серверу (рис. 3.5).

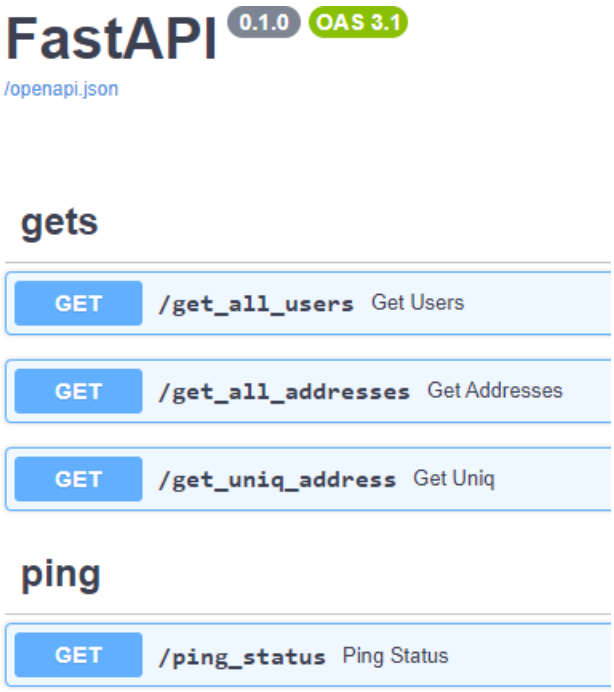


Рисунок 3.3 – Можливі запити до API-серверу

```
[
    {
        "id": 99,
        "latitude": 46.337942,
        "longitude": 30.68608,
        "electricity_status": true
    },
    {
        "id": 102,
        "latitude": 46.350114,
        "longitude": 30.676079,
        "electricity_status": true
    },
    {
        "id": 103,
        "latitude": 46.365763,
        "longitude": 30.651883,
        "electricity_status": true
    },
]
```


Рисунок 3.4 – Відповідь API-серверу

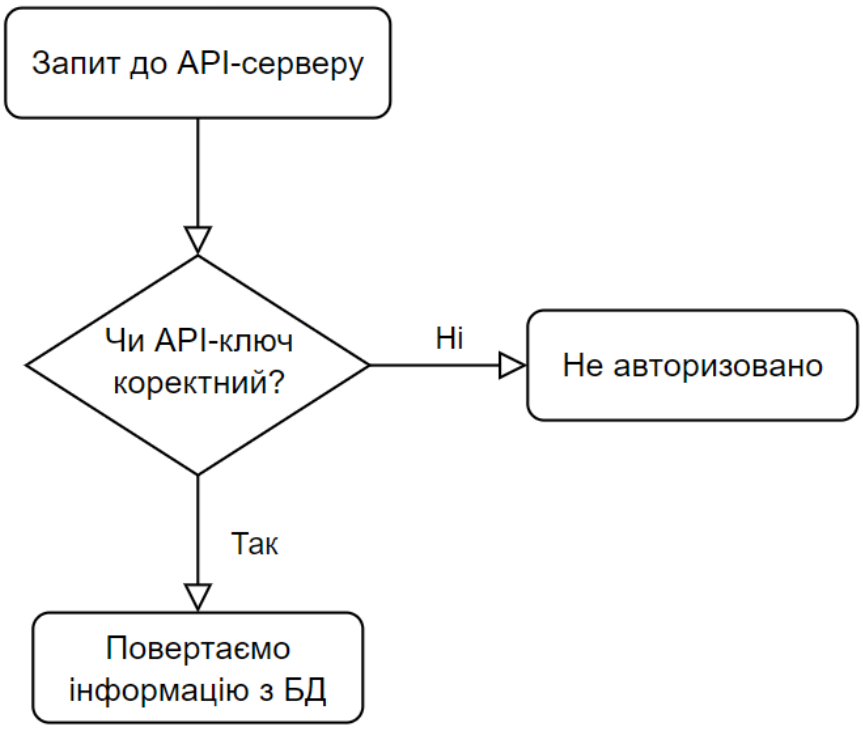


Рисунок 3.5 – Блок-схема запиту до API-серверу

3.2 Розробка Telegram-боту

У роботі було вирішено використати бібліотеку Aiogram 3 для створення телеграм бота. Було реалізовано кілька команд, які забезпечують взаємодію користувача з ботом. Кожна команда обслуговується спеціальним обробником (хендлером), який реагує на певні запити користувачів. Далі буде детально описано логіку роботи та призначення основних команд, реалізованих у рамках бота [19].

Команда «add_address». Ця команда призначена для додавання адреси користувачем у систему. Коли користувач відправляє команду add_address, йому пропонується ввести свою адресу у форматі координат або надіслати геолокацію через Telegram. Далі, користувач вводить або надсилає координати своєї адреси, які зберігаються в стані діалогу. Після введення адреси, система запитує користувача, чи коректно відображена адреса, що дозволяє користувачеві підтвердити або відредагувати надану інформацію. Ця команда допомагає забезпечити точність і актуальність даних у системі.

Команда «get_api_key» . Ця команда призначена для генерації та видачі API-ключа користувачу. Коли користувач відправляє команду /get_api_key, спочатку відбувається зчитування його ідентифікатора (ID) користувача в Telegram. Після цього система звертається до функції get_token, яка перевіряє наявність зареєстрованого токена для даного користувача в базі даних. Якщо токен знайдено, користувачу повертається його API-ключ з інструкцією про те, як його використовувати. Ця команда є критичною для забезпечення безпеки взаємодії з ботом, оскільки API-ключ є унікальним ідентифікатором, який дозволяє авторизувати запити до бота.

Команда «get_api_key». Ця команда призначена для генерації та видачі API-ключа користувачу. Коли користувач відправляє команду /get_api_key, спочатку відбувається зчитування його ідентифікатора (ID) користувача в Telegram. Після цього система звертається до функції get_token, яка перевіряє наявність зареєстрованого токена для даного користувача в базі даних. Якщо токен знайдено, користувачу повертається його API-ключ з інструкцією про те, як його

використовувати. Ця команда є критичною для забезпечення безпеки взаємодії з ботом, оскільки API-ключ є унікальним ідентифікатором, який дозволяє авторизувати запити до API-серверу.

Команда «remove_my_data». Вона дає користувачам можливість видалити всі свої дані з системи бота. При її активації відбувається запит користувацького ID, після чого система викликає функцію delete_user, яка відповідає за видалення даних користувача з бази. У відповідь користувач отримує повідомлення про статус видалення: успішне видалення або повідомлення про те, що дані вже були видалені раніше. Ця команда є важливою для забезпечення конфіденційності користувачів та дотримання правил захисту персональних даних.

Команда «status». Ця команда дозволяє користувачу дізнатися статус електропостачання прив’язаної точки. Після введення команди, бот звертається до функції get_my_status, яка перевіряє статус електрики для ID користувача. Відповідно до результату перевірки, користувач отримує повідомлення, що «Світло увімкнено!» або «Світло вимкнено!».

3.3 Розгортання комплексної програмної системи

Для розгортання комплексної програмної системи використовується Docker Compose, який дозволяє керувати багатоконтейнерними Docker-застосунками. Він дозволяє задати усі параметри застосунків усього в одному файлі, який за замовчуванням іменується як docker-compose.yml. Цей файл містить конфігурацію усіх сервісів, що необхідні для застосунків, який включає налаштування образів, портів, змінних оточення, залежностей між сервісами, та томів для зберігання даних. Завдяки Docker Compose, всі компоненти системи можуть бути розгорнуті в одному або декількох контейнерах з легкістю, що спрощує розробку та тестування.

Розглянемо файл docker-compose.yml, який містить сервіси (контейнери) , які необхідні для запуску комплексної системи:

1. tg_bot – сервіс, який відповідає за виконання телеграм бота, у який необхідно передати змінні віртуального оточення у вигляді «змінна=значення».

Для запуску бота було передано змінну, яка передає телеграм токен самого бота, що має наступне значення: «TG_BOT_TOKEN=1234:ABCD». Щоб захистити дані від витоку, використовуємо команду «env_file», вона допоможе підтягнути «.env» файл з зовнішнього джерела.

2. redis – сервіс, який використовується для кешування та прискорення роботи комплексної системи, наприклад для задання лімітів API-серверу. Для цього контейнеру було необхідно задати наступні команди:

– «ports: - "6379:6379"»: відкриття портів для зовнішнього доступу до контейнеру;

– «volumes: - /path/to/local/data:/root/redis»: задає незалежний образ диску, що дозволяє зберігати данні навіть при повному видаленні контейнеру.

3. web: сервіс, який використовується для запуску API-сервера, що обробляє веб-запити до застосунку. Для збірки образу цього контейнера було необхідно вказати наступні команди:

– «volumes: - ./:/code:ro»: забезпечує монтування коду додатка всередину контейнера тільки для читання;

– До оточення передано «DATABASE_URL»: змінна для підключення до бази даних, використовується для налаштування доступу до бази даних PostgreSQL;

– «ports: - "8000:80"»: відкриває порт 8000 на локальній машині та перенаправляє його на порт 80 у контейнері, що дозволяє доступ до веб-сервера через мережу.

4. db – сервіс, який використовується для роботи бази даних PostgreSQL. Для запуску контейнера з базою даних було необхідно визначити:

– «image: postgres:16-alpine»: використання легкої версії образу PostgreSQL на базі Alpine Linux;

– «volumes: - postgres_data:/var/lib/postgresql/data/»: створення та використання незалежного об'єму для зберігання даних бази даних, що дозволяє зберігати дані навіть після перезапуску контейнера;

– «expose: - 5434»: відкриття порту для внутрішніх з'єднань у мережі Docker;

– «ports: - "5434:5432"»: відкриття порту 5434 для доступу до бази даних ззовні, що є корисним для розробки, тестування та підключення до бази даних ззовні.

5. map – контейнер, що відповідає за мапу Folium, яка звертається до самого API-серверу, тому її можна розмістити незалежно від основної системи. Для цього контейнеру необхідно передати лише порт, за яким вона відкривається. Команда має вигляд «expose: - 8001».

3.4 Створення бази даних

Для початку створення бази даних необхідно визначити необхідні таблиці, із необхідними полями. У системі баз даних ми маємо дві основні сутності: api_users та addresses. Розглянемо створені таблиці:

1. api_users – це таблиця, що представляє користувачів API. Кожен запис у таблиці має наступні поля:

– id: унікальний ідентифікатор користувача, який автоматично збільшується при додаванні нового користувача;

– api_key: ключ API, який генерується автоматично за допомогою безпечного генератора токенів;

– created_at: дата та час створення запису користувача, встановлюється автоматично на поточний момент створення;

– last_usage: дата та час останнього використання API користувачем, може бути null якщо API не використовувалось;

– tg_id: унікальний ідентифікатор користувача в Telegram, забезпечує зв'язок між користувачем у базі даних та його обліковим записом у Telegram.

Ця таблиця також має зв'язок "один до багатьох" з таблицею addresses, де один користувач може мати декілька адрес.

2. addresses – таблиця, що зберігає інформацію про адреси. Кожен запис у таблиці включає:

– id: унікальний ідентифікатор адреси;

– is_private: булеве значення, що вказує, чи є адреса приватною;

– tg_id: ідентифікатор користувача API в Telegram, який використовується для зв'язку з таблицею api_users;

– electricity_status: булеве значення, що відображає статус наявності електрики на адресі;

– latitude: широта місцезнаходження адреси;

– longitude: довгота місцезнаходження адреси.

Таблиця addresses має зворотній зв'язок з таблицею api_users через поле tg_id, формуючи відносини "багато до одного", де кожна адреса асоційована з одним користувачем API. Переглянути ERD-діаграму сутностей [20] та зв’язків можливо на рисунку 3.5.

Для створення та управління таблицями api_users та addresses в базі даних використовується бібліотека SQLAlchemy. Що дозволило використати декларативний стиль з’єднання. Використовуючи декларативний стиль, є можливість описувати структури таблиць і зв'язки між ними у вигляді класів Python. Класи наслідуються від базового класу Base, який служить основою для автоматичного визначення колонок і таблиць у базі даних.

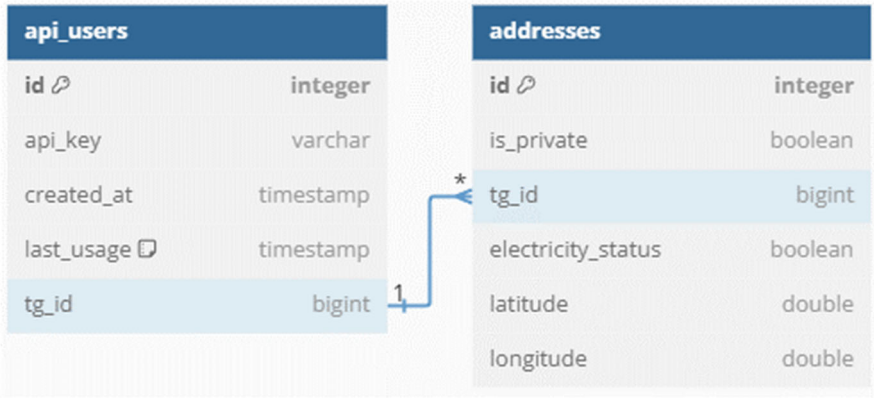


Рисунок 3.5 – ERD-діаграма бази даних

3.6 Розробка WEB-мапи

Для розробки інтерактивної мапи вирішено використати бібліотеку Folium. Це дозволило інтегрувати динамічні дані з геолокаційним контекстом та

відобразити їх у вигляді маркерів на карті, забезпечуючи інтуїтивно зрозуміле зображення статусу електропостачання за різними адресами.

Для забезпечення оперативного відгуку системи на зміни, впроваджено асинхронні запити до API за допомогою бібліотеки aiohttp, що дозволяє миттєво завантажувати актуальні дані про стан електропостачання.

Кожен отриманий з API адрес позначається на карті маркером, кольори яких різняться в залежності від стану електропостачання: зелений для активного стану та червоний для вимкненого. Маркери обладнані вікнами, які надають детальну інформацію про час останнього оновлення та поточний стан. Ця інформація критично важлива, адже дозволяє користувачам зрозуміти часові рамки змін у стані електропостачання.

Завершальна карта із всіма маркерами доступна через веб-інтерфейс, розроблено за допомогою FastAPI. Веб-сервер uvicorn запускає додаток, що забезпечує доступ до інтерактивної карти через будь-який веб-браузер, на якому може переглядати дані як з комп’ютера, так і з мобільних пристроїв. Це робить систему доступною для кінцевих користувачів, які можуть у реальному часі спостерігати за станом електропостачання у відповідних локаціях. Переглянути інтерактивну мапу, маркери, та вікна стану можливо переглянути на рисунку 3.6.

3.7 Тестування працездатності системи

Тестування працездатності системи є важливим етапом розробки, який дозволяє оцінити, наскільки розроблена система відповідає поставленим вимогам та функціонально працює в межах очікуваних сценаріїв. Мета цього етапу полягає в тому, щоб виявити та усунути можливі помилки в програмному продукті, забезпечити надійність і коректну роботу всієї системи.

Першочерговим завданням було проведення модульного тестування окремих компонентів. Кожен модуль розробленої системи тестувався окремо на відповідність технічним вимогам. Модульне тестування дозволило виявити потенційні помилки на ранніх етапах, коли їх виправлення не вимагало значних

витрат ресурсів. Зокрема, були протестовані модулі API-сервера, Telegram-бота та бази даних, що дозволило переконатися у правильній передачі та обробці даних між цими компонентами.

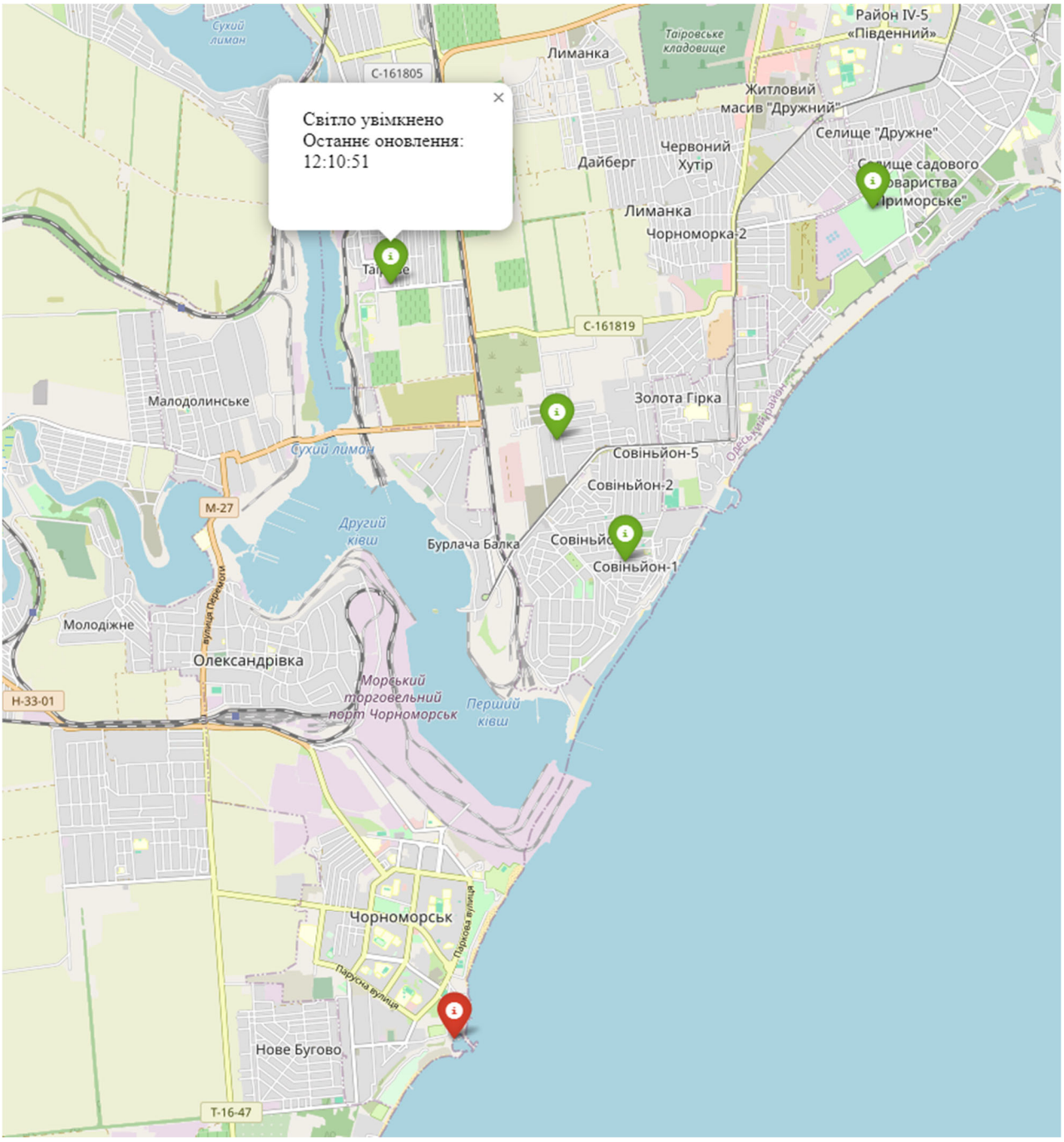


Рисунок 3.6 – Інтерактивна мапа

Далі було проведено інтеграційне тестування, яке дало змогу оцінити коректну роботу компонентів у взаємодії один з одним. Інтеграційне тестування охоплювало перевірку повного циклу обробки даних, починаючи від введення

через Telegram-бот, передавання їх до API-сервера та збереження в базі даних. Під час цього етапу були виявлені та виправлені декілька незначних проблем у логіці обробки даних.

Завершальним етапом тестування стало системне тестування, яке включало комплексну перевірку всього програмного продукту в цілому. Метою системного тестування було перевірити працездатність системи у реальних умовах експлуатації та під час різних навантажень. Було перевірено, як система справляється з різними типами користувацьких запитів, асинхронною обробкою даних та іншими сценаріями. В результаті було підтверджено, що система здатна стабільно обробляти запити, зберігаючи високу продуктивність навіть під час пікових навантажень.

3.6 Висновки до розділу 3

У розділі 3 розглянуто розробку комплексної програмної системи для моніторингу наявності енергопостачання. Основні компоненти розглянуті у цьому розділі включають створення API-серверу на основі FastAPI, розробку Telegram-бота з використанням бібліотеки Aiogram 3, розгортання системи через Docker Compose, та створення бази даних з використанням SQLAlchemy.

Аналіз розробки API-серверу продемонстрував ефективність асинхронної обробки запитів та використання схем даних для забезпечення строгих вимог до типізації даних. Це забезпечило високу продуктивність та надійність у взаємодії з базою даних і відповіді користувачам.

Розробка Telegram-бота зосереджена на забезпеченні інтерактивного спілкування з користувачами та управлінні основними функціями, як генерація API-ключів та видалення даних. Такий підхід забезпечив зручність та безпеку користування системою.

Впровадження Docker Compose дозволило легко керувати розгортанням декількох сервісів, забезпечуючи їхню ізольованість і одночасне функціонування

без конфліктів. Це забезпечило гнучкість, масштабованість та безвідмовність системи.

Створення та управління базою даних демонструє чітку організацію даних, ефективну інтеграцію з іншими компонентами системи та підвищує загальну продуктивність завдяки оптимізації запитів.

Цей розділ забезпечив детальне розуміння технічних аспектів системи та її компонентів, підтверджуючи можливості для подальшої оптимізації та розширення. Такий підхід до розробки забезпечує не тільки високу надійність та продуктивність, але й враховує потреби користувачів і забезпечує зручність використання, відкриваючи шлях для майбутніх удосконалень.

ВИСНОВКИ

Робота охоплює комплексне дослідження та розробку системи з використанням сучасних інструментів і технологій. Початковим етапом було дослідження предметної області та аналіз асинхронних операцій, що надало розуміння ключових вимог і особливостей реалізації. Було виявлено специфічні труднощі в асинхронній розробці API та здійснено аналіз наявних систем енергопостачання, щоб інтегрувати оптимальні підходи в проєктування.

Ретельно обравши технології та інструменти, перейдено до проєктування архітектури комплексної системи, забезпечуючи можливість її масштабування та модифікації. Ключовим елементом став API-сервер, створений для ефективної взаємодії між різними компонентами системи. Telegram-бот було розроблено як користувацький інтерфейс, надаючи користувачам швидкий і зручний доступ до необхідної інформації. Розгортання системи вимагало комплексної організації інфраструктури, включаючи створення бази даних для безпечного зберігання та обробки інформації. Після завершення розробки було проведено комплексне тестування працездатності системи, яке підтвердило її відповідність поставленим вимогам, надійність і стабільність роботи.

Було проведено аналіз наявних інструментів та їх варіацій, спроєктовано та розроблено програмний продукт, включно з Telegram-ботом та інтерактивною мапою і розгорнуто комплексну систему. Додатково створено базу даних та забезпечено її інтеграцію з іншими частинами системи, а також виконано детальне тестування для підтвердження функціональної готовності. Бакалаврська робота ставить перед собою мету створення ефективної, масштабованої та надійної системи управління інформацією. Інтеграція сучасних інструментів і асинхронних технологій забезпечила продуктивний та гнучкий програмний продукт, який легко адаптується під конкретні вимоги користувачів і має великий потенціал для подальшого розвитку.

Під час роботи над поставленими задачами особливу увагу було приділено оптимізації всіх компонентів системи для забезпечення їх стабільної роботи.

Проєктування архітектури передбачало ретельне планування кожного рівня, щоб система залишалася масштабованою та здатною впоратися зі зростанням обсягу даних і навантаженням. API-сервер, що є центральною частиною системи, розроблено з урахуванням асинхронної обробки запитів, що дозволило значно покращити продуктивність.

Telegram-бот реалізовано з урахуванням зручності користувачів. Інтерфейс бота інтуїтивно зрозумілий, а можливість взаємодії з системою через прості текстові команди дозволяє користувачам швидко отримувати необхідну інформацію або надсилати дані. Вибір Telegram-бота як інструмента взаємодії також дозволив спростити процес авторизації та доступу до системи.

База даних, яка забезпечує збереження всіх операційних даних системи, спроєктована для ефективного та швидкого доступу до інформації. База побудована так, щоб забезпечити цілісність даних і підтримку високих навантажень. Розгортання системи здійснено на хмарній інфраструктурі, що забезпечило легке масштабування та високу доступність сервісу.

Після завершення розробки всіх компонентів було проведено детальне тестування системи. Тестування охопило всі аспекти працездатності, включаючи стабільність роботи при високих навантаженнях, коректність обробки даних і безпеку. Тестування підтвердило, що система відповідає початковим вимогам і може ефективно працювати в умовах високого навантаження.

Підсумовуючи, було досягнуто поставлених цілей завдяки використанню сучасних інструментів і технологій. Було успішно створено комплексну систему управління даними з гнучкою архітектурою, здатною до масштабування, що дозволяє адаптуватися до нових вимог. Інтеграція асинхронних операцій та Telegram-бота в поєднанні з надійною базою даних забезпечили ефективний і зручний для користувачів продукт. Система готова до використання в реальних умовах, і завдяки своїй архітектурі може бути розширена або модифікована для задоволення нових потреб.

СПИСОК ВИКОРИСТАННИХ ДЖЕРЕЛ

1. Педан М.О. Розроблення автоматизованої системи електропостачання споживачів Полтавської області. Полтава: Національний університет імені Юрія Кондратюка, 2024. 93 с.
2. Системи електропостачання. Елементи теорії та приклади розрахунків : навч. посіб. / М. Й. Бурбело, О. О. Бірюков, Л. М. Мельничук. Вінниця : ВНТУ, 2011. – 204 с.
3. Фомін О. Асинхронне програмування як сучасний підхід у вирішенні алгоритмічних задач за допомогою концепцій мови програмування Javascript. *Collection of Scientific Papers «SCIENTIA»*, 2022. С. 8–12.
4. Use Async-Await With SwiftUI. URL: https://blog.egesucu.com.tr/use-async-await-with-swiftui-4bd12a0c7898
5. What is REST? URL: https://restfulapi.net/
6. Muqorobin M., Nendy A. Comparison of PHP Programming Language with Codeigniter Framework in Project CRUD. International Journal of Computer and Information System (IJCIS), 2022. Vol. 03, Issue 03. PP.. 94-98.
7. Двірна О. А.,. Паніон М. В. Сучасні тенденції в розробці телеграм-ботів. *Полтавська політехніка імені Юрія Кондратюка*, 2022.С. 367-369.
8. Розробка з asyncio¶. URL: https://docs.python.org/uk/3/library/asyncio-dev.html
9. ЛУН Місто. URL: https://misto.lun.ua/
10. ДТЕК. URL: https://www.dtek-oem.com.ua/ua
11. FastAPI, 2024 [Електронний ресурс]. – URL: https://fastapi.tiangolo.com/
12. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: https://www.postgresql.org/
13. Redis. URL: https://redis.io/
14. Folium. URL: https://python-visualization.github.io/folium/latest/
15. Ibrahim M.H., Sayagh M., Hassan, A. E. A study of how Docker Compose is used to compose multi-component systems. *Empir Software Eng* , 2021. Vol. 26, 128. URL: https://link.springer.com/article/10.1007/s10664-021-10025-1

16. Telegram Bot API. URL: https://core.telegram.org/bots/api
17. Pydantic. URL: https://docs.pydantic.dev/latest/
18. he Python SQL Toolkit and Object Relational Mapper. SQLAlchemy. URL: https://www.sqlalchemy.org/
19. Aiogram. URL: https://docs.aiogram.dev/uk-ua/latest/
20. Діаграма зв’язків сутностей: приклади, символи та вказівки для створення. URL: https://www.mindonmap.com/uk/blog/relationship-diagram/

ДОДАТКИ

Додаток А. Програмний код

Додаток А.1. Код головного файлу API-сервера

```python
import redis.asyncio as redis
import uvicorn
from fastapi import FastAPI, Depends
from fastapi.middleware.cors import CORSMiddleware

from fastapi_limiter import FastAPILimiter
from fastapi_limiter.depends import RateLimiter

from starlette.middleware.base import BaseHTTPMiddleware

from api.routes.ping import ping_router
from api.routes.posts import router
from api.schemas.models import HealthResponse

from contextlib import asynccontextmanager

from database.connection import sessionmanager
from database.models import Base


1 usage  bebus
@asynccontextmanager
async def lifespan(_: FastAPI):
    redis_connection = redis.from_url( url: "redis://redis:6379", encoding="utf8")
    await FastAPILimiter.init(redis_connection)
    yield
    await FastAPILimiter.close()


app = FastAPI(lifespan=lifespan)

origins = ["*"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(router=router)
app.include_router(router=ping_router)


 bebus
@app.get( path: "/", response_model=HealthResponse, dependencies=[Depends(RateLimiter(times=60, seconds=60))])
async def health():
    async with sessionmanager.connect() as connection:
        await connection.run_sync(Base.metadata.create_all)
    return HealthResponse(status="Ok")
```

Додаток А.2 Код WEB-мапи

```python
import datetime
import uvicorn
from fastapi import FastAPI
import folium
from starlette.responses import HTMLResponse
import aiohttp

app = FastAPI()
my_map = folium.Map(control_scale=True, location=(48.90793813554786, 31.393633712579252), zoom_start=6.5)


1 usage  bebus
async def get_all_addresses():
    URL = "http://web/get_all_addresses"
    async with aiohttp.ClientSession() as session:
        params = {"api_key": "3TRRwA_-5l_xyEuWztGzww"}
        async with session.get(URL, params=params) as response:
            print(response)
            answer: list[dict] = await response.json()
            print(answer)
            return answer


1 usage  bebus *
async def add_markers(folium_map):
    format_string = "%Y-%m-%dT%H:%M:%S.%f"
    addr = await get_all_addresses()
    for point in addr:
        if point["last_update"] is not None:
            last_update = datetime.datetime.strptime(point["last_update"], format_string).strftime("%H:%M:%S")
        else:
            last_update = "Немає інформації"
        if point["electricity_status"]:
            iframe = folium.IFrame(f"Світло увімкнено <br>Останнє оновлення: <br>{last_update}")
            popup = folium.Popup(iframe, min_width=160, max_width=160)
            folium.Marker(location=[point['latitude'], point['longitude']],
                          icon=folium.Icon(color='green'),
                          popup=popup).add_to(folium_map)
        else:
            iframe = folium.IFrame(f"Світло вимкнено! <br>Останнє оновлення: <br>{last_update}")
            popup = folium.Popup(iframe, min_width=160, max_width=160)
            folium.Marker(location=[point['latitude'], point['longitude']],
                          icon=folium.Icon(color='red'),
                          popup=popup).add_to(folium_map)


bebus
@app.get( path: "/", response_class=HTMLResponse)
async def root():
    my_map = folium.Map(control_scale=True, location=(48.90793813554786, 31.393633712579252), zoom_start=6.5)
    await add_markers(my_map)
    body_html = my_map.get_root()
    return body_html.render()


if __name__ == '__main__':
    uvicorn.run(app, port=8001, host='0.0.0.0')
```

Додаток А.3 Код взаємодії API-серверу із базою даних

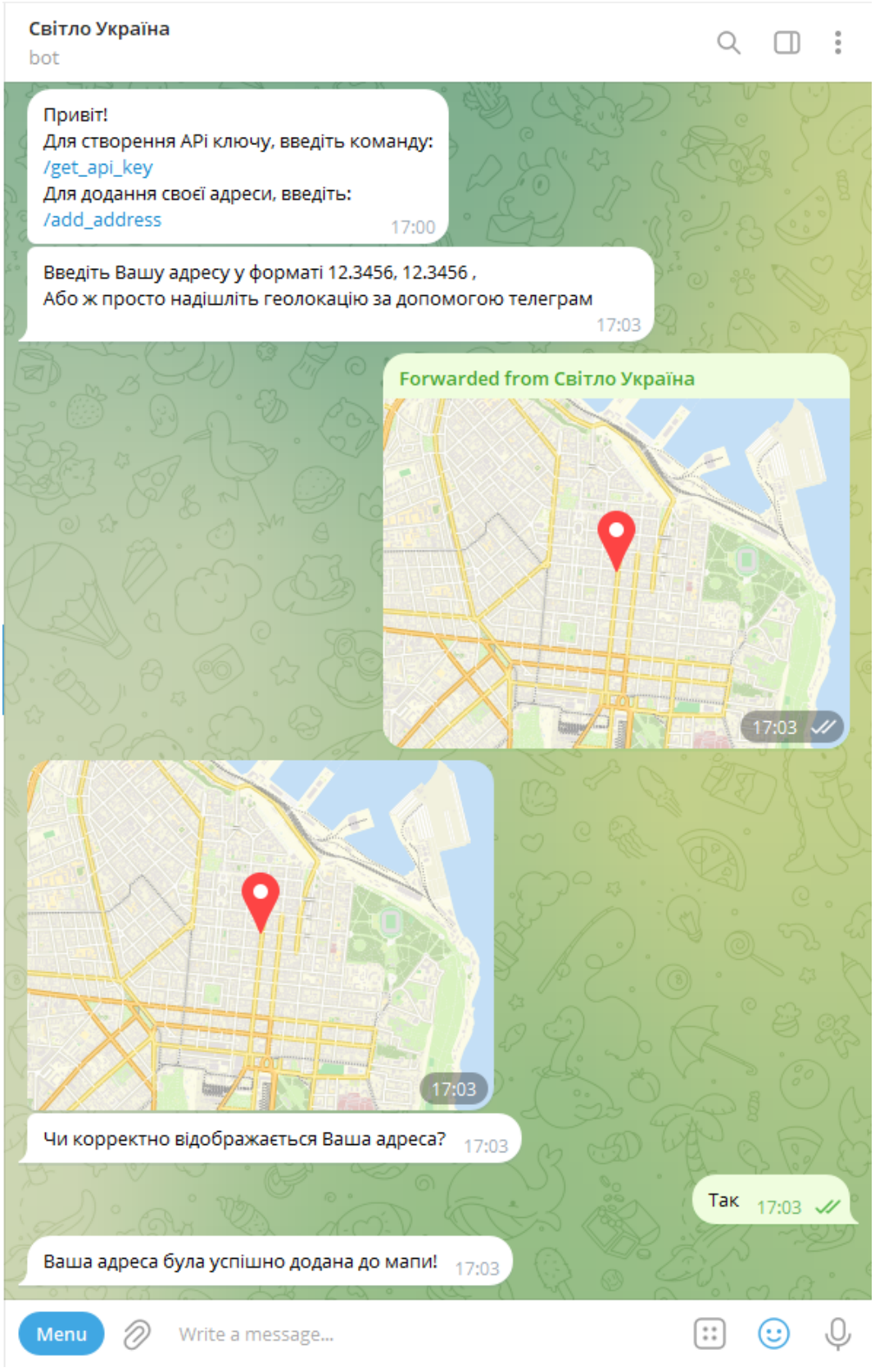
```python
async def create_api_user(session: AsyncSession, api_user: ApiUser):
    db_user = ApiUsers(tg_id=api_user.tg_id)

    async with session:
        stmt = select(ApiUsers).where(ApiUsers.tg_id == api_user.tg_id)
        result = await session.execute(stmt)

    result = result.scalar_one_or_none()

    if result is not None:
        return result
    else:
        async with session:
            session.add(db_user)
            await session.commit()
            await session.refresh(db_user)
        return db_user


1 usage  bebus
async def post_create(session: AsyncSession, post: Post):...


2 usages  bebus
async def get_all_users(session: AsyncSession, limit: int = 10, offset: int = 0):
    async with session:
        stmt = select(Posts).offset(offset).limit(limit)
        result = await session.execute(stmt)
        return result.scalars().all()


6 usages  bebus
async def check_api_key_exists(session: AsyncSession, api_key: str):
    async with session:
        stmt = select(ApiUsers).where(ApiUsers.api_key == api_key)
        result = await session.execute(stmt)
        return result.one_or_none()

    else:
        print("API key not exist!")


2 usages  bebus
async def ping_status_to_db(session: AsyncSession, electricity_status: bool, address_id: int, api_key):
    async with session.begin():
        stmt = select(ApiUsers).where(ApiUsers.api_key == api_key)
        result = await session.execute(stmt)
        user = result.scalars().first()

        if user:
            if address_id == -1:
                await session.execute(
                    update(Addresses)
                    .where(Addresses.tg_id == user.tg_id)
                    .values(last_update=datetime.datetime.now(), electricity_status=electricity_status)
                )
            else:
                await session.execute(
                    update(Addresses)
                    .where(Addresses.tg_id == user.tg_id)
                    .where(Addresses.id == address_id)
                    .values(last_update=datetime.datetime.now(), electricity_status=electricity_status)
                )
            await session.commit()
        else:
            print("No user found with the provided API key")
```

Додаток А.4 Код головного файлу телеграм бота

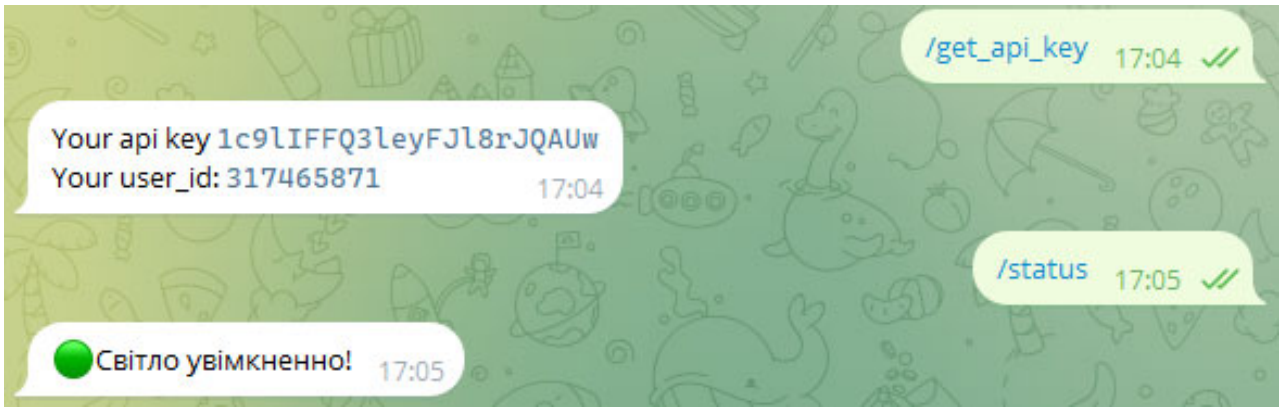
```
@dp.message(Command("get_api_key"))
async def start_message(message: Message):
    user_id = message.from_user.id
    usr = await get_token(user_id)
    await message.answer(f'Your api key <code>{usr.api_key}</code> \nYour user_id: <code>{user_id}</code>')


 bebus
@dp.message(Command("remove_my_data"))
async def delete_data(message: Message):
    user_id = message.from_user.id
    delete_status = await delete_usr(user_id)
    if delete_status:
        await message.answer(f'Your data was deleted!')
    else:
        await message.answer(f'Your data already deleted!')


 bebus
@dp.message(F.text == "💡Статус світла")
@dp.message(Command("status"))
async def status(message: Message):
    user_id = message.from_user.id
    status = await get_my_status(user_id)

    if status:
        if status.electricity_status:
            await message.answer('🟢Світло увімкненно!')
        else:
            await message.answer('⚫Світло вимкненно!(')
    else:
        await message.answer('В Вас ще немає жодної адреси!')


 bebus
@dp.message(F.location)
async def get_location(message: Message):
    await message.answer(f'{message.location.latitude}, {message.location.longitude}')


1 usage  bebus
async def main():
    async with sessionmanager.connect() as connection:
        await connection.run_sync(Base.metadata.create_all)

    await bot.delete_webhook(drop_pending_updates=True)
    await dp.start_polling(bot)


if __name__ == '__main__':
    asyncio.run(main())
```

Додаток Б. Інтерфейси телеграм боту

Додаток Б.1. Додання адреси до мапи



Додаток Б.2. Отримання API-ключа та перевірка статусу енергетики



Додаток Б.3. Меню телеграм боту