НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Агрегатор публічних екологічних даних України

Виконав: студент 4 курсу, групи ІО-02
(шифр групи)

Воловик Олександр Андрійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник ас. Ковальчук Олександр Миронович _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ст. викладач Виноградов Ю. М. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальність 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Воловика Олександра Андрійовича

1. Тема проєкту *Агрегатор публічних екологічних даних України*
керівник проєкту ас. Ковальчук Олександр Миронович,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 27.05 2024 року № 2112-с
2. Термін здачі студентом закінченого проєкту *11.06 2024 р.*
3. Вихідні дані до проєкту *технічна документація. теоретичні та статистичні дані*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Огляд існуючих систем збору та аналізу екологічних даних, огляд технологій для реалізації систем збору та аналізу екологічних даних, опис архітектури і компонентів розробленої системи збору та аналізу екологічних даних, результати тестування розробленої системи збору та аналізу екологічних даних

5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень)
схема взаємодії користувача з Android-застосунком системи (структурна схема), схема інфраструктури системи (функціональна схема), діаграма бази даних системи (принципова схема)

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Виноградов Ю. М.		

7. Дата видачі завдання 1 грудня 2023 р.

Календарний план

№ п/п	Найменування етапів диплом-ного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*30.11.2023-25.12.2023*	
2.	*Вивчення та аналіз завдання*	*26.12.2023-25.01.2024*	
3.	*Розробка архітектури та загальної структури системи*	*26.01.2024-15.02.2024*	
4.	*Розробка структур окремих підсистем*	*16.02.2024 -29.02.2024*	
5.	*Програмна реалізація системи*	*01.03.2024-30.04.2024*	
6.	*Оформлення пояснювальної записки*	*01.05.2024-31.05.2024*	
7.	*Проходження нормоконтролю та перевірка на плагіат*	*03.06.2024-07.06.2024*	
8.	*Захист*	*18.06.2024*	

Студент-дипломник _____ Олександр ВОЛОВИК
(підпис)

Керівник проєкту _____ Олександр КОВАЛЬЧУК
(підпис)

Анотація

У даній дипломній роботі виконано аналіз та розробку агрегатора публічних екологічних даних України. Проведено порівняльний аналіз існуючих систем, виявлено їх переваги та недоліки. На основі аналізу обрано технології для реалізації системи, включаючи методи збору, зберігання та аналізу даних. Розроблено архітектуру системи, яка поєднує монолітний та мікросервісний підходи для забезпечення масштабованості. Проведено тестування всіх компонентів системи, включаючи збирачі даних, API, мобільний застосунок та Telegram-бот, що підтвердило їх коректність та функціональність. Розроблена система забезпечує зручний доступ до екологічних даних для різних користувачів, сприяючи покращенню екологічного моніторингу в Україні.

Annotation

This thesis analyzes and develops an aggregator of public environmental data in Ukraine. A comparative analysis of existing systems was conducted, their advantages and disadvantages were identified. Based on the analysis, the technologies for the system implementation, including methods of data collection, storage and analysis, were selected. The system architecture is developed, which combines monolithic and microservice approaches to ensure scalability. All components of the system, including data collectors, API, mobile application, and Telegram bot, were tested, which confirmed their correctness and functionality. The developed system provides convenient access to environmental data for different users, contributing to the improvement of environmental monitoring in Ukraine.

№ рядка	Формат	Позначення	Найменування	Кількість	Примітка
1			Документація загальна		
2			Розроблена заново		
3					
4	А4	ІАЛЦ 467200.001 ОА	Агрегатор публічних	1	
5			екологічних даних України		
6			Опис альбому		
7					
8	А4	ІАЛЦ 467200.002 ТЗ	Агрегатор публічних	3	
9			екологічних даних України		
10			Технічне завдання		
11					
12	А4	ІАЛЦ 467200.003 ПЗ	Агрегатор публічних	95	
13			екологічних даних України		
14			Пояснювальна записка		
15					
16	А4	ІАЛЦ 467200.004 Е1	Агрегатор публічних	1	
17			екологічних даних України		
18			Схема взаємодії користувача з		
19			Android-застосунком системи		
20			(структурна схема)		
21					
22	А4	ІАЛЦ 467200.005 Д2	Агрегатор публічних	1	
23			екологічних даних України		
24			Схема інфраструктури системи		
25			(функціональна схема)		
26					
27	А4	ІАЛЦ 467200.006 Д3	Агрегатор публічних	1	
28			екологічних даних України		
29			Діаграма бази даних системи		
30			(принципова схема)		
31					
32	А4	ІАЛЦ 467200.007 Д4	Агрегатор публічних	12	
33			екологічних даних України		
34			Текст програмного коду		

Змн.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ 467200.001 ОА		
Розроб.		Воловик О.А.			Агрегатор публічних екологічних даних України Опис альбому	Літ.	Арк. 1 / Аркушів 1
Перевір.		Ковальчук О.М.					
Н. Контр.		Виноградов Ю.М.				КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02	
Затверд.							

Технічне завдання

Зміст

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку архітектурної концепції системи автоматизованої агрегації та аналізу геопросторових даних, а також на подальшу підтримку та вдосконалення розробленої архітектури системи.

Областю застосування цієї системи є автоматизований збір та аналіз геопросторової інформації у централізованому сховищі.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», який був затверджений факультетом "Інформатики та обчислювальної техніки" кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут імені Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка агрегатора публічних екологічних даних України.

Призначення розробки - забезпечити об'єднання та уніфікований доступ до екологічних даних з різних джерел для зручного аналізу та використання.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література по комп`ютерним системам, публікації в періодичних виданнях та публікації в мережі Інтернет щодо даної теми.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваного продукту

— Підтримка збору даних через відкриті API та вебскрейпинг.

— Відкритий доступ до зібраних даних через веб-інтерфейси, мобільні застосунки та API.

— Стійкість до збоїв і легка масштабованість системи.

— Вичерпна документація з описом функцій та інструкціями.

5.2. Вимоги до програмного забезпечення

1. Операційна система:
   — MS Windows 8, MS Windows 10 або MS Windows 11
   — macOS 10.15 (Catalina) або вище
   — Linux (сучасні дистрибутиви, наприклад, Ubuntu 20.04 або вище)
2. Java Development Kit (JDK) версії 19 або вище

5.3. Вимоги до апаратної частини

— Комп'ютер на базі процесора Intel Core i3 або еквівалентний AMD Ryzen 3 та вище

— Оперативної пам'яті не менше 1024 Мбайт

— Вільний простір жорсткого диску не менше 2048 Мбайт

6. ЕТАПИ РОЗРОБКИ

Назва етапу розробки	Термін виконання
Затвердження теми проєкту	*30.11.2023-25.12.2023*
Вивчення та аналіз завдання	*26.12.2023-25.01.2024*
Розробка архітектури та загальної структури системи	*26.01.2024-15.02.2024*
Розробка структур окремих підсистем	*16.02.2024 -29.02.2024*
Програмна реалізація системи	*01.03.2024-30.04.2024*
Оформлення пояснювальної записки	*01.05.2024-31.05.2024*

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

PM	(Particulate Matter) Тверді частинки
API	(Application Programming Interface) Прикладний програмний інтерфейс
AQI	(Air Quality Index) Індекс якості повітря
HTML	(HyperText Markup Language) Мова розмітки гіпертексту
DOM	(Document Object Model) Об'єктна модель документа
СУБД	Система управління базами даних
JSON	JavaScript Object Notation
BSON	Binary JSON
ACID	(Atomicity, Consistency, Isolation, Durability) Атомарність, узгодженість, ізольованість, довговічність
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
URI	Uniform Resource Identifier
БД	База даних
URL	Uniform Resource Locator
ORM	(Object-relational mapping) Об'єктно-реляційна проєкція
CI	(Continuous Integration) Безперервна Інтеграція
JDK	Java Development Kit
JAR	Java ARchive

ВСТУП

У сучасному світі, де екологічні питання стають невід’ємною складовою глобальної агенди, доступ до зібраних екологічних даних має вирішальне значення для прийняття обґрунтованих рішень та вдосконалення стратегій управління довкіллям. Україна, як і багато інших країн, стикається з проблемою розпорошеності та недоступності цих даних через їх різноманітність форматів та джерел.

Ця дипломна робота присвячена розробці агрегатора публічних екологічних даних України, який спрямований на об'єднання розрізнених джерел інформації в єдину систему. Основною метою цього проекту є створення уніфікованих інтерфейсів доступу до екологічних даних для різних користувачів, таких як громадські активісти, науковці, урядовці та інші зацікавлені сторони.

Крім того, розробка включає створення аналізаторів для виявлення закономірностей, аномалій та інших важливих висновків, що базуються на зібраних даних. Використання методів машинного навчання дозволить ефективно аналізувати та прогнозувати екологічні тенденції, що є ключовим аспектом цього проекту.

Розвиток цього агрегатора є кроком у напрямку покращення екологічного моніторингу та розумного прийняття рішень в Україні. Ця робота поєднує в собі технологічний прогрес з важливою суспільною місією захисту довкілля, створюючи інструмент, який може стати важливою складовою екологічної політики та практики країни.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ СИСТЕМ ЗБОРУ ТА АНАЛІЗУ ЕКОЛОГІЧНИХ ДАНИХ

1.1. Актуальність та доцільність розроблення системи збору та аналізу екологічних даних

В сучасному світі, де зростає усвідомлення екологічних проблем, необхідність у вдосконаленні систем управління екологічною інформацією стає критичною. Різноманітність викликів, що постають перед довкіллям, вимагає від нас не лише реакції, але й попередження та стратегічного планування.

Зростання темпів індустріалізації [1] та збільшення антропогенного впливу на довкілля [2] призводять до необоротних змін у природних екосистемах [3]. Відповідно, ми стикаємося з необхідністю адаптації наших методів управління довкіллям до нових реалій. Це означає, що потреба у вчасній, точній та комплексній інформації про стан довкілля стає надзвичайно важливою.

Забезпечення доступності такої інформації для всіх зацікавлених сторін — від урядових органів до громадськості — є першочерговим завданням. Тільки завдяки цій інформації можна приймати обґрунтовані рішення щодо охорони довкілля, розробляти ефективні стратегії та плани дій, а також вчасно реагувати на потенційні екологічні кризи.

В контексті нашої країни, розроблення агрегатора публічних екологічних даних має низку вагомих переваг і відповідає викликам сучасності:

1. Підвищення доступності інформації: Об'єднання розпорошених джерел даних в єдину систему дозволить забезпечити максимально широкий доступ до екологічної інформації для всіх зацікавлених сторін.
2. Сприяння прийняттю обґрунтованих рішень: Забезпечення доступу до актуальних та комплексних даних про екологічний стан дозволить урядовим органам, науковцям, громадськості та іншим зацікавленим

сторонам приймати обґрунтовані рішення щодо охорони довкілля та реагування на екологічні виклики.

3. Підвищення ефективності моніторингу: Застосування аналізаторів для виявлення закономірностей та аномалій у зібраних даних допоможе у вчасному виявленні проблемних ситуацій та прийнятті відповідних заходів.

Отже, у контексті зростаючих викликів, які перед нами ставлять екологічні проблеми, покращення систем управління екологічною інформацією стає невідкладним завданням. Вчасна та комплексна інформація є ключем до успішного збереження довкілля та підтримки сталого розвитку для майбутніх поколінь.

1.2. Огляд існуючих систем збору та аналізу екологічних даних

1.2.1 Проект “SaveEcoBot”

SaveEcoBot – “єдина в Україні екологічна система, яка поєднує дані про поточний стан довкілля, про забруднення, забруднювачів та інструменти захисту довкілля.” [4]. Один з сервісів цієї системи – карту якості повітря, продемонстровано на рис. 1.1.

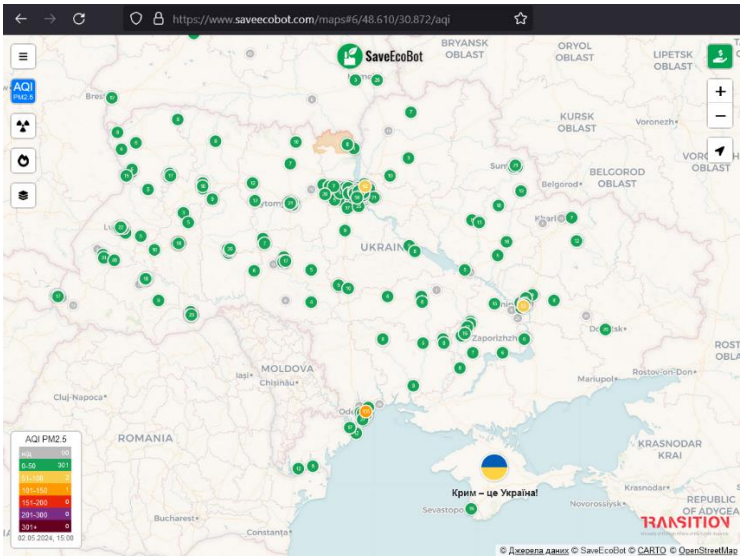


Рисунок 1.1 – Карта якості повітря SaveEcoBot

SaveEcoBot здійснює агрегацію даних з різних джерел, серед яких громадські і комерційні проєкти, органи місцевого самоврядування, університети та станції окремих жителів країни. Доступ до зібраної інформації можна здійснити декількома способами: через веб-сайт, за допомогою Telegram-боту, Viber-боту, або у машиночитному вигляді через API (проте, доступ до API обмежений, і надається лише після спілкування з представниками проєкту)

Серед типів даних, які агрегує ця система, є: температура, тиск, вологість, засмічення повітря(PM1.0, PM2.5, PM10.0), радіаційний фон та дані про пожежі.

Деяка кількість зібраних записів доступна на веб-сайті системи, проте для доступу до більшої кількості історичних даних, потрібно мати доступ до API.

На основі агрегованих даних здійснюється деяка базова аналітика, зокрема: обчислення значення AQI за допомогою значень PM та формування статистики пожеж по областях.

Частота оновлення даних є різною, в залежності від джерела (наприклад, кожну годину, кожні 3 години, кожні 24 години).

1.2.2 Система “ЛУН Місто Air”

"ЛУН Місто Air" [5] - це українська real-time система моніторингу якості повітря, яка оперує на принципах волонтерської мережі. Учасники цієї мережі встановлюють сенсори на своїй території, що дозволяє отримувати актуальні дані про забруднення повітря в реальному часі. На основі зібраних даних, система пропонує надає такі сервіси: карта на веб-сайті проєкту, Telegram-бот, Viber-бот, віджет у браузері, віджети для iPhone та Apple Watch.

Карту якості повітря, яку надає цей сервіс, показано на рис. 1.2.

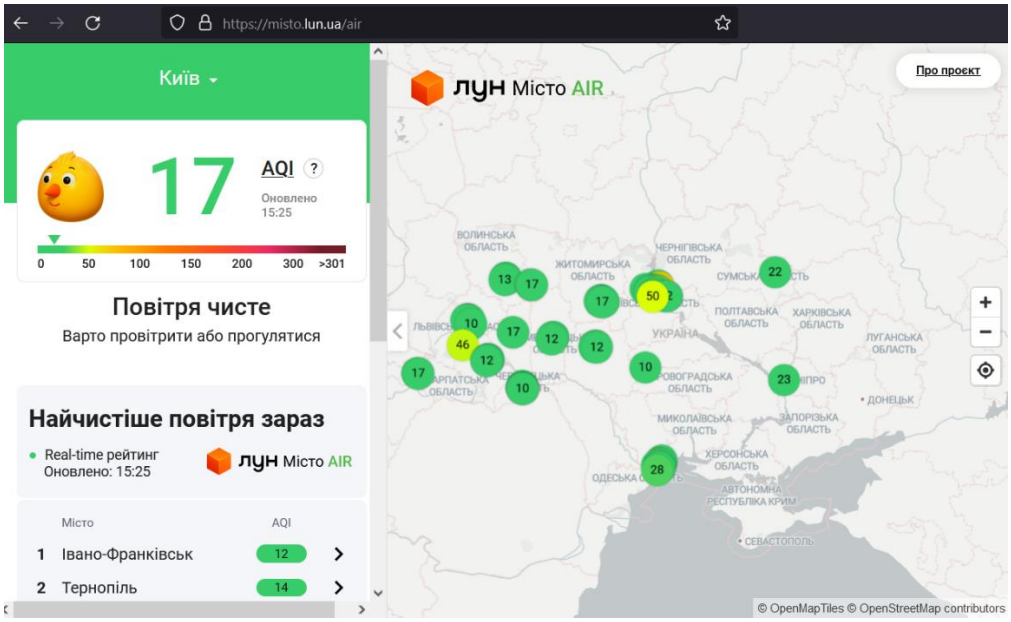


Рисунок 1.2 – Карта якості повітря ЛУН Місто Air

Сенсорні станції вимірюють значення PM1.0, PM2.5, PM10.0, температури та вологості повітря.

Система надає доступ до обчислених значень AQI у вигляді середнього значення за годину (з можливістю отримати доступ до даних за останні 48 годин), або у вигляді середнього значення за добу (з можливістю отримати доступ до даних за останні 30 днів). Немає механізмів доступу до старіших даних або конкретних виміряних значень PM, температури чи вологості.

Дані оновлюються в реальному часі.

1.2.3 Проект “World Air Quality Index”

Проект «Світовий індекс якості повітря» [6] - це некомерційний проект, започаткований у 2007 році. Його місія - сприяти підвищенню обізнаності громадян про забруднення повітря та надавати уніфіковану інформацію про якість повітря в усьому світі.

Карту індексу якості повітря, яку надає цей проект, показано на рис. 1.3.

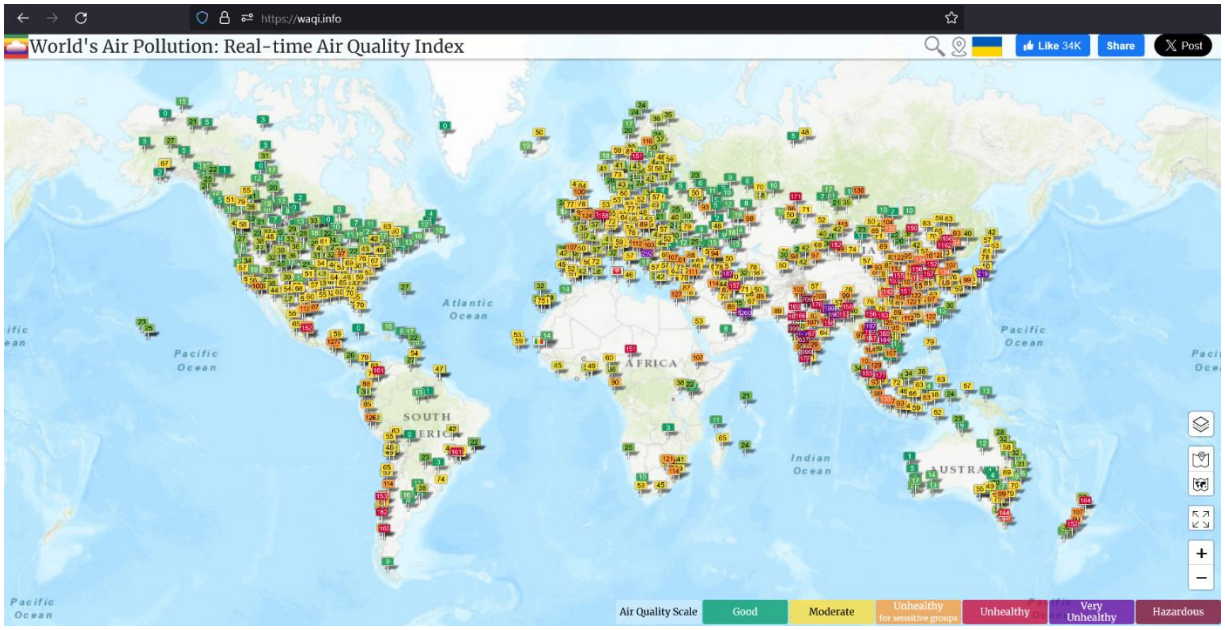


Рисунок 1.3 – Карта індексу якості повітря World Air Quality Index

Проект надає прозору інформацію про якість повітря для більш ніж 130 країн, охоплюючи понад 250 000 станцій моніторингу якості повітря у 2 000 великих містах, через два веб-сайти: aqicn.org та waqi.info.

Ця система збирає широкий спектр даних (значення PM2.5 та PM10.0, кількість озону (O_3), діоксиду азоту (NO_2), діоксиду сірки (SO_2) та оксиду вуглецю (CO) у повітрі), на основі яких обчислює значення AQI. Крім цього, система надає сервіс прогнозування індексу якості повітря у майбутньому (до 5 діб). Значення індексу якості повітря доступні на веб-сайті та через API. Також можливо зробити запит на отримання детальних даних за певний період.

Історичні дані доступні починаючи з 2012 року.

Дані на World Air Quality Index публікуються в режимі реального часу, а тому на момент публікації не є валідованими. Для підвищення якості використовується набір алгоритмів штучного інтелекту в режимі реального часу для виявлення аномальних даних та автоматичного «вимкнення» даних, отриманих від несправних станцій.

1.2.4 Порівняльна таблиця розглянутих систем збору та аналізу екологічних даних

Таблиця 1.1 – Порівняльна таблиця розглянутих систем збору та аналізу екологічних даних

Критерій	SaveEcoBot	ЛУН Місто Air	World Air Quality Index
Типи даних, які збираються	Температура, тиск, вологість, засмічення повітря (PM1.0, PM2.5, PM10.0), радіаційний фон, пожежі.	Температура, вологість повітря, засмічення повітря (PM1.0, PM2.5, PM10.0).	Засмічення повітря (PM1.0, PM2.5, PM10.0, O_3, NO_2, SO_2 CO).
Джерела даних	Громадські і комерційні проєкти, органи місцевого самоврядування, університети та станції окремих жителів країни.	Волонтерська мережа.	Громадські і комерційні проєкти.
Способи представлення даних	Веб-сайт, Telegram-бот, Viber-бот, API(доступ обмежений).	Веб-сайт, Telegram-бот, Viber-бот, віджет у браузері, віджети для iPhone та Apple Watch.	Веб-сайт, API.

Кінець таблиці 1.1

Критерій	SaveEcoBot	ЛУН Місто Air	World Air Quality Index
Доступність історичних даних	Обмежена кількість зібраних записів доступна на веб-сайті, а для доступу до більшої кількості історичних даних необхідно отримувати доступ до API.	Середні значення AQI за годину (за останні 48 годин), середні значення AQI за доби (за останні 30 діб) доступні на веб-сайті.	Значення індексу якості повітря доступні на веб-сайті та через API. Для отримання детальних даних за певний період необхідно робити запит через форму на веб-сайті.
Частота оновлення даних	В залежності від джерела: кожну годину, кожні 3 години, кожні 24 години.	В реальному часі.	В реальному часі.
Обробка даних	Обчислення значень AQI, групування за населеними пунктами, формування статистики пожеж по областях.	Обчислення значень AQI, групування за населеними пунктами.	Обчислення значень AQI, групування за країнами, прогнозування AQI.

1.3. Проблематика розроблення систем збору та аналізу екологічних даних

1.3.1 Огляд проблеми стандартизації даних

Проблема відсутності стандартизації даних є спільною для багатьох систем агрегації. Джерела інформації про стан довкілля дуже різноманітні: датчики на метеостанціях, супутникові знімки, датчики забруднення повітря та води, дані громадських ініціатив. Як правило, кожне джерело надає дані у власному форматі. Це є проблемою для систем-агрегаторів, оскільки дані з джерел мають бути приведені до єдиного, стандартизованого вигляду, перш ніж їх можна буде використовувати.

Відсутність єдиних стандартів для збору, обробки та представлення даних у сфері екології ускладнює інтеграцію між різними джерелами інформації та може призводити до неточностей у виявленні тенденцій та аномалій.

Існують декілька способів подолати проблему нестачі стандартизації даних про стан довкілля:

1. Розробка та впровадження стандартів:

- Створення та прийняття на державному рівні стандартів для збору, обробки та представлення даних про довкілля.
- Ці стандарти можуть базуватися на міжнародних стандартах, таких як ISO 14001 [7] або Open Geospatial Consortium Standards [8].
- Важливо, щоб до розробки стандартів залучалися всі зацікавлені сторони, включаючи державні органи, наукові кола, приватний сектор та громадські організації.

2. Використання існуючих протоколів та форматів даних:

- Замість того, щоб розробляти нові стандарти з нуля, можна використовувати вже існуючі протоколи та формати даних, такі як CF Conventions [9] або SensorML [10].
- Це може допомогти прискорити процес стандартизації та зробити дані більш доступними для широкого кола користувачів.

3. Створення централізованих сховищ даних:

- Створення централізованих сховищ даних, де користувачі можуть знаходити та отримувати доступ до стандартизованих даних про довкілля.
- Це може полегшити інтеграцію даних з різних джерел та проведення аналізу.

Важливо зазначити, що не існує єдиного універсального рішення проблеми нестачі стандартизації даних. Найкращий підхід буде залежати від конкретних потреб та контексту. Однак, шляхом впровадження вищезазначених заходів, можна значно покращити якість та доступність даних про стан довкілля.

1.3.2 Огляд проблеми надійності даних

Проблема надійності та точності даних є ключовою в системах збору і аналізу екологічних даних. Необхідність надійних та точних даних виникає з важливості використання цих даних для прийняття рішень у сфері охорони довкілля, прогнозування екологічних тенденцій та оцінки ефективності заходів.

Проблеми, пов'язані з надійністю та точністю даних, можуть включати:

1. Калібрування та обслуговування датчиків: Для забезпечення точних вимірювань потрібно регулярно калібрувати та обслуговувати датчики, що використовуються для збору даних. Несправні датчики можуть призводити до неточностей у вимірюваннях.
2. Якість обладнання та методів вимірювання: Низька якість обладнання або застосування неоптимальних методів може призвести до неточностей та спотворень у даних.
3. Вплив умов навколишнього середовища: На точність вимірювань можуть впливати умови навколишнього середовища, такі як температура, вологість, атмосферний тиск тощо

В залежності від джерела спотворень даних, можна використовувати різні методи подолання описаної проблеми:

Калібрування та обслуговування датчиків:

- Регулярно калібрувати та обслуговувати датчики згідно з інструкціями виробника.
- Вести записи про калібрування та обслуговування датчиків.
- Замінювати датчики, які вийшли з ладу або не відповідають технічним характеристикам.

Використання якісного обладнання та методів вимірювання:

- Використовувати датчики та інше обладнання від авторитетних виробників.
- Застосовувати перевірені та стандартизовані методи вимірювання.
- Регулярно перевіряти обладнання на наявність пошкоджень або несправностей.

Моніторинг умов навколишнього середовища:

- Здійснювати моніторинг умов навколишнього середовища, таких як температура, вологість, атмосферний тиск тощо.
- Використовувати дані про умови навколишнього середовища для коректування вимірювань датчиків.
- Встановлювати датчики в місцях, де вплив умов навколишнього середовища на точність є мінімальним.

Застосування методів контролю якості даних:

- Використовувати методи контролю якості даних для виявлення та виправлення помилок у даних. Це може включати перевірку на наявність пропущених значень, дублікатів, аномальних значень та інших помилок.
- Застосовувати методи статистичного аналізу для оцінки якості даних.

Забезпечення прозорості та документування:

- Документувати методи збору, обробки та аналізу даних.
- Забезпечити прозорість методів контролю якості даних.
- Робити дані та документацію доступними для зацікавлених сторін.

1.3.3 Огляд проблеми доступності даних

Ще однією ключовою проблемою в системах-агрегаторах є доступність даних. Оскільки системи-агрегатори не генерують первинні дані самостійно, а отримують їх з різних джерел, то їх функціонування повністю залежить від доступності цих джерел. Схема функціонування агрегатора наведена на рисунку 1.4.

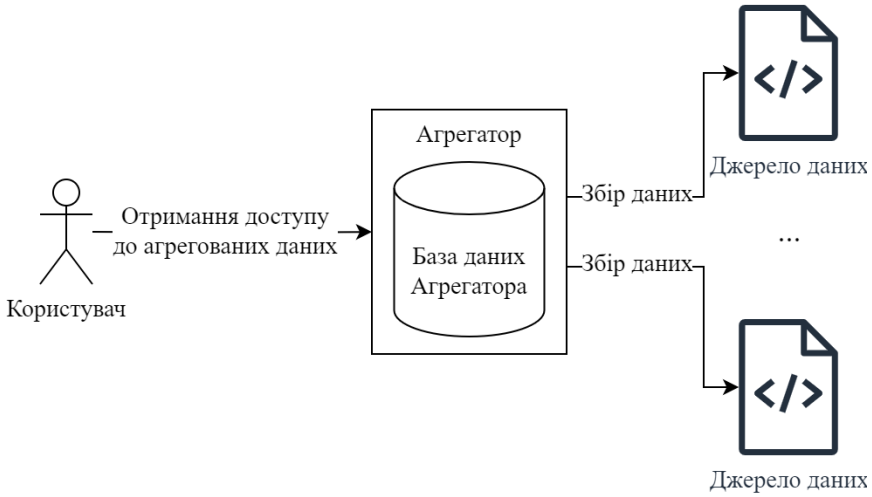


Рисунок 1.4 – Схема функціонування агрегатора даних

1.3.4 Огляд проблеми постійного зростання обсягів даних

Системи-агрегатори відкритих екологічних даних стикаються з рядом проблем з масштабованістю, пов'язаних з постійним зростанням обсягу даних. Основні проблеми з масштабованістю включають:

1. Пропускна здатність мережі: Зростання обсягу даних, які передаються, може призвести до перевантаження мережі і, в результаті, проблем з доступом до системи.
2. Зберігання даних: Зберігання великих обсягів даних може бути дорогим та складним для організації.
3. Обробка даних: Обробка великих обсягів даних може бути ресурсомісткою та потребувати значних обчислювальних потужностей.

Існує декілька способів подолати проблеми з масштабованістю:

1. Горизонтальне масштабування: Додавання нових серверів до системи може допомогти розподілити навантаження та покращити продуктивність.
2. Використання хмарних технологій: Хмарні сервіси можуть надати гнучкість та масштабованість, необхідні для обробки великих обсягів даних.
3. Оптимізація запитів: Оптимізація запитів до бази даних може допомогти покращити їхню продуктивність.
4. Використання кешування: Кешування результатів запитів може допомогти зменшити навантаження на базу даних та покращити час відгуку.
5. Використання розподілених систем зберігання даних: Розподілені системи зберігання даних, такі як Hadoop [11] або Cassandra [12], можуть допомогти масштабувати зберігання даних.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі даної дипломної роботи було досліджено та розглянуто питання актуальності розробки агрегатора публічних даних про екологічний стан навколишнього середовища України. Розглянуто існуючі системи з аналогічним чи подібним функціоналом. Результати дослідження наведено в таблиці 1.1. Також виконано опис основних проблем, з якими можливо стикнутися при розробці системи такого класу, та можливих способів їх подолання.

Отже, розроблювана система має: забезпечувати зручні та зрозумілі інтерфейси доступу до агрегованих даних, мати механізми оцінки надійності та точності зібраних даних, бути легкою у розширенні (додаванні нових типів, джерел даних), стабільно функціонувати в умовах постійного зростання обсягів даних.

РОЗДІЛ 2
ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ СИСТЕМ ЗБОРУ ТА АНАЛІЗУ ЕКОЛОГІЧНИХ ДАНИХ

2.1 Огляд методів збору даних з існуючих джерел

Розглянемо методи та інструменти, які використовуються для збору екологічних даних з існуючих джерел.

Одним із основних методів доступу до даних є використання відкритих API. Цей метод дозволяє отримати доступ до різноманітних джерел даних, які можуть бути використані для аналізу екологічних параметрів. Такі API часто надають доступ до різноманітних типів даних, включаючи дані про якість повітря, рівень забруднення води, температурні та погодні дані. Проте, не всі сервіси надають доступ до своїх даних через API.

В тих випадках, коли сервіси не надають способів отримати дані через API, можливо використати вебскрейпинг – процес перетворення у структуровані дані інформації з вебсторінок, які призначені для перегляду людиною за допомогою браузера. Як правило, робота цього методу полягає в аналізі та вилученні необхідний даних з HTML або DOM-дерев сторінок.

2.1.1 Використання відкритих API

Відкриті API – це програмні інтерфейси, які надають доступ до даних та функціональності різних систем та сервісів.

Особливості:

- Широкий спектр даних. Доступ до даних з різних джерел, таких як державні органи, наукові установи, громадські проєкти тощо.
- Актуальність. Дані оновлюються в режимі реального часу або з певною періодичністю, близькою до реального часу.
- Прозорість. Документація API зазвичай доступна публічно, що дає можливість перевірити якість та надійність даних.

– Стандартизація. API зазвичай базуються на стандартах, що полегшує їх інтеграцію з іншими системами.

Переваги:

– Швидкість збору даних. Завдяки доступу до даних з різних джерел, можна швидко збирати великі обсяги інформації.
– Зниження витрат. Немає потреби в купівель та обслуговуванні власних датчиків та інфраструктури.
– Масштабованість. Можна легко розширити систему за рахунок нових джерел даних.
– Гнучкість. Можна гнучко налаштувати систему збору даних відповідно до потреб, використовуючи фільтрацію, сортування та інші інструменти.
– Доступність. Відкриті API доступні безкоштовно або за доступною ціною.

Недоліки:

– Якість даних. Не всі API гарантують високу якість та надійність даних.
– Зміна API. Розробники API можуть змінювати інтерфейс та формат даних, що може потребувати додаткових зусиль з адаптації системи.
– Залежність від сторонніх сервісів. Система збору даних стає залежною від працездатності сторонніх сервісів.

Важливо зазначити:

– Не всі API мають однаковий рівень якості та надійності.
– Деякі API можуть потребувати реєстрації або авторизації.

2.1.2 Вебскрейпинг

Вебскрейпинг - це процес автоматичного вилучення даних з веб-сайтів. Він може бути ефективним методом збору екологічних даних, які публікуються на різних веб-ресурсах, таких як державні портали, сайти екологічних організацій, новинні ресурси тощо.

Особливості:

– Динамічний збір даних. Можливість збирати дані з постійно оновлюваних веб-сайтів.
– Збір структурованих та неструктурованих даних. Можливість вилучати як структуровані дані (таблиці, списки), так і неструктуровані дані (текст, зображення).
– Автоматизація процесу. Збір даних може бути автоматизований для регулярного оновлення.

Переваги:

– Широкий спектр даних. Можливість збирати дані з практично будь-якого джерела.
– Актуальність. Дані можуть збиратися в режимі реального часу.
– Гнучкість. Можливість налаштувати скрейпер для збору конкретних даних.
– Зниження витрат. Може бути більш економним методом збору даних, ніж розгортання власних датчиків.

Недоліки:

– Складність розробки. Розробка ефективного веб-скрепера може потребувати знань з програмування та веб-технологій.
– Етичні міркування. Важливо дотримуватися правил та умов використання веб-сайтів під час скрейпингу.
– Зміна структури сайту. Зміни структури або формату веб-сайту можуть призвести до поломки скрейпера.
– Блокування. Деякі веб-сайти можуть блокувати автоматизовані запити, що ускладнює збір даних.

Важливо зазначити:

– Перед початком веб-скрейпингу необхідно ознайомитися з правилами та умовами використання цільового веб-сайту.
– Слід використовувати етичні методи скрейпингу, щоб не перевантажувати сервери та не порушувати роботу веб-сайтів.

– Необхідно бути готовим до адаптації скрейпера у випадку змін структури веб-сайту.

Вебскрейпинг може бути потужним інструментом для збору екологічних даних з веб-сайтів, але важливо використовувати його відповідально та бути готовим до технічних складнощів.

2.2 Огляд систем зберігання та обробки даних

2.2.1 СУБД PostgreSQL

PostgreSQL, як система управління базами даних (СУБД), є однією з найпоширеніших та широко використовуваних у світі, відповідно до статистики на рис. 2.1. Вона відома своєю високою ефективністю, надійністю та розширюваністю, що робить її привабливим вибором для зберігання та обробки різноманітних типів даних, включаючи екологічні дані. PostgreSQL заснована на об'єктно-реляційній архітектурі, що дозволяє зручно організовувати та управляти складними структурами даних.

420 systems in ranking, May 2024

Rank			DBMS	Database Model	Score		
May 2024	Apr 2024	May 2023			May 2024	Apr 2024	May 2023
1.	1.	1.	Oracle	Relational, Multi-model	1236.29	+2.02	+3.66
2.	2.	2.	MySQL	Relational, Multi-model	1083.74	-3.99	-88.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	824.29	-5.50	-95.80
4.	4.	4.	PostgreSQL	Relational, Multi-model	645.54	+0.49	+27.64
5.	5.	5.	MongoDB	Document, Multi-model	421.65	-2.31	-14.96
6.	6.	6.	Redis	Key-value, Multi-model	157.80	+1.36	-10.33
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	135.35	+0.57	-6.28
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	128.46	+0.97	-14.56
9.	9.	↑ 11.	Snowflake	Relational	121.33	-1.87	+9.61
10.	10.	↓ 9.	SQLite	Relational	114.32	-1.69	-19.54
11.	11.	↓ 10.	Microsoft Access	Relational	104.92	-0.49	-26.26
12.	12.	12.	Cassandra	Wide column, Multi-model	101.89	-1.97	-9.25

Рисунок 2.1 – Статистика популярності СУБД [13]

Однією з ключових переваг PostgreSQL є її можливість ефективно опрацьовувати великі обсяги даних. Це досягається завдяки оптимізованим механізмам обробки запитів, які дозволяють забезпечити високу швидкодію роботи з базами даних навіть при значних обсягах інформації. Більш того, PostgreSQL

підтримує розподілені системи, що дозволяє масштабувати роботу з даними для великих обсягів інформації та забезпечує високу доступність системи.

У зв'язку зі своєю об'єктно-реляційною природою, PostgreSQL надає різноманітні можливості для роботи з різними типами даних, включаючи структуровані дані та географічні дані. Це робить її досить гнучкою для зберігання та обробки екологічних даних різного характеру. Також важливо відзначити, що PostgreSQL підтримує транзакції, що забезпечує цілісність даних та робить її популярним вибором для додатків, які потребують високого рівня надійності.

Загалом, PostgreSQL є потужним інструментом для зберігання та обробки екологічних даних, який відповідає вимогам надійності, ефективності та розширюваності. Її відкритий вихідний код також сприяє поширенню та розвитку цієї технології у науковому та промисловому середовищах.

2.2.2 СУБД Oracle Database

Oracle Database - це комерційна реляційна система керування базами даних (СУБД), розроблена компанією Oracle Corporation [14]. Вона є однією з найпопулярніших СУБД у світі, яку використовують багато великих компаній та організацій у різних галузях [13]. Oracle Database відома своєю високою продуктивністю, масштабованістю, надійністю та безпечністю.

Переваги використання Oracle Database:

- Широкий спектр функцій. Oracle Database пропонує широкий спектр функцій для роботи з даними, включаючи складні запити, тригери, використання процедурної мови PL/SQL.
- Висока продуктивність. Oracle Database здатна обробляти великі обсяги даних з високою швидкістю, що робить її ідеальною для ресурсоємних застосунків.

- Масштабованість. Oracle Database може масштабуватися для задоволення потреб організацій будь-якого розміру, від невеликих робочих груп до великих корпоративних систем.
- Надійність. Oracle Database має репутацію надійної та безпечної СУБД, що робить її цінним вибором для критичних для бізнесу застосунків.
- Безпека. Oracle Database пропонує широкий спектр функцій безпеки для захисту конфіденційних даних.
- Підтримка. Oracle Corporation пропонує цілодобову підтримку користувачів Oracle Database.

Недоліки використання Oracle Database:

- Вартість. Oracle Database - це комерційна СУБД, яка може бути дорогою для деяких організацій, особливо для малого та середнього бізнесу.
- Складність. Oracle Database - це складна СУБД, яка потребує знань та досвіду для адміністрування та використання.
- Залежність від постачальника. Oracle Database - це продукт компанії Oracle Corporation, тому організації залежать від Oracle у питаннях підтримки, оновлень та вдосконалення продукту.

2.2.3 СУБД MongoDB

MongoDB - це відома NoSQL база даних, що використовує модель даних BSON [15], роблячи її гнучкою та масштабованою. Її популярність стрімко зростає завдяки простоті використання, високій продуктивності та зручності роботи з великими обсягами даних.

Основні характеристики MongoDB:

- Документо-орієнтована модель. MongoDB зберігає дані у форматі BSON, де кожен запис є документом, а не рядком у таблиці.
- Гнучкість. MongoDB не має жорсткої схеми, що дозволяє їй легко адаптуватися до мінливих потреб та нових типів даних.

Переваги використання MongoDB:

– Масштабованість. MongoDB може горизонтально масштабуватися на декілька серверів, що робить її здатною обробляти великі обсяги даних з високою швидкістю.
– Простота використання. MongoDB має інтуїтивно зрозумілий інтерфейс та API, що робить її доступною для розробників з різним рівнем досвіду.
– Відкритий код. MongoDB – це вільно доступна СУБД з відкритим кодом.
– Активне співтовариство. MongoDB має велике та активне співтовариство розробників та користувачів, що забезпечує доступ до численних ресурсів та підтримку.

Недоліки використання MongoDB:

– Необхідність розуміння BSON. Для роботи з MongoDB потрібне ознайомлення та розуміння формату BSON, що може бути складним для деяких користувачів.
– Менша зрілість. MongoDB - це порівняно нова технологія, тому вона може мати меншу зрілість та підтримку, ніж деякі інші СУБД.

2.2.4 СУБД Apache Cassandra

Apache Cassandra – це розподілена NoSQL база даних з відкритим кодом, яка використовується для зберігання та обробки великих обсягів даних з високою доступністю та масштабованістю.

Основні характеристики Apache Cassandra:

– Розподілена архітектура. Дані в Cassandra розподілені на декілька вузлів, що робить її стійкою до збоїв окремих вузлів та здатною обробляти великі обсяги даних.
– Відсутність єдиної точки відмови. Cassandra не має централізованого сервера, що робить її стійкою до збоїв будь-якого окремого вузла.

Переваги використання Apache Cassandra:

– Висока доступність. Cassandra гарантує доступність даних навіть у разі збою одного або декількох вузлів, що робить її цінним інструментом для критичних для бізнесу застосунків.
– Масштабованість. Завдяки власній структурі, Cassandra може горизонтально масштабуватися, що надає їй здатність обробляти зростаючі обсяги даних.
– Висока продуктивність. Cassandra відома своєю швидкістю обробки запитів, що робить її цінним інструментом для динамічних веб-додатків та аналітики даних.
– Відкритий код. Cassandra – це вільно доступна програма з відкритим кодом.

Недоліки використання Apache Cassandra:

– Складність. Cassandra має складну архітектуру, що може потребувати знань та досвіду для адміністрування та використання.
– Необхідність планування. Cassandra потребує ретельного планування та налаштування для забезпечення оптимальної продуктивності та масштабованості.
– Обмежена підтримка ACID. За задумом, Cassandra цінує доступність і відмовостійкість більше, ніж узгодженість(consistency) [16].

2.2.5 Пошуково-аналітична система Elasticsearch

Elasticsearch [17] – це розподілена пошукова система та NoSQL платформа аналітики даних, яка використовується для пошуку, аналізу та візуалізації великих обсягів даних.

Основні характеристики Elasticsearch:

– Розподілена архітектура. “ Elasticsearch - це розподілене сховище документів. Замість того, щоб зберігати інформацію у вигляді рядків стовпчикових даних, Elasticsearch зберігає складні структури даних, які були серіалізовані у вигляді JSON-документів. Якщо у вас є кілька вузлів Elasticsearch

у кластері, збережені документи розподіляються по всьому кластеру, і до них можна отримати доступ з будь-якого вузла.” [17]

- Масштабованість. Elasticsearch може горизонтально масштабуватися на декілька вузлів, що робить її здатною обробляти зростаючі обсяги даних.
- Переваги використання Elasticsearch:
- Потужний пошук та аналітика. Elasticsearch пропонує широкий спектр можливостей пошуку, включаючи повнотекстовий пошук, пошук за фразами та геопросторовий пошук.
- Візуалізація даних. Elasticsearch пропонує різні інструменти для візуалізації даних, такі як гістограми, діаграми та карти.
- Відкритий код. Elasticsearch – це вільно доступна програма з відкритим кодом.

Недоліки використання Elasticsearch:

- Складність. Elasticsearch має складну архітектуру, що може потребувати знань та досвіду для адміністрування та використання.
- Обмежені гарантії ACID. Elasticsearch не підтримує ACID-транзакції для змін, що стосуються декількох документів, але зміни в межах окремих документів є ACID-транзакціями [18].

2.3 Огляд аналітичних методів та інструментів

2.3.1 Методи статистичного аналізу

Методи статистичного аналізу в контексті екологічних даних використовуються для виявлення залежностей, паттернів та важливих відмінностей між різними наборами даних. Вони дозволяють вченим і дослідникам отримати об'єктивну інтерпретацію екологічної інформації і висунути науково обґрунтовані висновки.

Один з основних методів статистичного аналізу - кореляційний аналіз, дозволяє встановлювати взаємозв'язки між різними змінними.

Кореляційний аналіз - це статистичний метод, що використовується для визначення ступеня взаємозв'язку між двома або більше змінними. Цей метод дозволяє встановлювати, наскільки зміна однієї змінної впливає на зміну іншої. Основна мета кореляційного аналізу полягає в тому, щоб з'ясувати, чи існує статистично значущий зв'язок між змінними, тобто чи є вони спільно залежними.

У контексті екологічних даних, кореляційний аналіз може бути застосований для вивчення взаємозв'язків між різними екологічними факторами. Наприклад, він може допомогти визначити, чи є зв'язок між рівнем забруднення повітря і кількістю випадків захворювань на дихальні захворювання у певній місцевості. Також може бути досліджено вплив різних факторів, таких як кліматичні умови або географічне розташування, на біорізноманіття або стан екосистеми.

У кореляційному аналізі використовується коефіцієнт кореляції, такий як коефіцієнт Пірсона [19], Спірмена [20] або Кендалла [21], для вимірювання сили та напрямку зв'язку між змінними. Коефіцієнт кореляції може бути позитивним, якщо зміна однієї змінної супроводжується збільшенням чи зменшенням іншої змінної, або негативним, якщо зміна однієї змінної відбувається у протилежному напрямку до зміни іншої змінної. Приклади діаграм розсіювання (точкових діаграм) для різних наборів даних з різними коефіцієнтами кореляції наведені на рисунку 2.2.

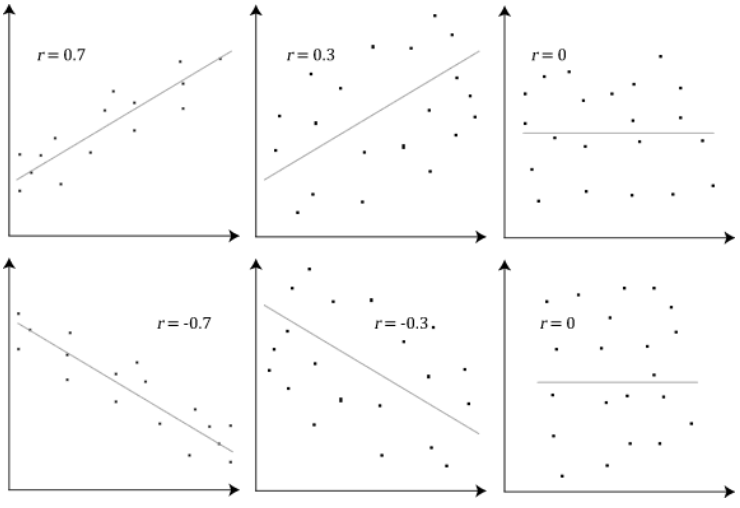


Рисунок 2.2

Однак важливо розуміти, що кореляція не завжди означає причинно-наслідковий зв'язок між змінними. Вона лише показує, наскільки сильно змінні залежать одна від одної, але не вказує на причинно-наслідкові зв'язки. Тому для отримання повного розуміння екологічних процесів і взаємозв'язків між ними часто потрібні додаткові дослідження та аналіз.

Інший важливий метод - регресійний аналіз, який дозволяє прогнозувати значення однієї змінної на основі значень інших змінних.

Регресійний аналіз - це статистичний метод, який використовується для вивчення взаємозв'язку між однією залежною змінною (змінною відгуку) та однією або декількома незалежними змінними (пояснюючими змінними). Основна мета регресійного аналізу - встановлення функціонального зв'язку між змінними, тобто прогнозування значень змінної відгуку на основі значень пояснюючих змінних.

У контексті екологічних даних, регресійний аналіз може бути застосований для розуміння та передбачення різноманітних екологічних явищ. Наприклад, він може допомогти визначити, які фактори впливають на забруднення повітря в певному регіоні, як зміни клімату впливають на розподіл видів або які фактори впливають на рівень водності у водоймах.

Регресійний аналіз може бути лінійним або нелінійним, в залежності від форми функції, яка використовується для моделювання взаємозв'язку між змінними. Лінійний регресійний аналіз передбачає лінійний зв'язок між змінними, тоді як нелінійний регресійний аналіз може враховувати складніші форми зв'язку, такі як квадратичні, експоненціальні або логарифмічні. Приклад лінійної регресії наведено на рисунку 2.3, а нелінійної – на рисунку 2.4.

Одним з основних етапів регресійного аналізу є визначення коефіцієнтів регресії, які вказують на силу та напрямок зв'язку між змінними. Визначення цих коефіцієнтів виконується за допомогою статистичних методів, таких як метод найменших квадратів.

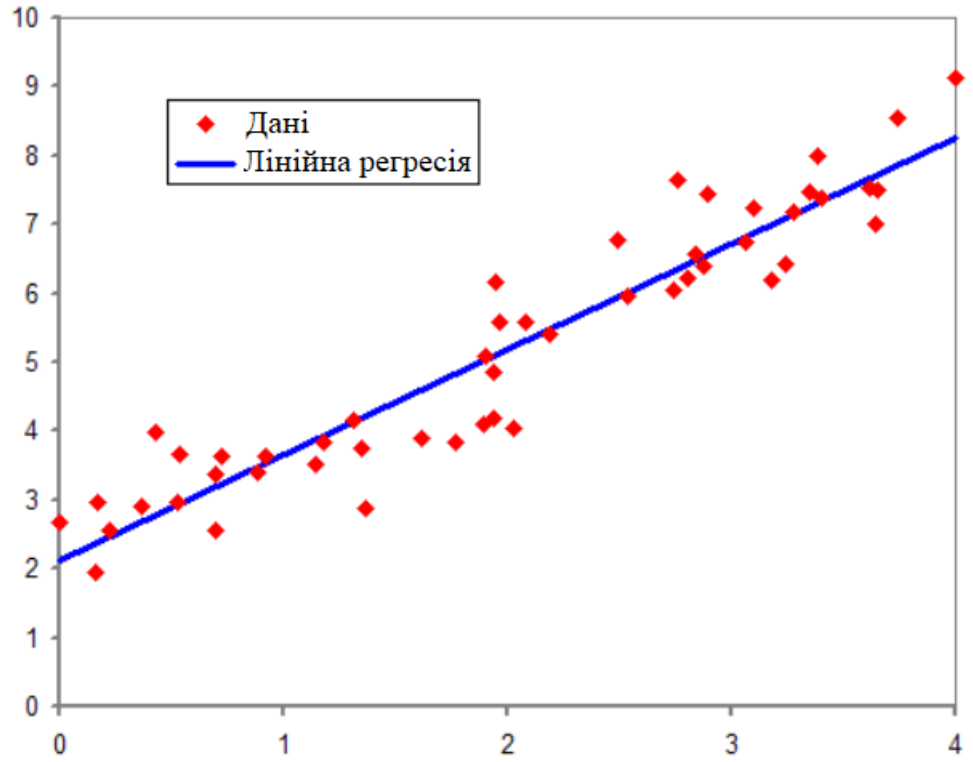


Рисунок 2.3 - Лінійна регресія для 50 випадкових точок нормально розподілених навколо прямої y=1.5x+2 (не показано)

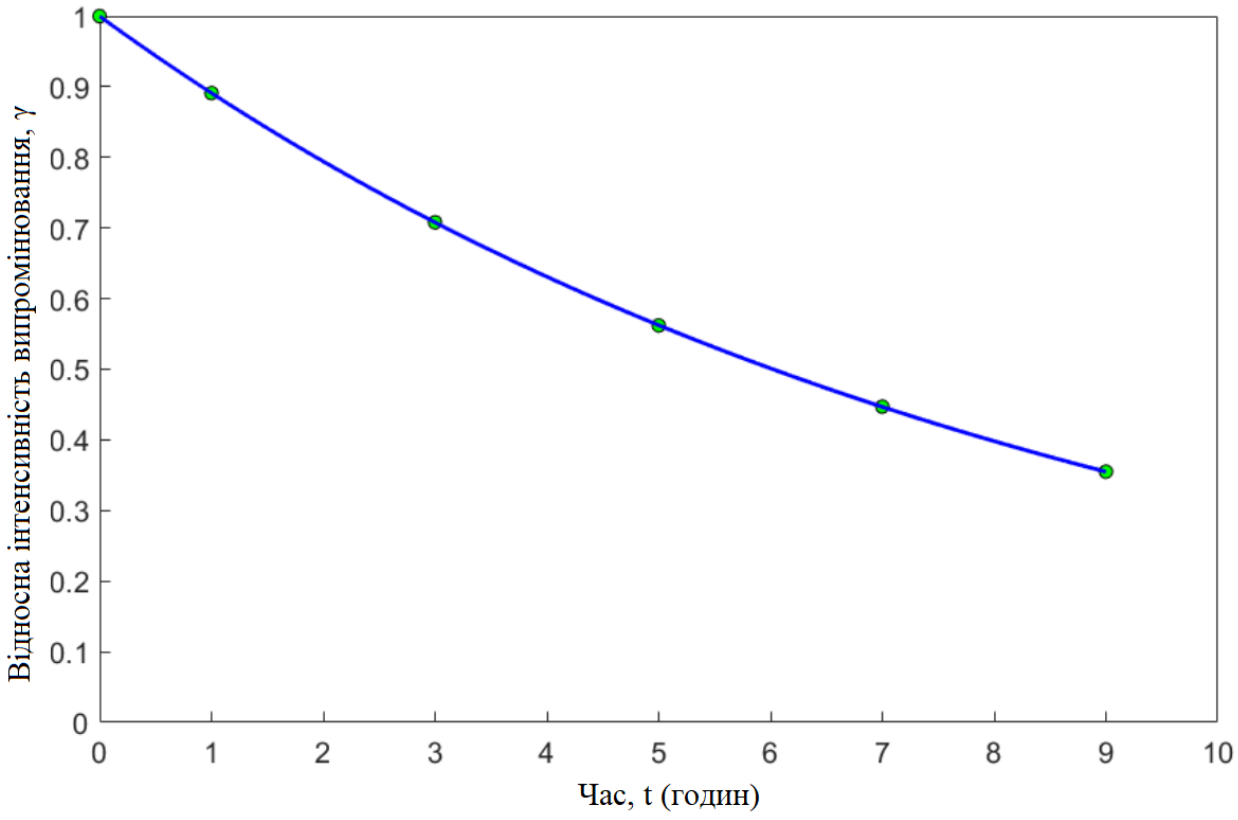


Рисунок 2.4 - Відносна інтенсивність випромінювання Технецію-99 як функція часу з використанням експоненціальної регресії [22]

Результати регресійного аналізу можуть бути використані для прогнозування значень змінної відгуку на основі значень пояснюючих змінних, а також для вивчення впливу різних факторів на екологічні процеси та явища.

Додатково до цього, методи аналізу дисперсії, факторного аналізу, класифікації та кластеризації також використовуються для виявлення взаємозв'язків і патернів у складних наборах екологічних даних. Вони допомагають ідентифікувати фактори, які впливають на зміни в екосистемах та визначити групи схожих за характеристиками об'єктів чи подій.

2.3.2 Методи машинного навчання

Методи машинного навчання - це клас алгоритмів та моделей, які дозволяють комп'ютерам "навчитися" робити прогнози або виконувати завдання на основі великої кількості даних без явного програмування. Основна ідея полягає в тому, що модель вивчається з даних, а не програмується напряму, що дозволяє виявляти складні зв'язки та виконувати завдання, що раніше були складні або неможливі для програмування за допомогою традиційних методів.

У контексті аналізу екологічних даних, методи машинного навчання можуть бути використані для різноманітних завдань, зокрема, таких як прогнозування забруднення повітря [23]. Деякі з найпоширеніших методів машинного навчання включають:

1. Навчання з учителем (Supervised Learning): Цей метод передбачає наявність маркованих даних, де кожен приклад має відому мітку або відповідь. Модель вивчається з цих даних та намагається передбачити мітку для нових прикладів. Прикладами алгоритмів навчання з учителем є лінійна регресія, метод опорних векторів (SVM) та нейронні мережі.
2. Навчання без учителя (Unsupervised Learning): У цьому випадку модель працює з немаркованими даними, тобто даними без відомих відповідей. Метою є виявлення прихованих структур або патернів у даних. Прикладами алгоритмів навчання без учителя є кластерний аналіз, метод головних компонент (PCA) та асоціативні правила.

Методи машинного навчання стають все популярнішими в аналізі екологічних даних, оскільки вони дозволяють виявляти складні зв'язки та робити прогнози на основі великих обсягів даних.

2.4 Огляд архітектурних підходів

Розглянемо два основних архітектурних підходи, які можуть бути використані для реалізації систем збору та аналізу екологічних даних: монолітна архітектура та мікросервісна архітектура. Кожен з цих підходів має свої переваги та обмеження, і вибір між ними може вплинути на проектування, розгортання та управління системою.

2.4.1 Монолітна архітектура

Монолітна архітектура - це традиційний підхід до розробки програмного забезпечення, де весь функціонал системи розміщений в одному цілісному додатку або сервісі. У такій системі всі компоненти, такі як інтерфейс користувача, логіка бізнес-процесів та база даних, об'єднані в одному монолітному додатку.

Зазвичай монолітні додатки мають одну велику кодову базу і не є модульними. Якщо виникає потреба щось оновити або змінити, розробники мають вносити зміни у весь стек одразу.

Спрощену схему функціонування сервісу з монолітною архітектурою наведено на рис. 2.5.

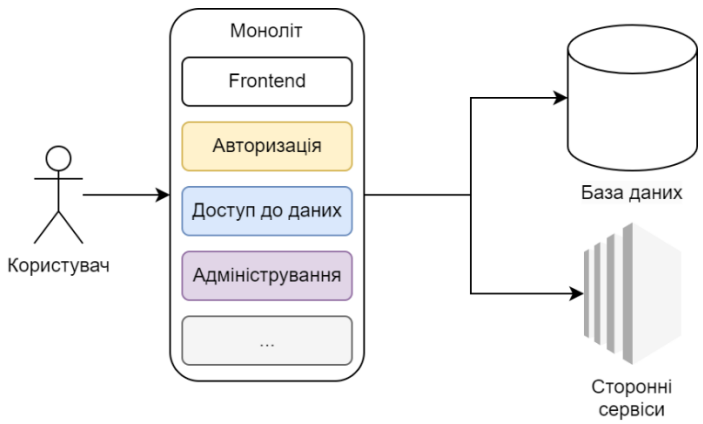


Рисунок 2.5 – Схема функціонування сервісу з монолітною архітектурою

Переваги монолітної архітектури:

- Менше наскрізних проблем. Наскрізні проблеми - це проблеми, які впливають на весь додаток, такі як ведення журналів, обробка, кешування та моніторинг продуктивності тощо. Оскільки у монолітному додатку цей функціонал зосереджений в одному місці, з ним легше впоратися.
- Безперешкодне налагодження та тестування. Якщо порівнювати монолітну архітектуру з архітектурою мікросервісів, то монолітні застосунки набагато легше налагоджувати і тестувати. Оскільки монолітний додаток працює як одна неподільна одиниця, ви можете проводити всі види тестування набагато швидше.
- Швидке та просте розгортання. Ще однією перевагою монолітної архітектури є швидке та просте розгортання.
- Простіша розробка. Оскільки монолітний підхід вважається традиційним способом побудови додатків, більшість кваліфікованих інженерів мають необхідні знання та можливості для розробки монолітних додатків.

Недоліки монолітної архітектури:

- Складність коду. Велику кодову базу в межах одного додатку важко підтримувати. Коли монолітний додаток масштабується, стає занадто складно зрозуміти і розділити право власності на частини.
- Висока взаємозалежність компонентів. Впровадження змін у такому великому та складному додатку з високим рівнем взаємозв'язку може стати проблемою. Будь-яка зміна коду впливає на всю систему, тому вона створює більше технічних проблем і повинна бути ретельно скоординована.
- Обмежена масштабованість. З монолітною архітектурою не є можливим масштабувати програмні компоненти незалежно, лише весь додаток.
- Бар’єри для нових технологій. Вкрай проблематично здійснювати міграції на більш актуальні технології в монолітному додатку, оскільки тоді доведеться переписувати значну частину програмного коду.

2.4.2 Мікросервісна архітектура

У той час як монолітний додаток є єдиним цілим, мікросервісна архітектура розбиває його на набір менших незалежних блоків. Ці блоки виконують кожен процес системи як окремий сервіс. Таким чином, всі сервіси інкапсулюють власну логіку та бази даних, а також виконують специфічні функції.

В цьому архітектурному підході вся функціональність розділена на модулі, що розгортаються незалежно та які взаємодіють один з одним за допомогою визначених інтерфейсів. Кожен сервіс охоплює власну сферу застосування і може бути оновленим, розгорнутим та масштабованим незалежно.

Спрощену схему функціонування інтернет-магазину з мікросервісною архітектурою наведено на рис. 2.6.

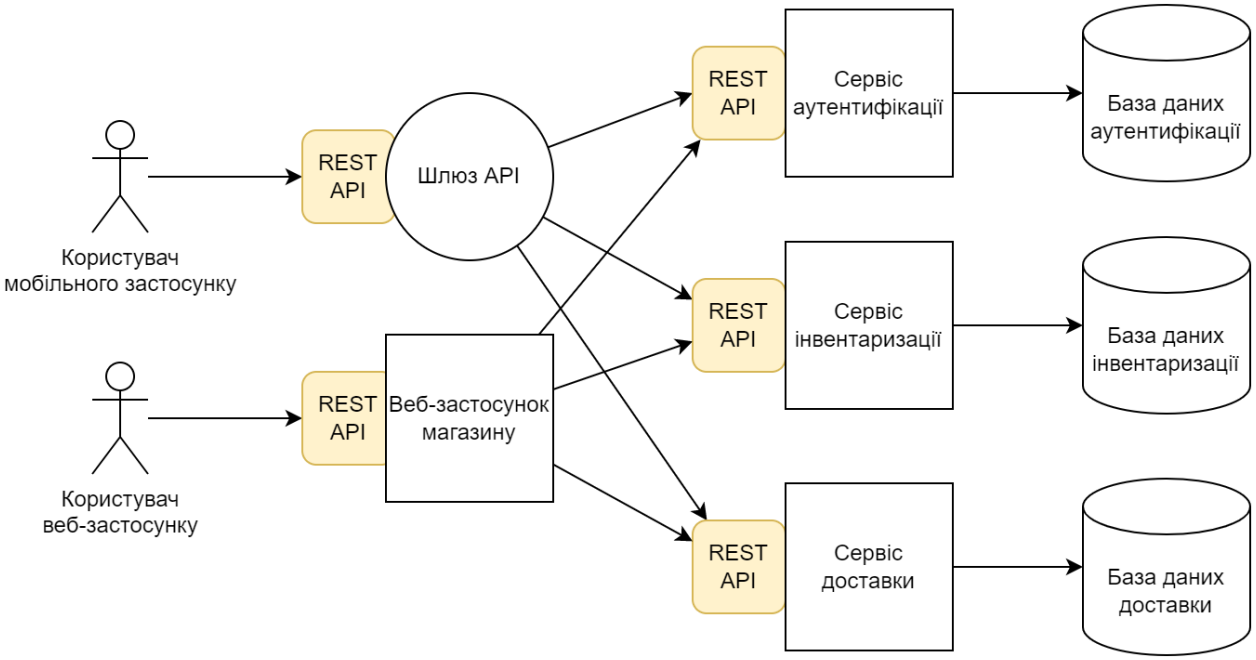


Рисунок 2.6 – Схема функціонування сервісу з мікросервісною архітектурою

Переваги мікросервісної архітектури:

– Незалежні компоненти. В порівнянні з монолітною архітектурою, ця архітектура пропонує набагато більшу гнучкість - всі сервіси можна розгортати та оновлювати незалежно. Більше того, помилка в одному мікросервісі впливає лише на певний сервіс, а не на весь додаток. Крім того, додавати нові функції до мікросервісного додатку набагато простіше, ніж до монолітного.

- Простіша підтримка. Розбитий на менші та простіші компоненти, додаток мікросервісу легше зрозуміти та керувати ним. Ви просто зосереджуєтесь на конкретній послузі, яка пов'язана з вашою бізнес-ціллю.
- Краща масштабованість. Ще однією перевагою підходу мікросервісів є те, що кожен елемент можна масштабувати незалежно. Таким чином, весь процес є більш економічно та часово ефективним, ніж у випадку з монолітами, коли весь додаток потрібно масштабувати, навіть якщо в цьому немає потреби. Крім того, кожен моноліт має обмеження щодо масштабованості, бо можливість горизонтального масштабування обмежена, або не існує, а вертикально масштабуватись можливо лише до певної межі.
- Гнучкість у виборі технології. Команди можуть вибирати технології, які найкраще відповідають конкретним вимогам і потребам кожного сервісу. Це дозволяє використовувати найефективніші інструменти для кожного компоненту системи, підвищуючи загальну продуктивність розробки. Також, це стимулює інновації, оскільки розробники мають можливість експериментувати з новими технологіями без необхідності переробки всього додатку.

Недоліки мікросервісної архітектури

- Відносна складність. Оскільки архітектура мікросервісів є розподіленою системою, вам доведеться налаштовувати та підтримувати зв'язки між усіма модулями та базами даних.
- Наскрізні проблеми. При створенні мікросервісного додатку вам доведеться мати справу з низкою глобально-системних проблем. До них відносяться конфігурація, ведення журналів, метрики, вимірювання продуктивності тощо.
- Труднощі з тестуванням. Велика кількість компонентів, що розгортаються незалежно, значно ускладнює тестування програмного забезпечення, в порівнянні з монолітною архітектурою.

2.5 Огляд методів представлення даних

2.5.1 Веб-інтерфейси

Веб-інтерфейс може бути основним засобом взаємодії користувача з системою. Він забезпечує зручний, інтуїтивно зрозумілий та ефективний спосіб пошуку, перегляду та взаємодії з даними. Інтерактивність веб-інтерфейсу робить його набагато привабливішим і кориснішим для користувача, оскільки дозволяє виконувати різні дії безпосередньо на веб-сторінці. Наприклад, користувач може фільтрувати дані за певними параметрами, сортувати їх за різними критеріями, здійснювати пошук за ключовими словами, взаємодіяти з різними візуалізаціями даних.

Усі розглянуті в підрозділі 1.2 існуючі системи використовують веб-сайти для комунікації з користувачами.

Однією з переваг веб-інтерфейсу є його універсальність та доступність. Його можна запустити на будь-якому пристрої, що має доступ до Інтернету та веб-браузер, незалежно від операційної системи чи типу пристрою. Це дозволяє користувачам отримувати доступ до даних у будь-який час і з будь-якого місця, що робить управління даними більш гнучким і зручним.

Також, веб-інтерфейс можна легко адаптувати та модифікувати відповідно до потреб користувачів та системних вимог. Наприклад, нові функції та можливості можуть бути додані шляхом оновлення веб-додатку без необхідності перевстановлення програмного забезпечення на пристрої кожного користувача.

Оскільки веб-інтерфейс працює у веб-браузері, розробники можуть працювати в середовищі розробки, якому вони надають перевагу, без необхідності враховувати конкретну операційну систему або апаратну платформу кінцевого користувача. Це дозволяє їм зосередитися на процесі розробки самого веб-інтерфейсу, забезпечуючи швидкий і ефективний цикл розробки.

Крім того, завдяки універсальності веб-технологій, веб-інтерфейси можна розробляти з використанням широкого спектру фреймворків і бібліотек, що

робить їх дуже гнучкими і адаптованими до потреб різних проектів. Розробники можуть використовувати широкий діапазон інструментів і технологій для створення веб-інтерфейсу, який відповідає конкретним потребам і вимогам проекту.

Такий підхід спрощує розробку та підтримку веб-інтерфейсу, оскільки дозволяє розробникам зосередитися на функціональності та дизайні, не витрачаючи час на оптимізацію під конкретні пристрої або операційні системи. Це робить веб-інтерфейси доступнішими і швидшими в розгортанні, що сприяє їх широкому впровадженню і використанню в різних галузях.

Таким чином, веб-інтерфейс може виступати в ролі ключового способу представлення даних, який забезпечує зручність, доступність та інтерактивність для користувачів.

2.5.2 Застосунки для мобільних пристроїв

Мобільні застосунки є важливим елементом сучасних систем представлення даних, оскільки вони надають користувачам зручний доступ до інформації з будь-якого місця та в будь-який час. Розробка застосунків для мобільних пристроїв включає в себе створення інтерфейсів, оптимізованих для використання на екранах малих розмірів, з урахуванням особливостей та можливостей сенсорних пристроїв. Згідно зі статистикою (Рисунок 2.7), мобільні пристрої мають більшу частку ринку, ніж стаціонарні ПК.

Мобільні застосунки дозволяють користувачам легко отримувати доступ до екологічних даних, отримувати сповіщення про їхнє оновлення та використовувати різноманітні функціональні можливості, такі як GPS.

Основні переваги мобільних застосунків включають їх мобільність, зручний інтерфейс користувача та можливість використання різноманітних функцій пристрою для збору та візуалізації даних. Крім того, вони можуть працювати в автономному режимі, що дозволяє користувачам отримувати доступ до інформації навіть у відсутності Інтернет-з'єднання.

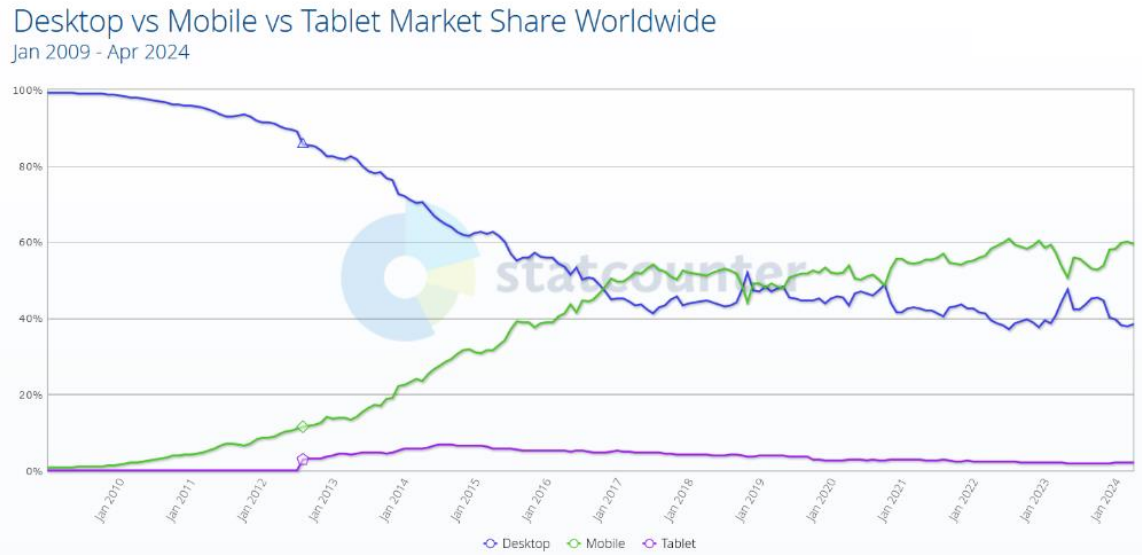


Рисунок 2.7 – Частка ринку настільних комп'ютерів, мобільних телефонів та планшетів у світі [24]

Розробка мобільних застосунків вимагає уваги до дизайну та оптимізації для різних платформ, таких як iOS та Android. Важливо також забезпечити безпеку та конфіденційність даних, особливо у випадку збору особистої інформації користувачів.

Застосунки для мобільних пристроїв можуть бути ефективним інструментом для розповсюдження екологічної інформації та підвищення обізнаності громадськості щодо проблем довкілля. Їхня доступність та зручність користування може сприяти активному участь у програмах моніторингу та захисту довкілля.

2.5.3 Боти в соціальних мережах

Боти в соціальних мережах - це програмні агенти, які автоматизують комунікацію та надають різноманітні сервіси через платформи соціальних мереж. Вони можуть бути використані для представлення екологічних даних широкій аудиторії, надання порад щодо екологічних питань, а також для отримання зворотного зв'язку від користувачів.

2 з 3 розглянутих в підрозділі 1.2 існуючих систем використовують ботів для комунікації з користувачами.

Основною перевагою використання ботів у соціальних мережах є їхня доступність для широкого кола користувачів. Боти можуть бути інтегровані в популярні месенджери, такі як Facebook Messenger, Telegram, Viber, а також у соціальні мережі, що дозволяє надавати користувачам доступ до екологічної інформації безпосередньо з їхніх улюблених платформ.

Використання ботів також дозволяє автоматизувати багато рутинних завдань, таких як надання відповідей на питання користувачів, надсилання сповіщень про оновлення даних або нагадування про важливі події в сфері екології.

Проте важливо враховувати, що ефективність ботів в соціальних мережах залежить від їхнього дизайну та взаємодії з користувачами. Вони повинні бути легкими у використанні та надавати корисну та цікаву інформацію, щоб залучити та утримати увагу аудиторії.

2.5.4 Програмні інтерфейси (API)

Програмний інтерфейс (API) - це набір правил та протоколів, які дозволяють різним програмам взаємодіяти одна з одною. У контексті представлення екологічних даних, API може використовуватись для забезпечення доступу до даних з системи моніторингу екології, дозволяючи розробникам та іншим зацікавленим сторонам інтегрувати ці дані у свої власні програми або сервіси.

API надається у вигляді набору програмних інструкцій та методів, які можна викликати для отримання потрібної інформації. Це може бути дані про якість повітря, рівень забруднення води, погодні умови тощо. Зазвичай, API надається у вигляді веб-служби, яка відповідає на запити користувачів та надсилає їм потрібні дані у вигляді структурованого формату, такого як JSON або XML.

Популярною архітектурою для побудови API є REST(Representational State Transfer). REST - це архітектурний стиль, який використовується для створення веб-служб, що дозволяють взаємодіяти з ресурсами через

стандартні HTTP-запити. У контексті представлення екологічних даних, REST є одним з ключових інструментів для надання доступу до цих даних розробникам та іншим зацікавленим сторонам. API, які відповідають архітектурному стилю REST, називають REST(-ful) API.

Основні принципи REST API включають в себе:

1. Ресурси та URI: Кожен ресурс (наприклад, дані про якість повітря або рівень забруднення води) представлений у вигляді URI (Uniform Resource Identifier). Користувачі можуть використовувати ці URI для доступу до ресурсів через HTTP-запити.
2. HTTP-методи: RESTful API використовує стандартні HTTP-методи, такі як GET, POST, PUT та DELETE, для взаємодії з ресурсами. Наприклад, GET-запит використовується для отримання даних, POST - для створення нових записів, PUT - для оновлення існуючих даних, а DELETE - для видалення даних.
3. Представлення даних: Дані, що надсилаються та отримуються через RESTful API, зазвичай представлені у форматі JSON або XML. Ці формати дозволяють структурувати дані та забезпечують зручну передачу та обробку інформації.
4. Відсутність стану: RESTful API не має стану, що означає, що кожен запит до сервера містить всю необхідну інформацію для виконання цього запиту. Сервер не зберігає стан клієнта між запитами.

Однією з ключових переваг використання API є можливість інтеграції екологічних даних у різноманітні додатки та сервіси, що використовуються широким колом користувачів. Наприклад, це може бути мобільний додаток для відстеження якості повітря, веб-сайт для аналізу екологічних тенденцій або інтерактивна карта з маркерами забруднених ділянок.

Проте варто враховувати, що використання API вимагає додаткових технічних знань та навичок у програмуванні. Користувачам, які бажають використовувати API, необхідно мати розуміння роботи з HTTP-запитами, обробки

JSON або XML-даних, а також знати основи автентифікації та авторизації для доступу до захищених ресурсів.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі даної дипломної роботи було розглянуто технології та підходи які можливо використати для реалізації основних аспектів системи збору та аналізу екологічних даних.

По-перше, були розглянуті методи збору даних з існуючих джерел, включаючи використання відкритих API та вебскрейпінг. Проаналізовано переваги та недоліки цих підходів, а також їхню придатність для отримання необхідної екологічної інформації.

По-друге, досліджено різноманітні системи зберігання та обробки даних, включаючи PostgreSQL, Oracle Database, MongoDB, Apache Cassandra та Elasticsearch. Кожна з цих систем має свої особливості та переваги, що дозволяють з різною ефективністю зберігати та обробляти великі обсяги екологічних даних.

По-третє, розглянуті аналітичні методи та інструменти, зокрема методи статистичного аналізу та методи машинного навчання. Ці методи дозволяють виконувати різноманітний аналіз екологічних даних та виявляти закономірності та тенденції.

По-четверте, розглянуті архітектурні підходи, зокрема монолітна та мікросервісна архітектури, які можуть бути використані для побудови складних систем з урахуванням вимог до масштабованості та ефективності.

Нарешті, досліджено різні методи представлення зібраних даних, включаючи веб-інтерфейси, застосунки для мобільних пристроїв, боти в соціальних мережах та програмні інтерфейси (API). Кожен із цих методів має свої переваги та специфічні особливості, що робить їх цікавими варіантами для використання в системах збору та аналізу екологічних даних.

РОЗДІЛ 3
ОПИС АРХІТЕКТУРИ І КОМПОНЕНТІВ РОЗРОБЛЕНОЇ СИСТЕМИ ЗБОРУ ТА АНАЛІЗУ ЕКОЛОГІЧНИХ ДАНИХ

3.1 Опис архітектури розробленої системи

На рисунку 3.1 наведено архітектуру системи та зв’язки між її компонентами.

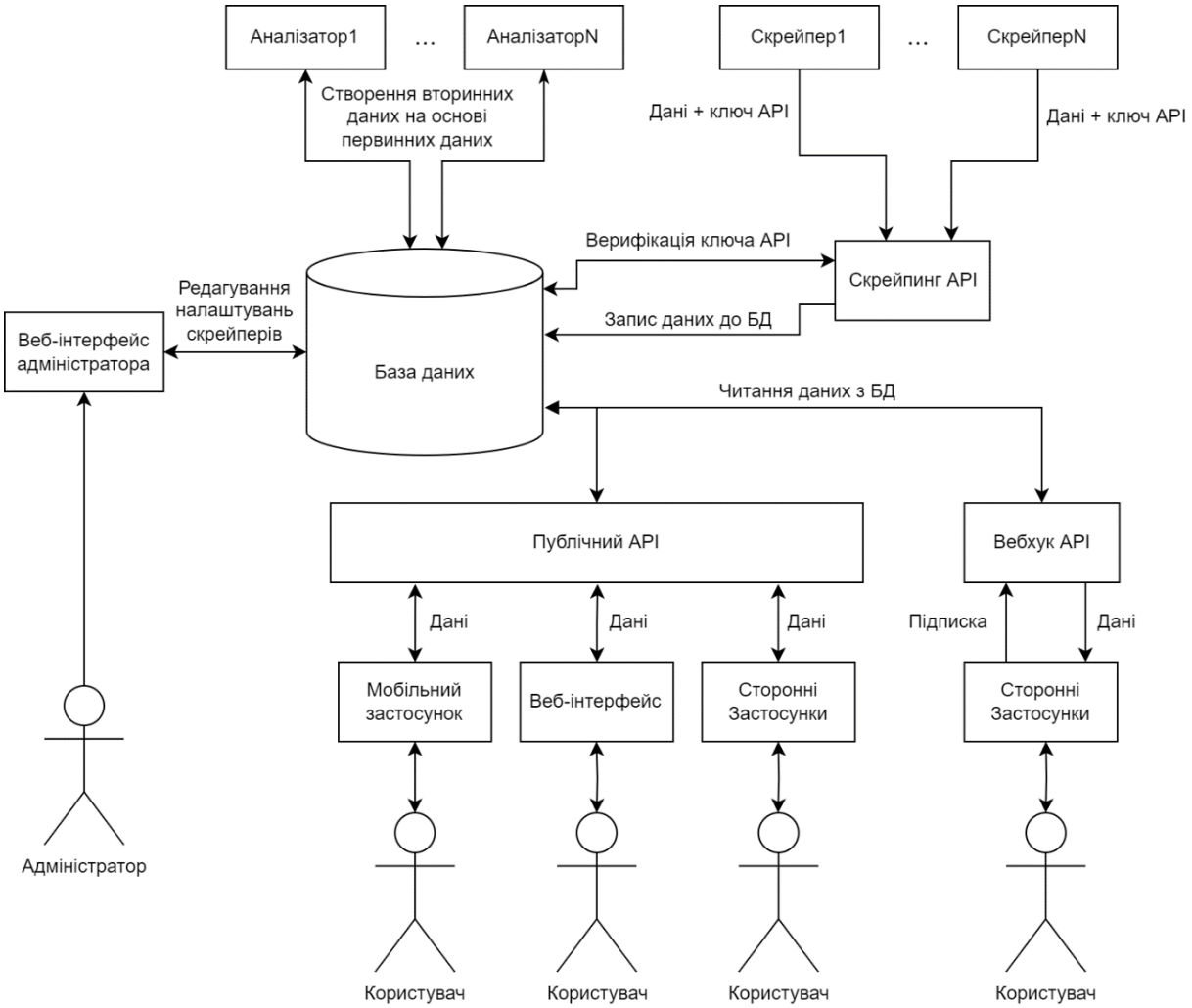


Рисунок 3.1

В проектуванні архітектури було використано поєднання підходів, розглянутих в підрозділі 2.4 даної роботи.

Від монолітного підходу в даній архітектурі – наявність єдиної бази даних, яка виконує роль централізованого сховища всіх даних системи. Від цієї бази даних безпосередньо або опосередковано залежать всі інші компоненти системи.

Підхід мікросервісів в даній архітектурі проявляється в тому, що система складається з декількох незалежних модулів, кожен з яких відповідає за певну функціональність. Скрейпери відповідають за збір даних, Скрейпинг API відповідає за прийом даних від Скрейперів та запис їх до бази даних, Аналізатори відповідають за генерацію вторинних даних на основі зібраних (первинних), веб-інтерфейс адміністратора відповідає за налаштування скрейперів, публічний API відповідає за надання первинних та вторинних даних користувачам, вебхук API відповідає за надання даних за допомогою механізму вебхуків.

Компоненти системи можуть бути реалізовані на різних мовах програмування та працювати на різних серверах, що дозволяє легше виконувати вертикальне масштабування та розподілення навантаження.

Система передбачає можливість додавання нових скрейперів та аналізаторів, що відповідає принципам мікросервісів, де можна додавати нові сервіси без значних змін в існуючій інфраструктурі.

Серед інших переваг архітектури:

- Підвищення гнучкості. Можна легко змінювати функціональність окремих модулів, не впливаючи на інші.
- Збільшення стійкості. Якщо один з модулів вийде з ладу, інші модулі зможуть продовжувати працювати.
- Полегшення розробки та тестування індивідуальних модулів. Модулі можна розробляти та тестувати незалежно один від одного.

3.2 Опис збирачів (скрейперів) даних системи

3.2.1 Український гідрометеорологічний центр

Український гідрометеорологічний центр державної служби України з надзвичайних ситуацій раз на день публікує виміряну на пунктах спостереження радіометричної мережі НГМС потужність експозиційної дози [25]. Дані подаються у вигляді карти з маркерами (рисунок 3.2) та таблиці (рисунок 3.3).

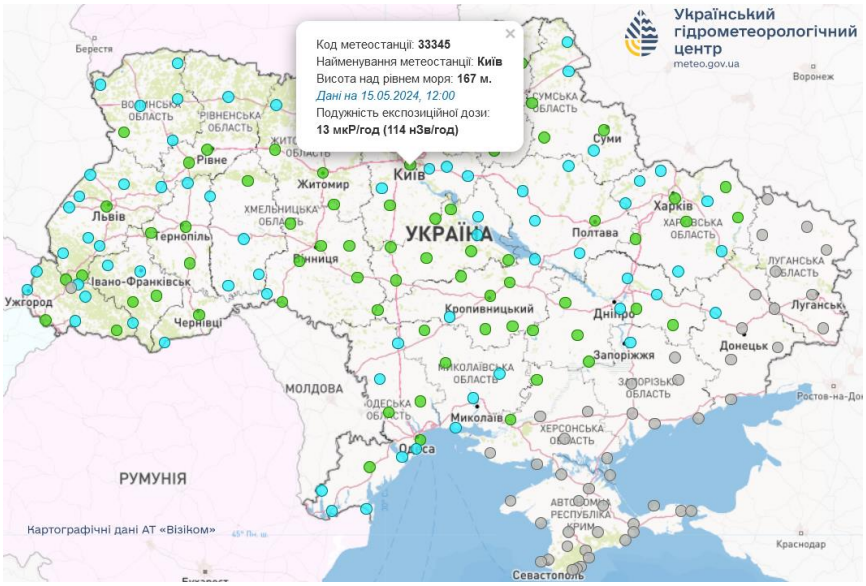


Рисунок 3.2 – Ситуація на пунктах спостереження радіометричної мережі НГМС (карта) [25]

Найменування метеостанції	Дані на	Подужність експозиційної дози (нЗв/год)
33049 Семенівка	15.05.2024, 09:00	87
33058 Дружба	14.05.2024, 09:00	79
33067 Світязь	15.05.2024, 09:00	79
33075 Любешів	15.05.2024, 09:00	87
33088 Сарни	15.05.2024, 09:00	96
33135 Чернігів	15.05.2024, 09:00	96
33136 Сновськ	15.05.2024,	96

Рисунок 3.3 – Ситуація на пунктах спостереження радіометричної мережі НГМС (таблиця) [25]

За допомогою дослідження запитів, які робить браузер для завантаження цих даних, було визначено, що потужність експозиційної дози отримується за URL https://www.meteo.gov.ua/_/m/radioday.js , а назва та координати пунктів спостереження за URL https://www.meteo.gov.ua/ua/_radio-posts.js. Зразки відповідей наведено на рисунку 3.4.

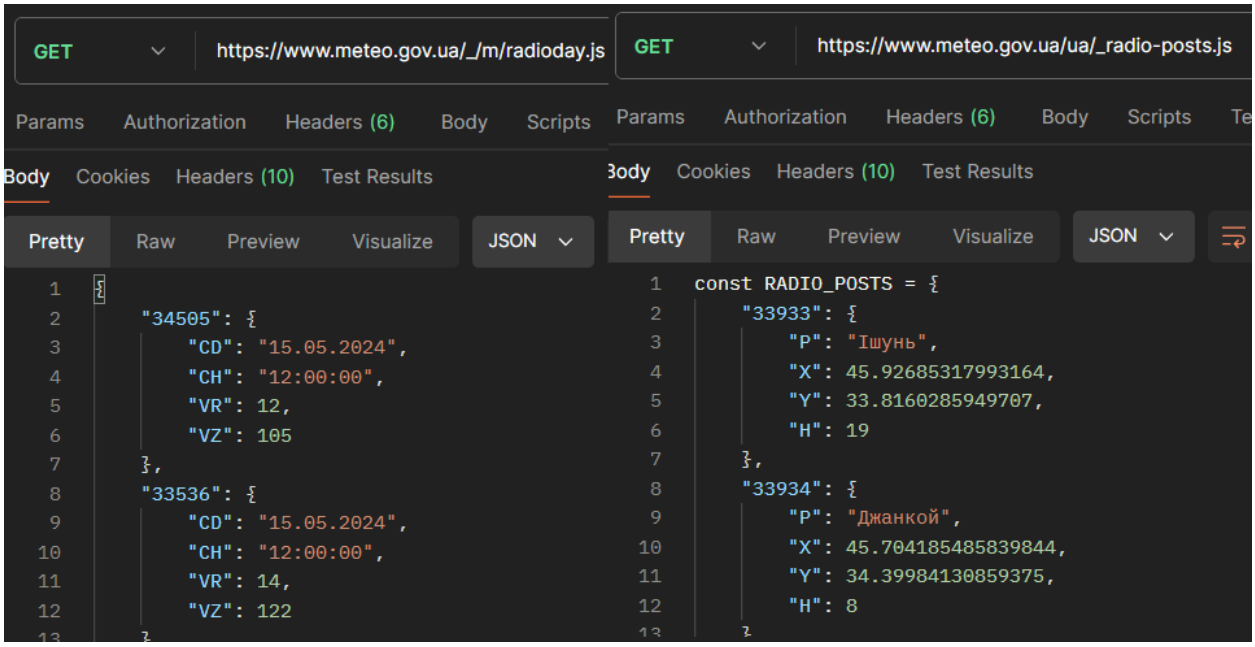


Рисунок 3.4

Записи подаються у вигляді JSON об’єктів, де ключ – число-ідентифікатор пункту спостереження, а значення – пари ключів та значень. “CD” та “CH” – день та час вимірювання відповідно. “VR” та “VZ” – потужність експозиційної дози в мкР/год та нЗв/год відповідно.

Для реалізації збирача (скрейпера) даних про радіологічний стан з цього джерела було використано мову програмування Kotlin [26]. В якості HTTP-клієнта – Ktor [27], для серіалізації та десеріалізації JSON – бібліотеку Gson [28].

3.2.2 ЕкоЗагроза

ЕкоЗагроза – “офіційний ресурс Міністерства захисту довкілля та природних ресурсів України, розроблений за підтримки Міністерства цифрової трансформації України, стандартизована форма для автоматичного збору та

фіксації інформації про екологічні загрози в режимі реального часу, з географічною прив'язкою до місцевості" [29]. Зібрані дані, зокрема, подаються у вигляді карти (рисунок 3.5).

Рисунок 3.5 – Пост контролю радіаційного стану ресурсу ЕкоЗагроза

За допомогою дослідження запитів, які робить браузер під час завантаження сторінки карти, вдалось визначити, що потужність експозиційної дози отримується за URL https://sensor.ecozagroza.gov.ua/api/sensors?layer=radiation. Приклад відповіді наведено на рисунку 3.6.

Рисунок 3.6

Для реалізації збирача (скрейпера) даних про радіологічний стан з цього джерела було використано ті ж засоби, що і для минулого: мова програмування Kotlin [26], HTTP-клієнт Ktor [27] та бібліотека Gson [28].

3.2.3 ЛУН Місто Air

"ЛУН Місто Air" [5] - українська real-time система моніторингу якості повітря, яку було детально розглянуто в пункті 1.2.2 даної роботи.

За допомогою дослідження запитів, які робить браузер під час завантаження даних для карти якості повітря (рисунок 1.2), було визначено, що актуальні дані про якість повітря знаходяться за URL https://misto.lun.ua/api/v1/air/stations. Приклад відповіді наведено на рисунку 3.7.

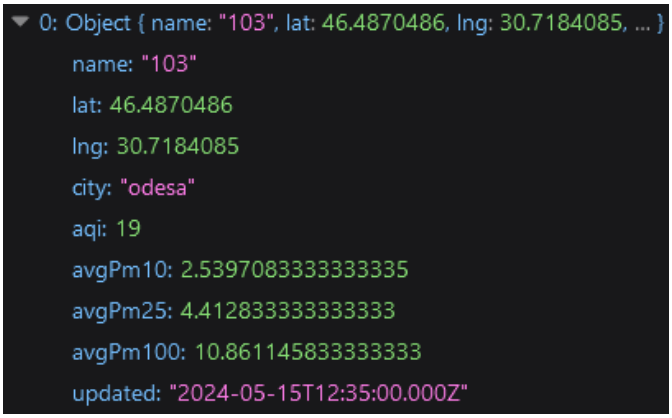


Рисунок 3.7

Для реалізації збирача (скрейпера) даних про якість повітря з цього джерела було використано ті ж засоби, що і для минулих: мова програмування Kotlin [26], HTTP-клієнт Ktor [27] та бібліотека Gson [28].

3.2.4 SaveDnipro

SaveDnipro [30] – громадська організація, яка займається пошуком шляхів впливу на екологічну ситуацію в Україні та світі, заснована в 2020 році. Є авторами проєкту SaveEcoBot, який було розглянуто в пункті 1.2.1 даної роботи.

Актуальні дані стану атмосферного повітря станцій, які було зібрано ГО "SaveDnipro" у форматі JSON доступні за посиланням https://api.saveecobot.com/output.json. Приклад даних наведено на рисунку 3.8.

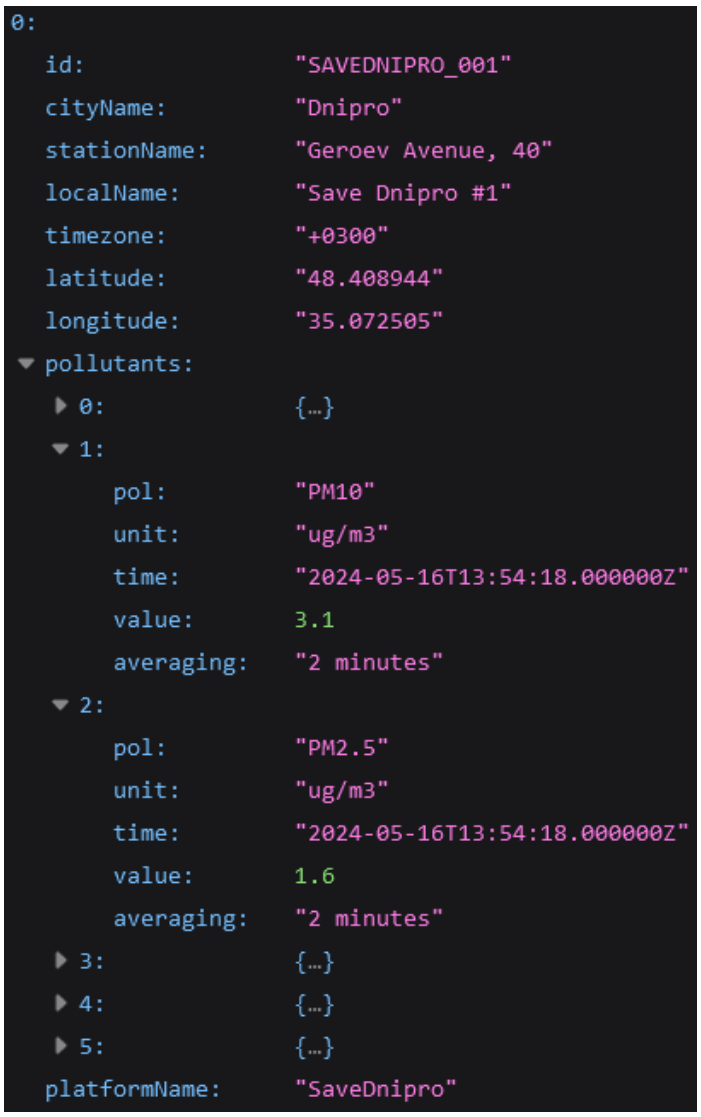


Рисунок 3.8

Для створення збирача (скрейпера) даних про якість повітря з цього джерела використано ті ж інструменти, що й раніше: мова програмування Kotlin [26], HTTP-клієнт Ktor [27] та бібліотека Gson [28].

3.2.5 Sensor.Community

Sensor.Community [31] – це глобальна ініціатива, що має на меті підвищення обізнаності про якість повітря. Проєкт був заснований у 2015 році як Luftdaten.info у Штутгарті, Німеччина, та з часом перейменований на

Sensor.Community, щоб відобразити його глобальний характер. Sensor.Community об'єднує тисячі волонтерів по всьому світу, які встановлюють датчики для вимірювання якості повітря та діляться цими даними через інтернет.

Проєкт спрямований на створення доступної, відкритої та точкової системи моніторингу якості повітря, яка дозволяє людям отримувати актуальні дані про рівень забруднення у своїй місцевості. Встановлюючи недорогі датчики, які вимірюють рівень дрібнодисперсного пилу (PM2.5 та PM10), волонтери сприяють формуванню розгалуженої мережі моніторингу, що надає актуальні дані в режимі реального часу.

Актуальні дані про стан атмосферного повітря на території України, зібрані Sensor.Community, доступні у форматі JSON за посиланням https://data.sensor.community/airrohr/v1/filter/country=UA. Приклад запису наведено на рисунку 3.9.

Рисунок 3.9

3.2.6 NASA FIRMS

Система пожежної інформації для управління ресурсами (FIRMS) [32] поширює дані про активні пожежі в режимі близькому до реального часу (NRT) від спектрорадіометра з помірною роздільною здатністю (MODIS) на борту супутників Aqua і Terra, а також набору радіометрів для отримання зображень в інфрачервоному діапазоні (VIIRS) на борту супутників S-NPP, NOAA 20 і NOAA 21 (офіційно відомих як JPSS-1 і JPSS-2). У глобальному масштабі ці дані доступні протягом 3 годин після супутникового спостереження.

Отримані дані про пожежі доступні у вигляді інтерактивної карти на вебсайті проєкту [33] (рисунок 3.10), та у форматі CSV через публічний API [34].

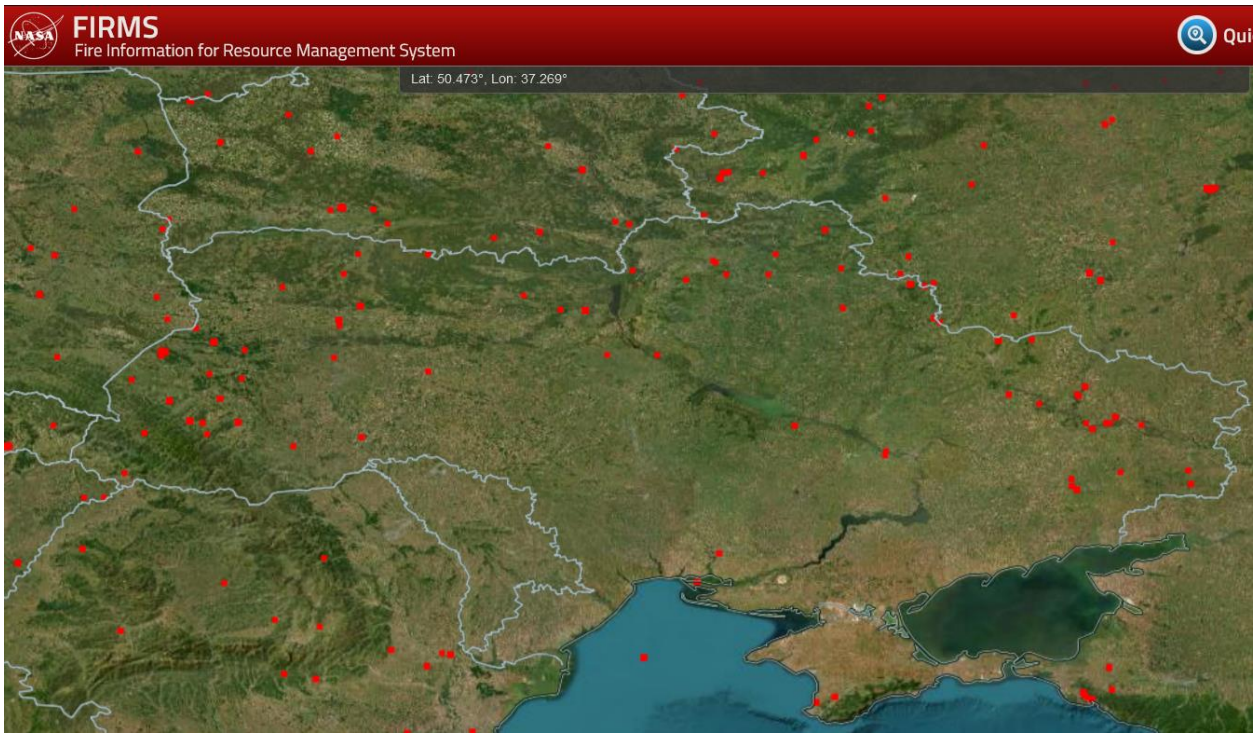


Рисунок 3.10

Для створення збирача (скрейпера) даних про пожежі з цього джерела використано мову програмування Kotlin [26] та HTTP-клієнт Ktor [27].

3.3 Опис бази даних системи

3.3.1 Oracle Autonomous Database

Для реалізації системи збору та аналізу екологічних даних в якості бази даних обрано Oracle Autonomous Database. Цей вибір обумовлений кількома важливими факторами, що забезпечують ефективність, надійність та доступність системи.

1. Програма Always Free [35]:

Однією з основних переваг Oracle Autonomous Database є наявність програми Always Free, в рамках якої Oracle Cloud надає можливість безкоштовного використання двох автономних баз даних. Це дозволяє повністю уникнути витрат на інфраструктуру, що є важливим фактором для будь-якого проєкту, особливо на початкових етапах розробки. Безкоштовний доступ до баз даних з повною функціональністю дозволяє зосередитись на розробці і вдосконаленні системи без необхідності турбуватися про витрати на зберігання даних.

2. Автономні можливості:

Oracle Autonomous Database є автономною базою даних, яка забезпечує автоматичне управління, налаштування, безпеку та ремонт. Це означає, що значна частина адміністративної роботи виконується автоматично, знижуючи потребу в постійному моніторингу та обслуговуванні бази даних. Автоматичні оновлення і патчі забезпечують захист від вразливостей, а автоматичне масштабування дозволяє ефективно керувати навантаженням без додаткових зусиль з боку розробників.

3. Висока продуктивність та надійність:

Oracle Autonomous Database пропонує високий рівень продуктивності та надійності. Завдяки оптимізованим алгоритмам обробки запитів і управління ресурсами, ця база даних забезпечує швидкий доступ до даних і високу швидкість виконання запитів. Надійні механізми резервного копіювання і

відновлення даних гарантують безпеку інформації і мінімізують ризики втрати даних.

4. Масштабованість:

Масштабованість є критично важливою властивістю для систем збору та аналізу екологічних даних, оскільки обсяги даних можуть швидко збільшуватись, що було детально розглянуто в пункті 1.3.4 даної роботи. Oracle Autonomous Database підтримує горизонтальне і вертикальне масштабування, що дозволяє легко адаптуватися до змінних навантажень і забезпечувати стабільну роботу системи при зростанні обсягів даних.

5. Підтримка різноманітних типів даних:

Oracle Autonomous Database підтримує різні типи даних, включаючи реляційні, JSON, XML та просторові дані. Це забезпечує гнучкість у зберіганні та обробці різноманітних екологічних даних, що є необхідним для комплексного аналізу інформації з різних джерел.

6. Інтеграція з хмарними сервісами:

Інтеграція з іншими хмарними сервісами Oracle Cloud дозволяє легко розширювати функціональність системи. Використання додаткових сервісів для аналізу даних, машинного навчання та візуалізації значно покращує можливості системи та забезпечує більш глибоке розуміння екологічних процесів.

Таким чином, вибір Oracle Autonomous Database для реалізації системи обґрунтований її безкоштовною доступністю в рамках програми Always Free, автономними можливостями управління, високою продуктивністю та надійністю, масштабованістю, підтримкою різноманітних типів даних та інтеграцією з хмарними сервісами. Ці переваги роблять Oracle Autonomous Database оптимальним рішенням для створення надійної та ефективної системи управління екологічною інформацією.

3.3.2 Схема

Схема бази даних наведена в додатку 2.

Схема бази даних містить сім таблиць: SCRAPERS, AIRQUALITYRECORD, RADIATIONRECORD, FIRERECORD, WEBHOOKS, та PENDINGWEBHOOKCALLS. Кожна таблиця містить набір стовпців, які зберігають конкретні дані, а також відносини між таблицями через зовнішні ключі.

Таблиця SCRAPERS.

Ця таблиця зберігає інформацію про різні джерела даних, з яких збираються екологічні дані.

- ID: Унікальний ідентифікатор (primary key).
- NAME: Назва джерела даних (varchar2(50)).
- API_KEY: Ключ API для доступу до даних (char(32)).
- DESCRIPTION_EN: Опис джерела англійською мовою (varchar2(2000)).
- DESCRIPTION_UK: Опис джерела українською мовою (varchar2(2000)).
- URL: URL джерела даних (varchar2(200)).

Таблиця AIRQUALITYRECORD.

Ця таблиця зберігає записи про якість повітря.

- ID: Унікальний ідентифікатор (primary key).
- LATITUDE: Широта місця вимірювання (float).
- LONGITUDE: Довгота місця вимірювання (float).
- TIMESTAMP: Час вимірювання (number(19)).
- PROVIDER: Зовнішній ключ, що посилається на таблицю SCRAPERS (number(12)).
- METADATA: Додаткова інформація про вимірювання (varchar2(500)).
- CREATEDAT: Час створення запису (number(19)).
- PM10: Концентрація частинок PM10 (float).
- PM25: Концентрація частинок PM2.5 (float).
- PM100: Концентрація частинок PM1.0 (float).

Таблиця RADIATIONRECORD.

Ця таблиця зберігає записи про рівень радіації.

- ID: Унікальний ідентифікатор (primary key).

- LATITUDE: Широта місця вимірювання (float).
- LONGITUDE: Довгота місця вимірювання (float).
- TIMESTAMP: Час вимірювання (number(19)).
- PROVIDER: Зовнішній ключ, що посилається на таблицю SCRAPERS (number(12)).
- METADATA: Додаткова інформація про вимірювання (varchar2(500)).
- CREATEDAT: Час створення запису (number(19)).
- DOSEINANOSIEVERT: Доза радіації в нанозівертах на годину (number(12)).

Таблиця FIRERECORD.

Ця таблиця зберігає записи про пожежі.

- ID: Унікальний ідентифікатор (primary key).
- LATITUDE: Широта місця пожежі (float).
- LONGITUDE: Довгота місця пожежі (float).
- TIMESTAMP: Час виявлення пожежі (number(19)).
- PROVIDER: Зовнішній ключ, що посилається на таблицю SCRAPERS (number(12)).
- METADATA: Додаткова інформація про пожежу (varchar2(500)).
- CREATEDAT: Час створення запису (number(19)).
- SCAN: Скан пожежі(в кілометрах) (float).
- TRACK: Трек пожежі(в кілометрах) (float).
- CONFIDENCE: Рівень впевненості в даних про пожежу (float).
- FIRERADIATIVEPOWER: Випромінювальна потужність пожежі (float).

Таблиця PENDINGWEBHOOKCALLS.

Ця таблиця зберігає записи про заплановані виклики вебхуків.

- ID: Унікальний ідентифікатор (primary key).
- CALLBACK_URL: URL для зворотного виклику (varchar2(300)).
- AIRQUALITYRECORD_ID: Зовнішній ключ, що посилається на таблицю AIRQUALITYRECORD (number(12)).

- RADIATIONRECORD_ID: Зовнішній ключ, що посилається на таблицю RADIATIONRECORD (number(12)).
- TESTRECORD_ID: Ідентифікатор тестового запису (number(12)).
- FIRERECORD_ID: Зовнішній ключ, що посилається на таблицю FIRERECORD (number(12)).

Таблиця WEBHOOKS.

Ця таблиця зберігає інформацію про вебхуки.

- ID: Унікальний ідентифікатор (primary key).
- LATITUDE: Широта місця (float).
- LONGITUDE: Довгота місця (float).
- CALLBACK_URL: URL для зворотного виклику (varchar2(300)).

Зв'язки між таблицями

1. SCRAPERS пов'язана з таблицями AIRQUALITYRECORD, RADIATIONRECORD та FIRERECORD через поле PROVIDER, що посилається на ID в таблиці SCRAPERS.
2. AIRQUALITYRECORD, RADIATIONRECORD, FIRERECORD пов'язані з таблицею PENDINGWEBHOOKCALLS через відповідні зовнішні ключі: AIRQUALITYRECORD_ID, RADIATIONRECORD_ID, FIRERECORD_ID.

3.3.3 Кешування

Для зменшення навантаження на базу даних було реалізовано кешування результатів запитів за допомогою бібліотеки Ehcache [36]. Ehcache - це кеш з відкритим вихідним кодом, який підвищує продуктивність, розвантажує базу даних і спрощує масштабування. Це найпоширеніший кеш на основі Java, оскільки він надійний, перевірений, повнофункціональний та інтегрується з іншими популярними бібліотеками та фреймворками.

Для різних типів запитів кешування різне. Наприклад, всі запити на отримання груп записів кешуються на 5 хвилин, що дозволяє значно зменшити

навантаження на базу даних, при цьому зберігши актуальність даних. А запити на підрахунок загальної кількості записів, які було зібрано з джерела, кешуються на 24 години, оскільки актуальність цих даних менш важлива для системи.

3.3.4 Пул з’єднань (Connection pooling)

Фреймворк Exposed ORM [37], який в проєкті використовується для перетворення об’єктів мови Kotlin в об’єкти бази даних (та навпаки) створює нове підключення до бази даних для кожної транзакції [38], що є дуже ресурсозатратним та тривалим процесом. Для вирішення цієї проблеми можна скористатись методом створення пулу з’єднань (connection pooling).

Connection Pooling (пул з'єднань) є технологією управління з'єднаннями до бази даних, яка використовується для підвищення продуктивності та ефективності роботи додатків, що потребують частого доступу до бази даних. Цей метод передбачає створення пулу з'єднань, які можна повторно використовувати, замість того, щоб створювати нові з'єднання кожного разу, коли додаток потребує доступу до бази даних.

Пул з'єднань функціонує наступним чином:

1. Ініціалізація пулу: Під час запуску додатку або на етапі конфігурації пул з'єднань ініціалізується та створюється певна кількість з'єднань до бази даних. Ця кількість залежить від конфігураційних налаштувань та потреб додатку.
2. Запит з'єднання: Коли додаток потребує доступу до бази даних, замість створення нового з'єднання, він запитує одне з наявних з'єднань з пулу.
3. Використання з'єднання: Додаток використовує з'єднання для виконання операцій з базою даних, таких як запити на читання, запис, оновлення чи видалення даних.

4. Повернення з'єднання до пулу: Після завершення операцій з базою даних з'єднання не закривається, а повертається до пулу, де воно знову стає доступним для використання іншим запитом додатку.

Для реалізації пулу з’єднань було використано бібліотеку HikariCP [39]. Максимальний розмір пулу – 4.

3.4 Опис аналітичних інструментів системи

3.4.1 Формування “знімків” екологічного стану

Одним із застосувань зібраних екологічних даних є створення "знімків" актуального стану довкілля на певний момент часу. Створені "знімки" можуть бути представлені, наприклад, у вигляді інтерактивних карт, як це роблять існуючі системи, розглянуті в підрозділі 1.2 цієї роботи. Ці "знімки" забезпечують візуалізацію екологічних показників у режимі реального часу, що дозволяє користувачам швидко оцінювати поточний стан довкілля в різних регіонах.

Для створення таких "знімків" використовується методика вибірки даних із бази даних, яка дозволяє отримати останні доступні записи з унікальними координатами (широта і довгота) за певний період часу. Цей підхід гарантує, що на інтерактивних картах відображаються найбільш актуальні дані, які є на момент запиту.

У даному проекті формування "знімків" екологічного стану здійснюється за допомогою SQL-запиту на стороні бази даних. Наведений алгоритм реалізовано у вигляді функції readLatestSubmittedRecordsWithDistinctLocations, яка виконує складний SQL-запит для вибірки даних. Код функції наведено на рисунку 3.11.

Метод працює наступним чином:

Спочатку формуються умови вибірки даних на основі часу. Якщо вказані параметри at (час запиту) та maxAge (максимальний вік даних), то вибірка

здійснюється за певний часовий проміжок. Якщо параметри не вказані, то вибірка здійснюється за всіма доступними даними.

Основна частина запиту включає в себе приєднання (join) таблиці записів (RecordsTable) до підзапиту, який групує дані за координатами (широтою і довготою) та вибирає останній запис (за часом) для кожної унікальної координати.

Після формування SQL-запиту він виконується на базі даних, а результати запиту перетворюються у відповідні об'єкти для подальшої обробки.

```
override suspend fun readLatestSubmittedRecordsWithDistinctLocations(
    at: Long?,
    maxAge: Long?
): List<T> = dbQuery {
    val sqlWhereTimestamp =
        if (at != null && maxAge != null)
            "WHERE timestamp < $at AND timestamp > ${at - maxAge}"
        else if (at != null)
            "WHERE timestamp < $at"
        else if (maxAge != null)
            "WHERE timestamp > ${System.currentTimeMillis() / 1000 - maxAge}"
        else ""

    val sqlStatement =
        """
            SELECT ${RecordsTable.allFieldsAsString( tableName: "t1")}
            FROM ${RecordsTable.nameInDatabaseCase()} t1
            JOIN (
                SELECT latitude, longitude, MAX(timestamp) AS latest_timestamp
                FROM ${RecordsTable.nameInDatabaseCase()}
                $sqlWhereTimestamp
                GROUP BY latitude, longitude
            ) t2 ON t1.latitude = t2.latitude AND t1.longitude = t2.longitude AND t1.timestamp = t2.latest_timestamp
        """.trimIndent()

    nativeSelect(sqlStatement).map { it.toEntity() } ^dbQuery
}
```

Рисунок 3.11 – Функція readLatestSubmittedRecordsWithDistinctLocations

3.4.2 Прогнозування забруднення повітря

Прогнозування забруднення повітря є важливим аспектом екологічного моніторингу, оскільки дозволяє передбачити можливі перевищення допустимих рівнів забруднюючих речовин і своєчасно реагувати. У цій роботі для прогнозування концентрацій забруднюючих речовин у повітрі використовувався інструмент Prophet [40], розроблений компанією Meta (раніше Facebook). Prophet є потужною бібліотекою для прогнозування часових рядів, яка добре

підходить для роботи з екологічними даними завдяки своїй здатності обробляти нерегулярні часові ряди та враховувати сезонність і свята.

Процес прогнозування складається з декількох етапів, що включають підготовку даних, тренування моделі, прогнозування та оцінку точності моделі.

Спочатку записи про якість повітря з таблиці AIRQUALITYRECORD для однієї з станцій в м. Київ були експортовані з бази даних у форматі CSV та оброблені за допомогою бібліотеки pandas. Обробка полягала в перетворенні поля TIMESTAMP у формат datetime, який є обов'язковим для роботи з Prophet. Крім того, значення PM10 використовувались як цільові (поле y), а всі рядки з відсутніми значеннями були видалені. Трансформовані дані були збережені у новий CSV-файл. Код python-скрипту, який це виконує, наведено на рисунку 3.12.

```python
import pandas as pd
from datetime import datetime

# Define the input CSV file path
input_csv_path = 'AIRQUALITYRECORD.csv'
output_csv_path = 'PM100.csv'

# Read the CSV file into a DataFrame
df = pd.read_csv(input_csv_path)

# Convert the 'TIMESTAMP' field to a datetime format and create the 'ds' field
df['ds'] = df['TIMESTAMP'].apply(lambda x: datetime.utcfromtimestamp(x).strftime('%Y-%m-%d %H:%M:%S'))

# Use the 'PM100' field for the 'y' value, and remove rows where 'PM100' is NaN
df = df.dropna(subset=['PM100'])
df = df.rename(columns={'PM100': 'y'})

# Select the relevant columns
df_meta_prophet = df[['ds', 'y']]

# Save the transformed DataFrame to a new CSV file
df_meta_prophet.to_csv(output_csv_path, index=False, quoting=2)  # quoting=2 for double quotes around strings

print(f"Transformation complete. Output saved to {output_csv_path}")
```

Рисунок 3.12 – Скрипт підготовки даних для Prophet

Після завантаження підготовлених даних з файлу PM100.csv, модель Prophet була налаштована і натренована на цих даних. Це дозволило моделі навчитися трендам і періодичним коливанням у даних про концентрацію

забруднюючих речовин. Код python-скрипту, який виконує тренування, наведено на рисунку 3.13.

```
import pandas as pd
from prophet import Prophet
from prophet.plot import plot_plotly

df = pd.read_csv('PM100.csv')
print(df.head())

m = Prophet()
m.fit(df)
```

Рисунок 3.13 – Скрипт тренування моделі Prophet

Після тренування моделі можна здійснювати прогнозування на майбутній період. У якості прикладу розглянемо створення прогнозу на наступні 7 днів з інтервалом у 10 хвилин. Результати прогнозу включають передбачені значення (yhat) та межі похибки (yhat_lower та yhat_upper). Код який виконує прогнозування, наведено на рисунку 3.14.

```
future = m.make_future_dataframe(freq='10min', periods=1008)
print(future.tail())

forecast = m.predict(future)
print(forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail())

plot_plotly(m, forecast).show()
```

Рисунок 3.14 – Скрипт прогнозування на основі навченої моделі Prophet

Приклад результатів прогнозування (у вигляді графіку) наведено на рисунку 3.15. Чорні точки – заміряні рівні засмічення повітря PM10, вертикальною червоною лінією позначено поточний момент, з якого починається прогнозування. Прогнозовані значення – синя лінія, синя область навколо лінії – межі похибки.

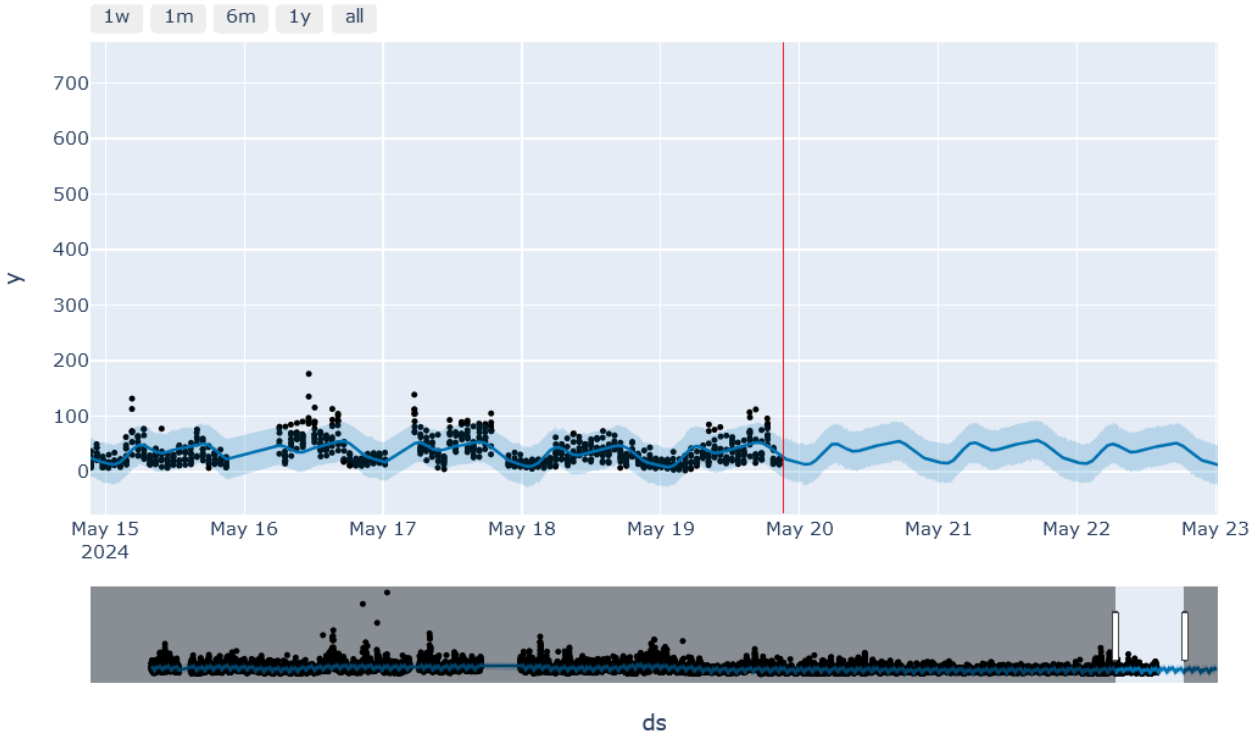


Рисунок 3.15 – Результат прогнозування значень PM10

Для оцінки точності моделі було використано метод крос-валідації, використовуючи модуль Prophet Diagnostics.

3.5 Опис методів представлення даних системи

3.5.1 Публічний API

Публічний API системи надає доступ до даних про якість повітря, радіаційний рівень, пожежі та джерела даних. API надає можливість отримувати записи застосовуючи фільтри та сортування, переглядати конкретні записи за їх ідентифікатором та отримувати “знімки” актуального стану (див. пункт 3.4.1). Документація надається у вигляді OpenAPI [41] специфікації як yaml-файл та у вигляді Swagger UI [42].

Приклад запиту на отримання записів з описом параметрів наведено на рисунку 3.16.

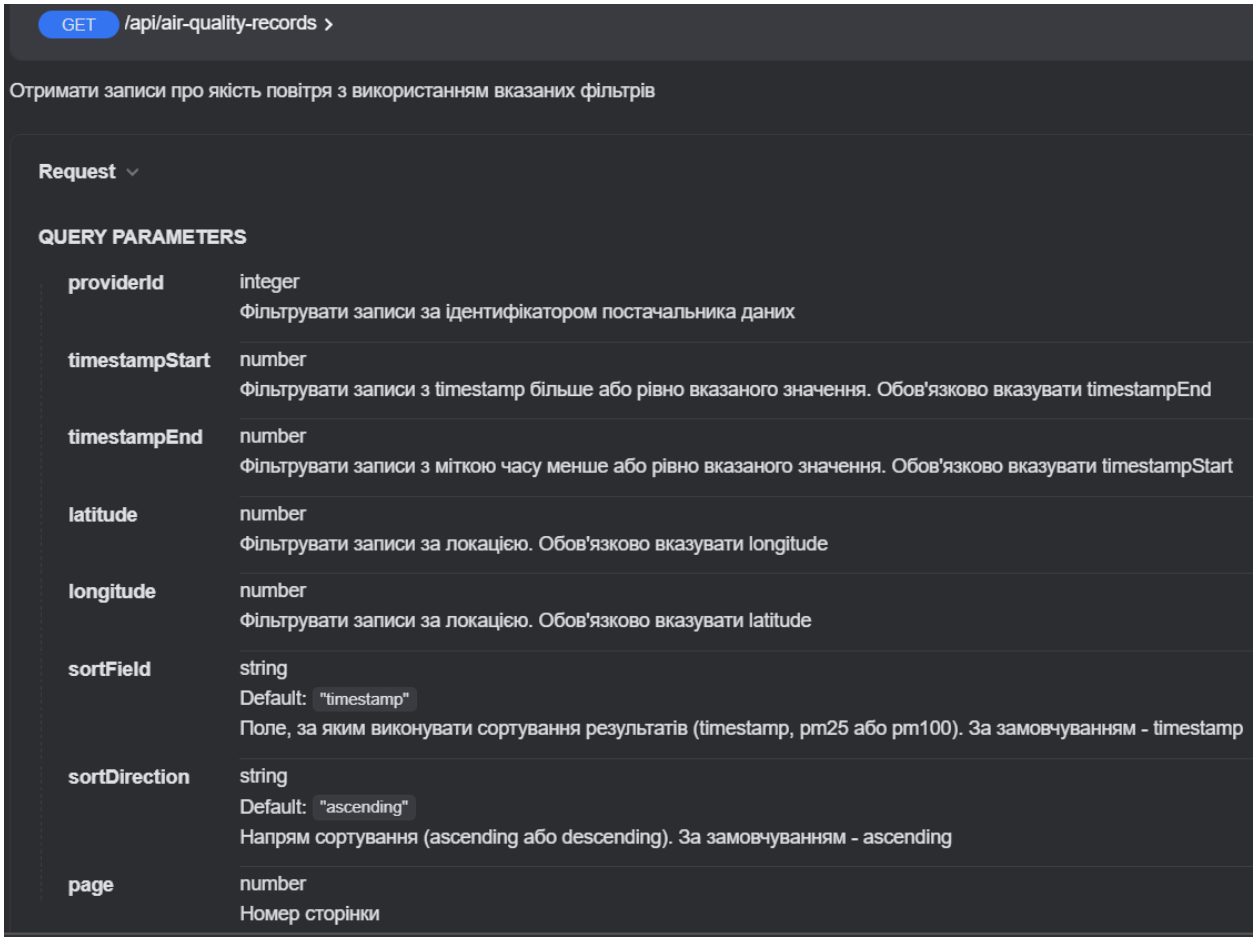


Рисунок 3.16 – Отримання записів про якість повітря

Також можна переглядати конкретні записи за їх ідентифікатором, що показано на рисунку 3.17.

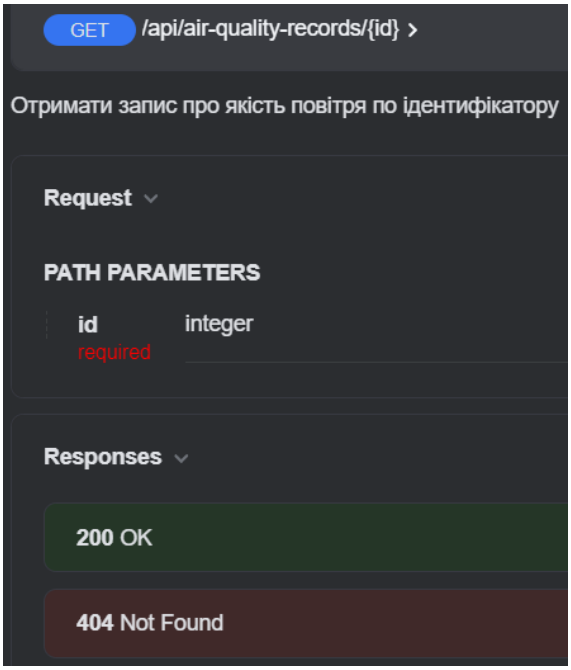


Рисунок 3.17 – Отримання запису про якість повітря за ідентифікатором

“Знімки” актуального на певний момент стану теж доступні через публічний API. Їх можна отримувати не тільки для поточного моменту, а й для часу в минулому, за допомогою опціонального параметру at. Регулювати максимальний вік записів, які потраплять до відповіді, можна за допомогою параметру maxAge. Детально цей алгоритм було розглянуто в підрозділі 3.4.1 даної роботи. Формат запиту наведено на рисунку 3.18.

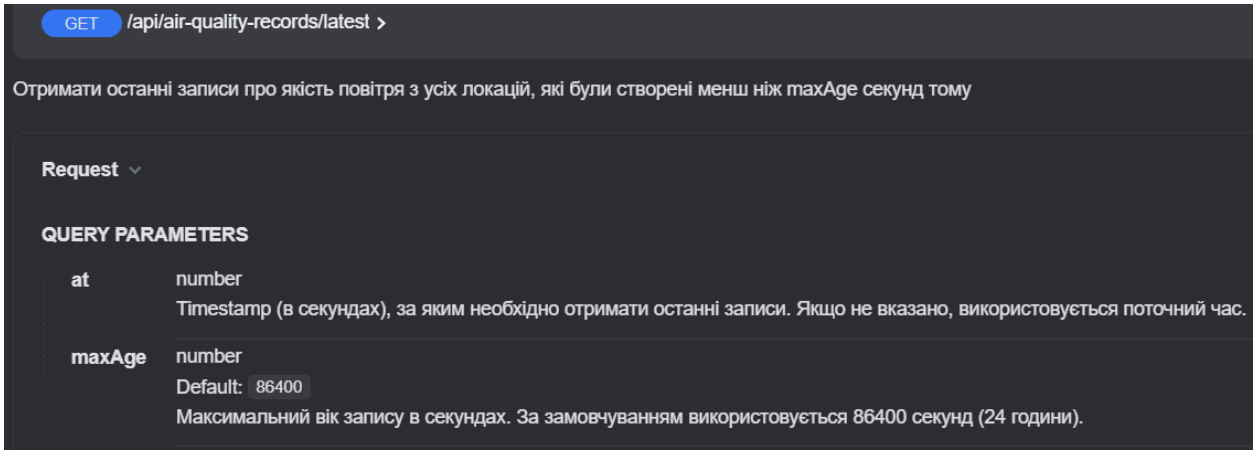


Рисунок 3.18 – Отримання “знімку” стану якості повітря

Аналогічні запити існують як для записів про радіаційний фон, так і про пожежі.

Дані про джерела даних (скрейпери) теж доступні через публічний API. Можна отримати відомості про всі джерела або про конкретне джерело за ідентифікатором.

3.5.2 Вебхук API

Вебхук API надає можливість отримувати дані в режимі реального часу за допомогою механізму вебхуків. Цей API дозволяє користувачам підписуватись на отримання повідомлень про нові записи, що надходять до системи, і автоматично отримувати ці дані через вказану callback URL-адресу. Якби цього механізму не існувало, то для отримання нових даних користувачам доводилося б постійно запитувати їх через публічний API (виконувати полінг). Порівняльну ілюстрацію цих механізмів наведено на рисунку 3.19.

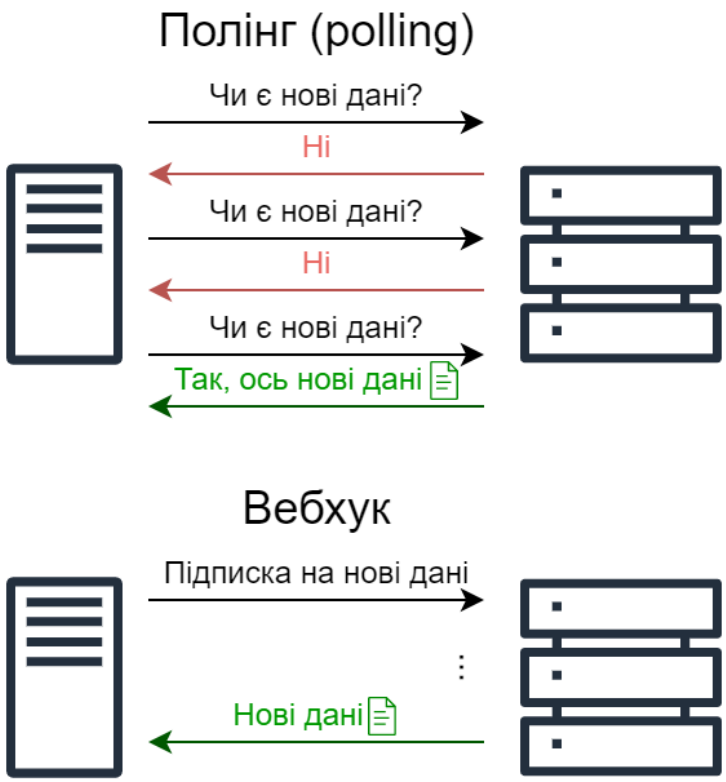


Рисунок 3.19 – Порівняння полінгу та вебхуків

Документація надається у вигляді OpenAPI [41] специфікації та Swagger UI [42], як і у випадку публічного API.

Всього цей API має 4 end-point-и:

1. Підписка. Це POST-запит, в тілі якого мають бути вказані координати місця, на яке користувач здійснює підписку, та callback URL, на який будуть надсилатись нові дані.
2. Перевірка підписок. Це GET-запит, який дозволяє користувачу отримати список підписок за callback URL.
3. Перевірка валідності callback URL. Це GET-запит, який дозволяє користувачу перевірити валідність переданого callback URL. Callback URL має повертати текст “UaEcoAggregator”, для того щоб пройти перевірку.
4. Відписка. Це POST-запит, призначений для відписки від отримання сповіщень через вебхук. Тіло запиту має бути таким же, яке було вказано при підписці.

3.5.3 Android-застосунок

У якості прикладу клієнта для публічного API (див. пункт 3.5.1) було розроблено Android-застосунок з двома основними функціями: інтерактивна карта та моніторинг.

За допомогою карти можна зручно отримувати інформацію про актуальний стан навколишнього середовища. За допомогою панелі налаштувань (показано на рисунку 3.20) можливо гнучко налаштувати типи та способи подання інформації, а також максимальній вік записів, які будуть відображені.

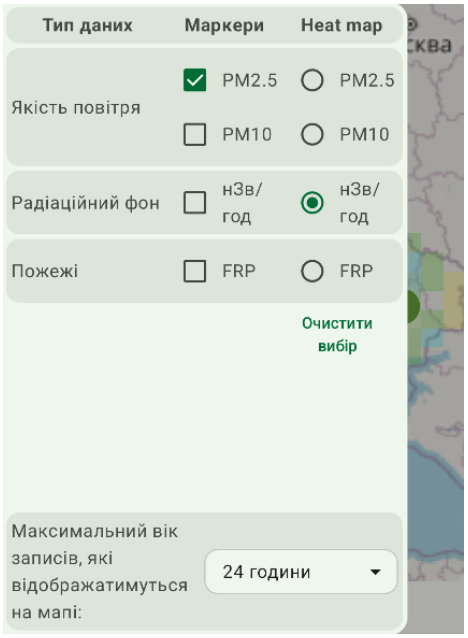


Рисунок 3.20 – Панель налаштувань карти

Реалізовано два способи подання інформації – маркери та heat map (теплова карта). На рисунку 3.21 продемонстровано обидва ці способи – значення PM2.5 у вигляді маркерів та теплова карта радіаційного фону.

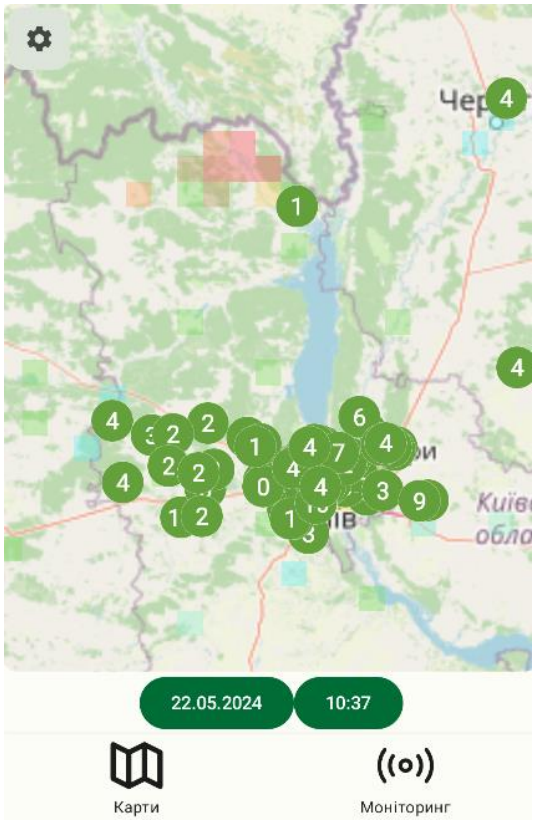


Рисунок 3.21 – Інтерактивна карта

Натиснувши на маркер, буде виконано перехід на сторінку запису. Цю сторінку показано на рисунку 3.22.

Рисунок 3.22 – Сторінка запису про якість повітря

На сторінці запису можна побачити останні виміряні показники, переглянути історичні дані за допомогою інтерактивного графіку, перейти на сторінку

постачальника даних (рисунок 3.23), та почати/зупинити моніторинг обраної локації.

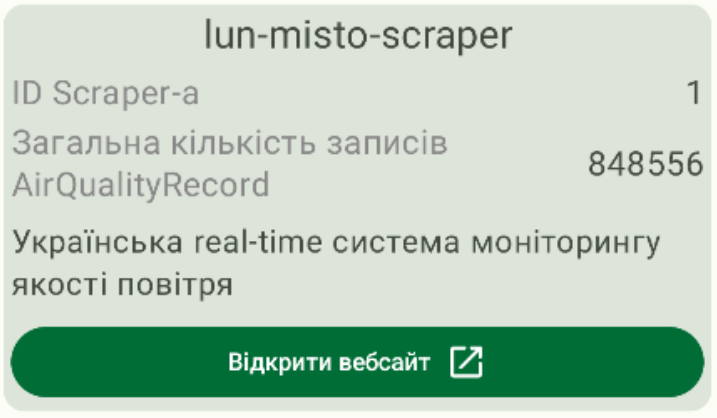


Рисунок 3.23 – Сторінка постачальника(джерела) даних

Сторінка джерела даних містить інформацію, таку як ідентифікатор, загальна кількість зроблених записів та стислий опис (українською або англійською мовами). З цієї сторінки також можна перейти на веб-сайт джерела.

Інша функція застосунку – моніторинг. Додавши локації до моніторингу, можна отримувати оновлення, які стосуються цих локацій, та переглядати останні актуальні дані.

Рисунок 3.23 – Сторінка моніторингу

3.5.4 Telegram-бот

У якості прикладу клієнта для вебхук API (див. пункт 3.5.2) було розроблено Telegram-бот. Бот надає зручний спосіб підписки на сповіщення про нові дані. Бот реалізовано на мові PHP, команди наведено в таблиці 3.1.

Таблиця 3.1 – Команди розробленого Telegram-боту

Команда	Опис
/subscribe <широта> <довгота>	Підписка на отримання сповіщень про зміни екологічних показників для вказаної локації.
/unsubscribe <широта> <довгота>	Відписка від отримання сповіщень про зміни екологічних показників для вказаної локації.
/subscriptions	Перегляд списку всіх активних підписок користувача.

Крім команд, якщо користувач надсилає боту свою локацію, бот, використовуючи публічний API системи (див. пункт 3.5.1), обробляє ці координати і повертає інформацію про п’ять найближчих станцій моніторингу якості повітря та п’ять найближчих станцій моніторингу радіаційного стану. Відповідь містить посилання на Google Maps [43] для кожної зі станцій, що дозволяє легко переглянути їх розташування.

Приклад сповіщення про нові дані наведено на рисунку 3.24.

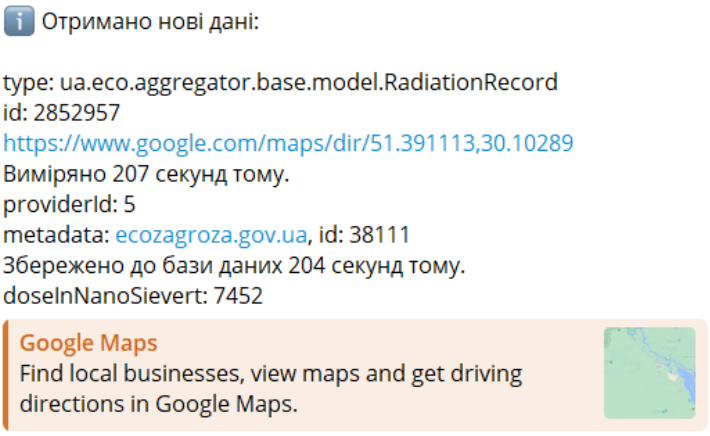


Рисунок 3.24 – Приклад сповіщення про нові дані від Telegram-боту

3.6 Опис розгортання системи

3.6.1 CI конвеєр на основі GitHub Actions

У цьому пункті описано впровадження конвеєра CI (безперервної інтеграції) для проекту, розробленого на основі GitHub Actions [44]. Цей конвеєр автоматизує процеси створення та публікації артефактів системи, тим самим забезпечуючи ефективну інтеграцію та доставку програмного забезпечення, уникаючи необхідності ручної роботи.

Файл конфігурації GitHub Actions workflow, названий “CI with Gradle”, визначає набір завдань, які виконуються при пуші до репозиторію тегів, що відповідають патерну v[0-9]+.[0-9]+.[0-9]+. Це гарантує, що конвеєр запускається лише для нових релізів.

Спрощений алгоритм роботи цього конвеєра наведено на рисунку 3.25.

Рисунок 3.25 – Спрощений алгоритм роботи GitHub Actions Workflow системи

3.6.2 Контейнеризація за допомогою Docker та Docker Compose

Docker [45] забезпечує ізольоване середовище для виконання програмного забезпечення. У розробленому проєкті Docker використовується для контейнеризації різних компонентів, що дозволяє забезпечити їхню портативність та легкість розгортання. Кожен компонент проєкту має свій власний Dockerfile, який визначає середовище виконання, залежності та інструкції для запуску програми. Приклад одного з Dockerfile-ів наведено на рисунку 3.26.

```
# OpenJDK 19 as the base image
FROM openjdk:19-jdk-slim

WORKDIR /app

# Download the latest release of PublicAPI.jar from GitHub
ADD https://github.com/oleksandrvolovyk/ua-eco-aggregator-public/releases/latest/download/PublicAPI.jar /app/PublicAPI.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "PublicAPI.jar"]
```

Рисунок 3.26 – Приклад одного з Dockerfile-ів системи

У цьому Dockerfile використовується образ OpenJDK 19 як базовий. Далі визначається робоча директорію /app та завантажується остання версія PublicAPI.jar з GitHub. Порт 8080 виставляється для доступу до застосунку, а команда ENTRYPOINT визначає, що саме буде виконано при запуску контейнера.

Для зручності розгортання та управління кількома контейнерами було використано Docker Compose [46]. Docker Compose дозволяє визначити всі сервіси, їхні залежності, змінні середовища та налаштування мережі в одному файлі compose.yaml. На рисунку 3.27 наведено фрагмент файлу compose.yaml.

У цьому файлі визначено кілька сервісів, кожен з яких відповідає за певний компонент системи. Кожен сервіс містить:

- build: вказує директорію з Dockerfile для створення образу.
- ports: визначає, які порти контейнера будуть відкриті на хості.
- environment: змінні середовища, необхідні для роботи сервісу.
- depends_on: визначає залежності між сервісами, що забезпечує коректний порядок їх запуску.
- restart: визначає політику перезапуску контейнера в разі його зупинки.

Docker Compose дозволяє легко керувати усіма цими сервісами, використовуючи прості команди, такі як docker-compose up для запуску всіх сервісів або docker-compose down для їх зупинки.

```yaml
services:
  scraper-api:
    build: ./ScraperAPI/.
    ports:
      - "8080:8080"
    environment:
      DB_CONNECTION_STRING: ""
      DB_USER: ""
      DB_PASS: ""
    restart: always

  lun-misto-scraper:
    build: ./LunMistoScraper/.
    depends_on:
      - scraper-api
    environment:
      SCRAPING_API_URL: "http://scraper-api:8080/air-quality-records"
      SCRAPING_API_KEY: ""
      POLLING_DELAY_IN_SECONDS: 3600 # every 60 minutes
    restart: always

  nasa-firms-scraper:
    build: ./NasaFirmsScraper/.
    depends_on:
      - scraper-api
    environment:
      SCRAPING_API_URL: "http://scraper-api:8080/fire-records"
      SCRAPING_API_KEY: ""
      FIRMS_API_KEY: ""
      POLLING_DELAY_IN_SECONDS: 10800 # every 3 hours
    restart: always
```

Рисунок 3.27 – Фрагмент файлу compose.yaml системи

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі даної дипломної роботи здійснено опис архітектури і компонентів розробленої системи збору та аналізу екологічних даних.

Було розглянуто архітектуру розробленої системи, яка наведена на рисунку 3.1. Здійснено порівняння підходів, використаних в проектуванні архітектури, з підходами, розглянутими в підрозділі 2.4 даної роботи. Було досліджено поєднання монолітного підходу, що передбачає централізовану базу даних, та підходу мікросервісів, що забезпечує незалежність і масштабованість компонентів системи.

Здійснено опис збирачів (скрейперів) даних системи, включаючи короткий опис джерела, метод збору даних, а також використані мови програмування, фреймворки та бібліотеки.

Обґрунтовано вибір бази даних проєкту. Роз'яснено схему бази даних, яка наведена в додатку 2. В пунктах 3.3.3 та 3.3.4 розглянуто особливості роботи з базою даних системи – застосування кешування та пулу з'єднань.

Описано реалізовані аналітичні інструменти системи: створення "знімків" екологічного стану на основі актуальних даних та прогнозування забруднення повітря за допомогою інструменту Prophet, що дозволяє передбачати концентрації забруднюючих речовин на основі часових рядів.

Здійснено опис реалізованих методів представлення даних системи: публічного API для доступу до даних, вебхук API для отримання даних у режимі реального часу, а також клієнтських застосунків для Android та Telegram, що забезпечують зручний інтерфейс для користувачів.

Розглянуто процес розгортання системи: впровадження CI конвеєра на основі GitHub Actions для автоматизації збірки та випуску нових версій програмного забезпечення, а також використання Docker і Docker Compose для контейнеризації компонентів системи, що забезпечує їхню портативність та легкість управління.

РОЗДІЛ 4
РЕЗУЛЬТАТИ ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ ЗБОРУ ТА АНАЛІЗУ ЕКОЛОГІЧНИХ ДАНИХ

4.1 Результати тестування збирачів (скрейперів) даних системи

4.1.1 Український гідрометеорологічний центр

Тести для збирача (скрейпера) даних про радіологічний стан з опублікованих вимірювань Українського гідрометеорологічного центру [25] поділені на два файли: MainKtTest.kt, де здійснюється тестування основного функціоналу (виконання HTTP запиту, отримання відповіді, перетворення відповіді у формат системи та надсилання результатів) та MeteoDataDeserializerTest, де здійснюється тестування серіалізації/десеріалізації JSON. Результати тестування наведено на рисунку 4.1.

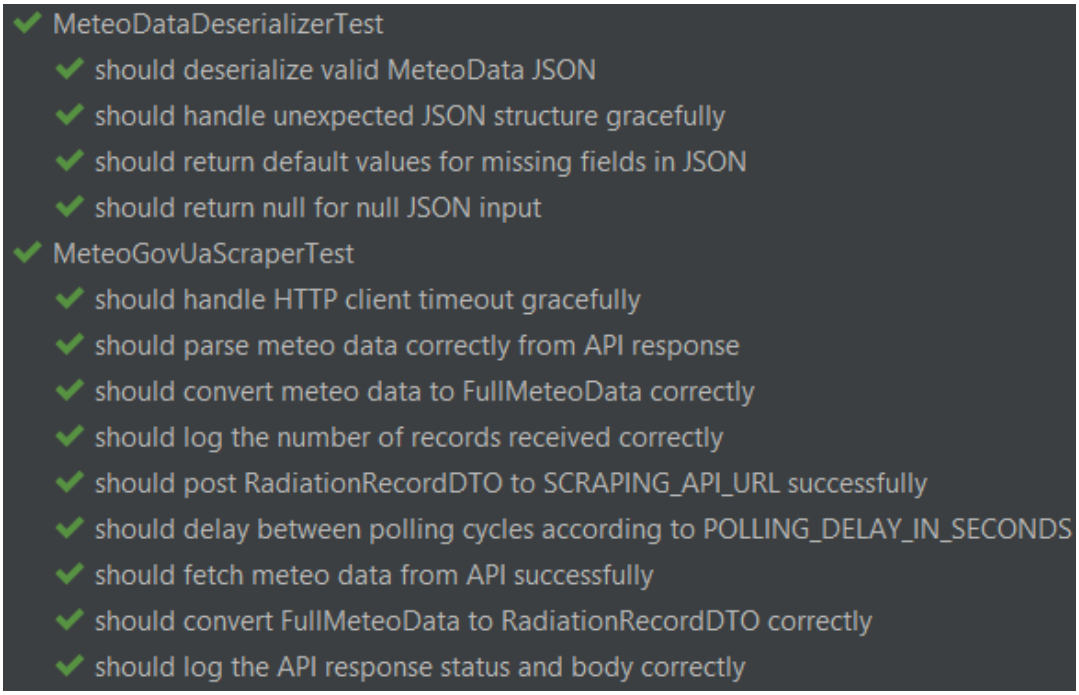


Рисунок 4.1 – Результати тестування збирача (скрейпера) даних з джерела “Український гідрометеорологічний центр”

4.1.2 ЕкоЗагроза

Тести для збирача (скрейпера) даних про радіологічний стан з опублікованих вимірювань ресурсу “ЕкоЗагроза” [29] розміщені у файлі EcoZagrozaGovUaTest.kt, де здійснюється тестування основного функціоналу (виконання HTTP запиту, отримання відповіді, перетворення відповіді у формат системи та надсилання результатів). Результати тестування наведено на рисунку 4.2.

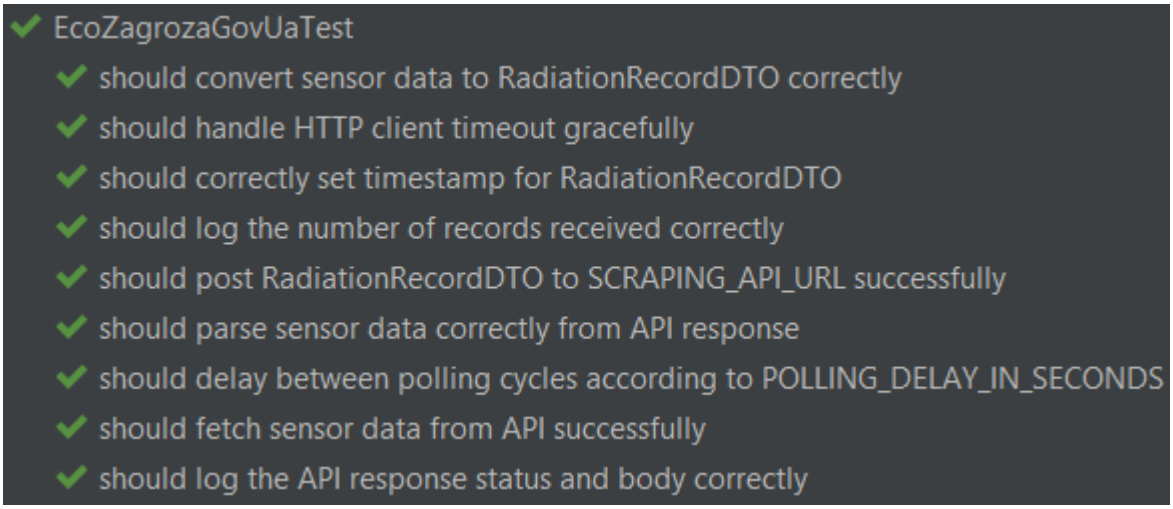


Рисунок 4.2 – Результати тестування збирача (скрейпера) даних з джерела “ЕкоЗагроза”

4.1.3 ЛУН Місто Air

Тести для збирача (скрейпера) даних про якість повітря з опублікованих вимірювань системи “ЛУН Місто Air” [29] розміщені у файлі LunMistoScraperTest.kt, де здійснюється тестування основного функціоналу (виконання HTTP запиту, отримання відповіді, перетворення відповіді у формат системи та надсилання результатів). Результати тестування наведено на рисунку 4.3.

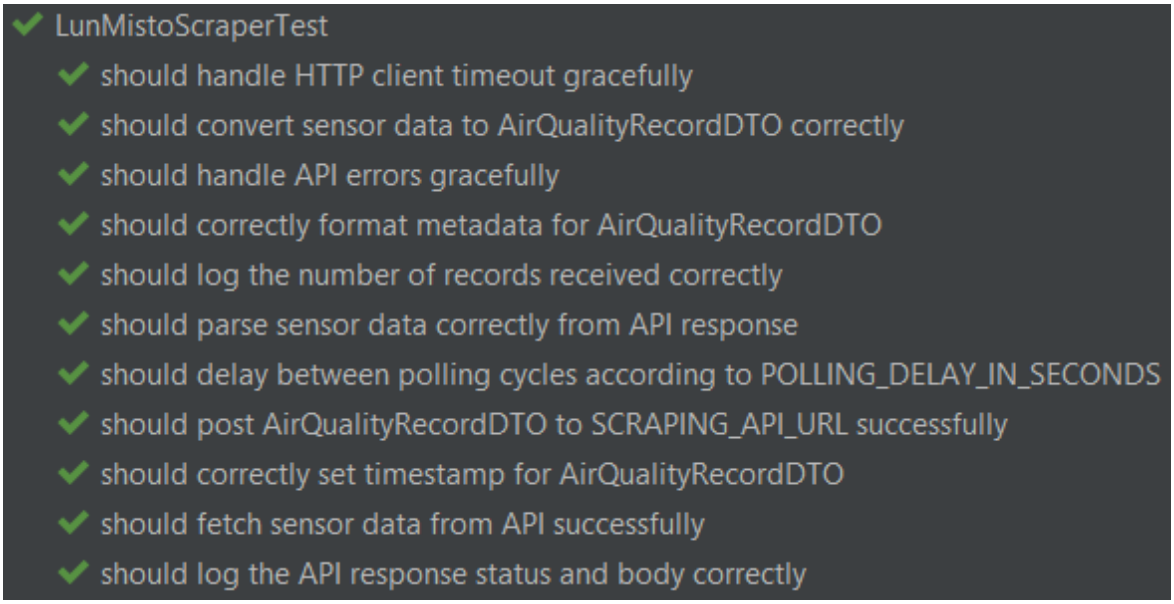


Рисунок 4.3 – Результати тестування збирача (скрейпера) даних з джерела “ЛУН Місто Air”

4.1.4 SaveDnipro

Тести для збирача (скрейпера) даних про якість повітря з опублікованих вимірювань організації “SaveDnipro” [30] розміщені у файлі SaveDniproScraperTest.kt, де здійснюється тестування основного функціоналу (виконання HTTP запиту, отримання відповіді, перетворення відповіді у формат системи та надсилання результатів). Результати тестування наведено на рисунку 4.4.

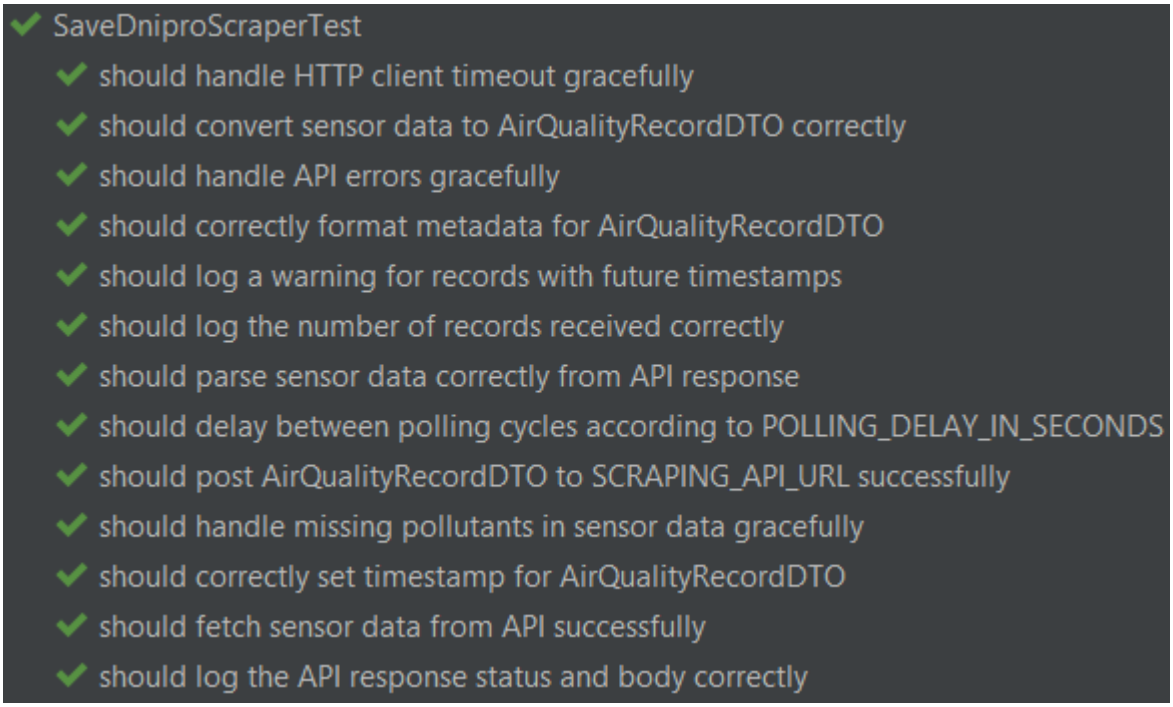


Рисунок 4.4 – Результати тестування збирача (скрейпера) даних з джерела “SaveDnipro”

4.1.5 Sensor.Community

Тести для збирача (скрейпера) даних про якість повітря з опублікованих вимірювань проекту “Sensor.Community” [31] розміщені у файлі SensorCommunityScraperTest.kt, де здійснюється тестування основного функціоналу (виконання HTTP запиту, отримання відповіді, перетворення відповіді у формат системи та надсилання результатів). Результати тестування наведено на рисунку 4.5.

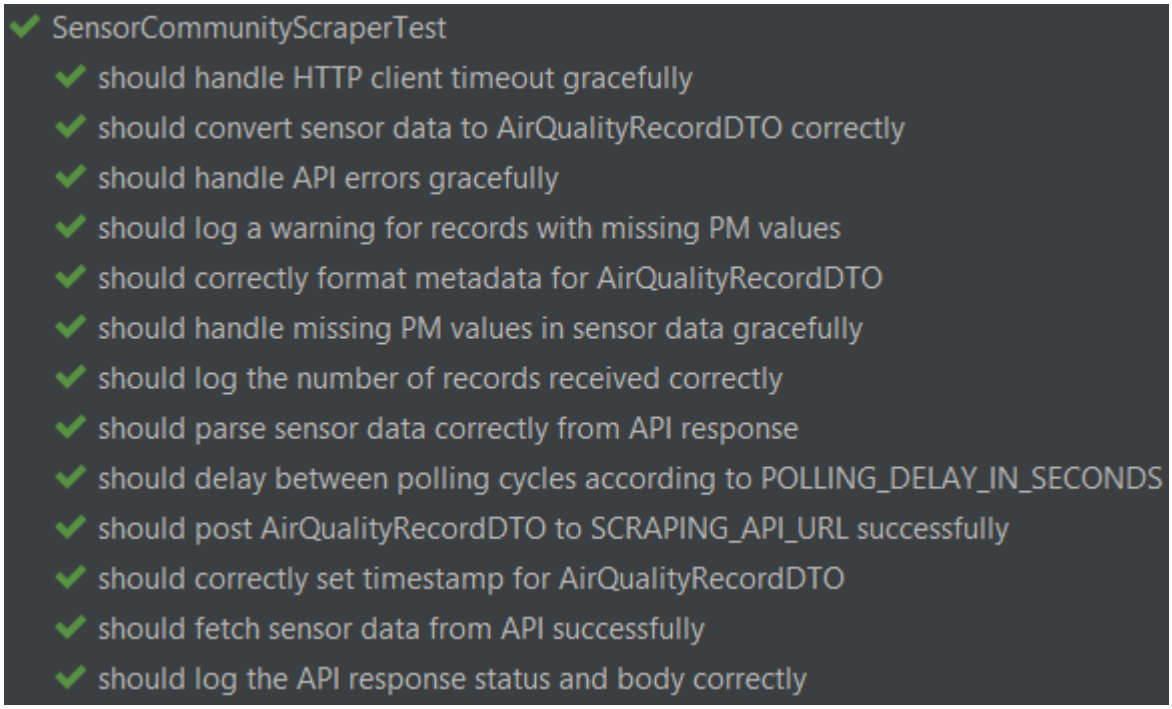


Рисунок 4.5 – Результати тестування збирача (скрейпера) даних з джерела “Sensor.Community”

4.1.6 NASA FIRMS

Тести для збирача (скрейпера) даних про пожежі з опублікованих вимірювань системи “NASA FIRMS” [32] розміщені у файлі NasaFirmsScraperTest.kt, де здійснюється тестування основного функціоналу (виконання HTTP запиту, отримання відповіді, перетворення відповіді у формат системи та надсилання результатів). Результати тестування наведено на рисунку 4.6.

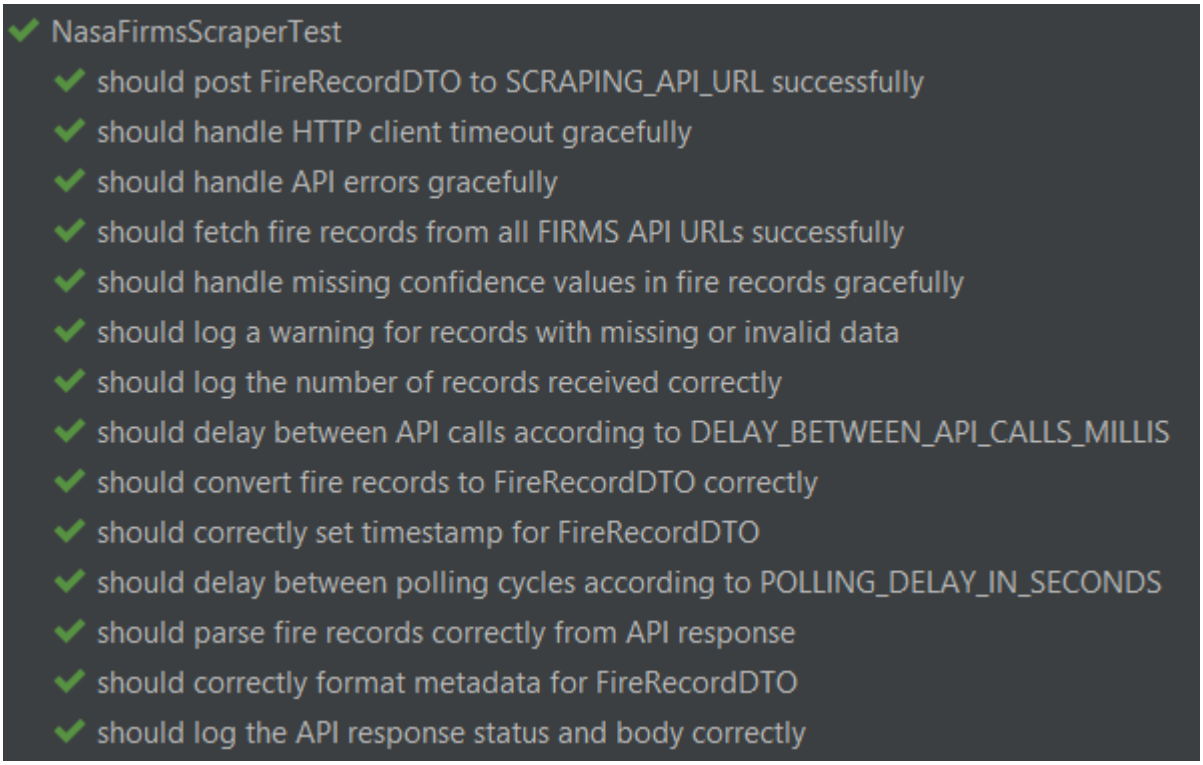


Рисунок 4.6 – Результати тестування збирача (скрейпера) даних з джерела “NASA FIRMS”

4.2 Результати тестування API системи

4.2.1 Публічний API

Було розроблено та виконано серію тестів для забезпечення коректної роботи різних аспектів публічного API, включаючи перевірку обробки запитів з різними параметрами, обробку помилок, сортування та фільтрацію даних. Тести охоплюють дві основні частини публічного API: RecordPublicAPITest (частина, що відповідає за агреговані записи) та ScraperPublicAPITest (частина, що відповідає за збирачі (скрейпери) даних).

Набір тестів RecordPublicAPITest перевіряє функціональність API для роботи з записами:

- Перевірка обробки обов'язкових параметрів. Тести гарантують, що при відсутності обов'язкових параметрів timestampStart, timestampEnd, latitude, longitude API повертає відповідні помилки (код 400).

- Фільтрація записів. Перевірено, що API коректно обробляє запити з фільтрами за широтою і довготою, timestampStart і timestampEnd, а також providerId.
- Обробка помилок. Тести перевіряють, що при виникненні несподіваної помилки сервер повертає код 500.
- Сортування та пагінація. Тести перевіряють правильність сортування записів за полями даних та часом, а також коректну роботу механізму пагінації.
- Отримання записів за ID. Перевірено, що API правильно повертає записи за їх ідентифікатором або повідомляє про їх відсутність (код 404).

Набір тестів ScraperPublicAPITest перевіряє функціональність API для роботи зі збирачами (скрейперами) даних:

- Отримання скрейперів за ID. Перевірено, що API коректно повертає дані про скрейпери за їх ідентифікатором або повідомляє про їх відсутність (код 404).
- Обробка помилок. Тести гарантують, що при виникненні несподіваної помилки сервер повертає код 500.
- Отримання всіх скрейперів. Перевірено, що API правильно повертає список усіх скрейперів.

Результати тестування наведено на рисунку 4.7.

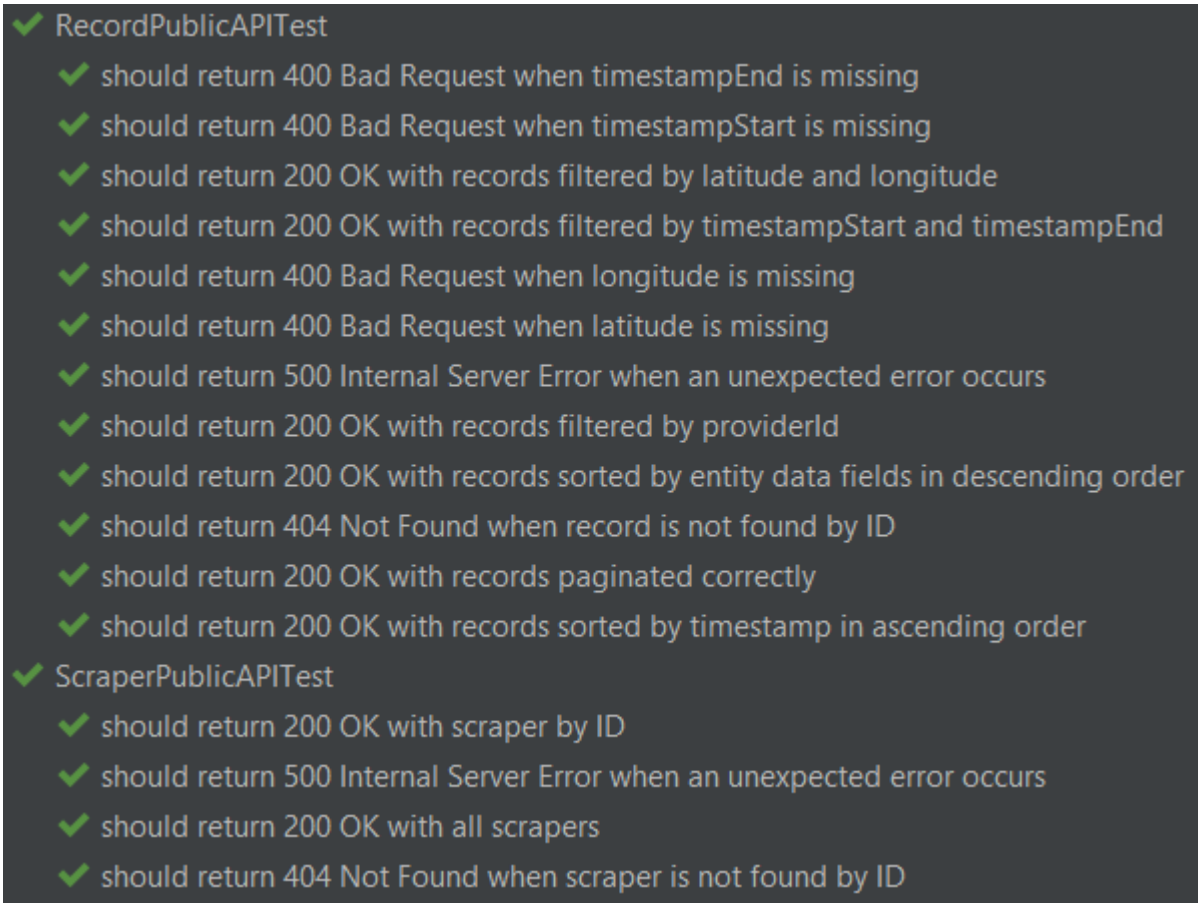


Рисунок 4.7 – Результати тестування публічного API системи

4.2.2 Вебхук API

Цей підрозділ описує результати тестування вебхук API системи. Тести були розроблені для перевірки коректності роботи механізмів підписки та відписки, валідації callback URL, а також обробки запитів з некоректними параметрами.

Набір тестів WebhookAPITest перевіряє функціональність вебхук API:

- Успішна підписка. Перевіряється, що API відповідає кодом 200 OK та повідомленням "Webhook added!" при успішній підписці.
- Некоректний callback URL при тестуванні. Перевіряється, що API відповідає кодом 400 Bad Request, якщо callback URL є некоректним під час тестування.
- Некоректний callback URL при підписці. Перевіряється, що API відповідає кодом 400 Bad Request, якщо callback URL є некоректним під час підписки.

- Некоректний callback URL при відписці. Перевіряється, що API відповідає кодом 400 Bad Request, якщо callback URL є некоректним під час відписки.
- Отримання списку підписок. Перевіряється, що API відповідає кодом 200 OK та надає список підписок для заданого URL.
- Успішна відписка. Перевіряється, що API відповідає кодом 200 OK та повідомленням "Webhook removed!" при успішній відписці.
- Успішне тестування валідності callback URL. Перевіряється, що API відповідає кодом 200 OK та повідомленням "Callback URL is valid!" при успішному тестуванні.

Результати тестування наведено на рисунку 4.8.

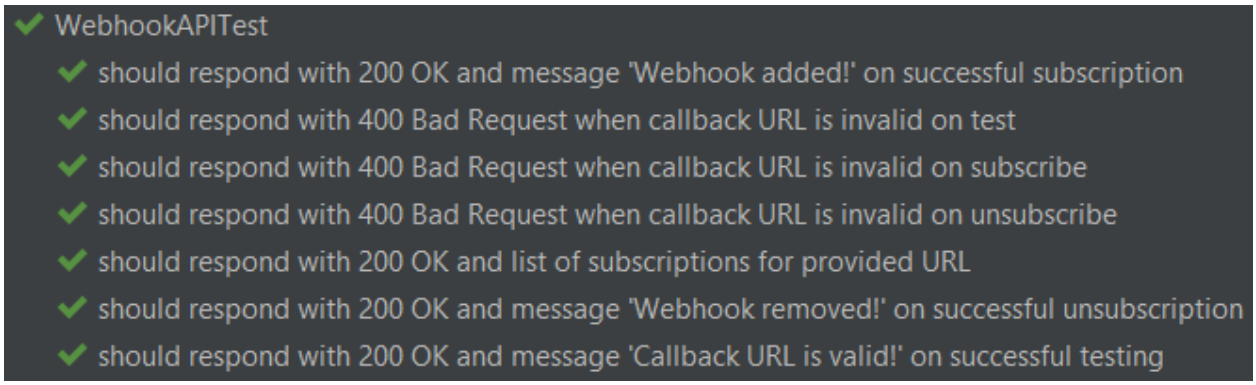


Рисунок 4.8 – Результати тестування вебхук API системи

4.3 Результати тестування Android-застосунку системи

4.3.1 Панель налаштувань карти

Тестування панелі налаштувань карти перевіряє функціональність панелі налаштувань карти в Android-застосунку.

1. setMapType_shouldUpdateMapType: Перевіряє, чи оновлюється тип карти при встановленні нового значення.
2. getMapType_shouldReturnCurrentMapType: Перевіряє, чи повертається поточний тип карти.
3. setMaxAge_shouldUpdateMaxAge: Перевіряє, чи оновлюється максимальний вік записів при встановленні нового значення.

4. getMaxAge_shouldReturnCurrentMaxAge: Перевіряє, чи повертається поточний максимальний вік записів.
5. setInformationType_shouldUpdateInformationType: Перевіряє, чи оновлюється тип інформації при встановленні нового значення.
6. getInformationType_shouldReturnCurrentInformationType: Перевіряє, чи повертається поточний тип інформації.

Результати наведено на рисунку 4.9.

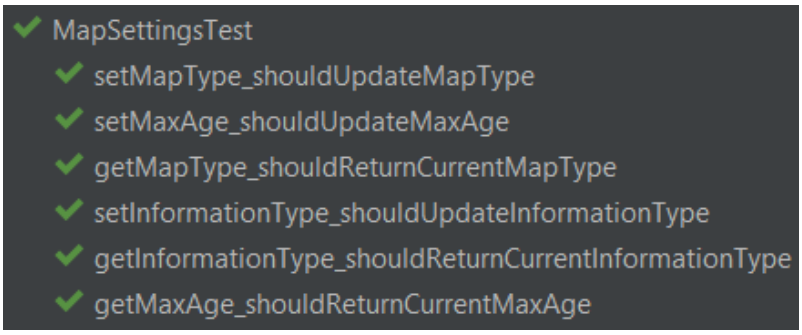


Рисунок 4.9 – Результати тестування панелі налаштувань карти Android-застосунку системи

4.3.2 Інтерактивна карта

Тестування інтерактивної карти перевіряє функціональність інтерактивної карти, яка дозволяє користувачам бачити актуальні дані про стан навколишнього середовища. Включає перевірки відображення маркерів та теплової карти, а також навігацію при натисканні на елементи карти.

1. displayMarkers_shouldShowMarkersOnMap: Перевіряє, чи відображаються маркери на карті.
2. displayHeatMap_shouldShowHeatMapOnMap: Перевіряє, чи відображається теплова карта на карті.
3. markerClick_shouldNavigateToRecordPage: Перевіряє, чи відбувається навігація на сторінку запису при натисканні на маркер.
4. heatMapClick_shouldNavigateToRecordPage: Перевіряє, чи відбувається навігація на сторінку запису при натисканні на теплову карту.

Результати наведено на рисунку 4.10.

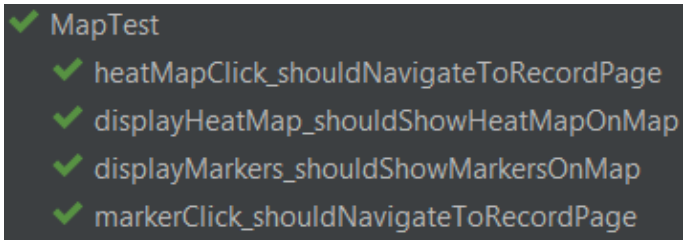


Рисунок 4.10 – Результати тестування карти Android-застосунку системи

4.3.3 Моніторинг

Тестування сторінки моніторингу перевіряє функціональність сторінки моніторингу, яка дозволяє користувачам додавати локації до моніторингу та отримувати оновлення, що стосуються цих локацій.

1. addLocationToMonitoring_shouldAddLocation: Перевіряє, чи додається локація до моніторингу.
2. removeLocationFromMonitoring_shouldRemoveLocation: Перевіряє, чи видаляється локація з моніторингу.
3. getMonitoredLocations_shouldReturnListOfLocations: Перевіряє, чи повертається список всіх локацій, які моніторяться.
4. receiveUpdates_shouldUpdateMonitoredData: Перевіряє, чи оновлюються дані моніторингу при отриманні нових даних.

Результати тестування наведено на рисунку 4.11.

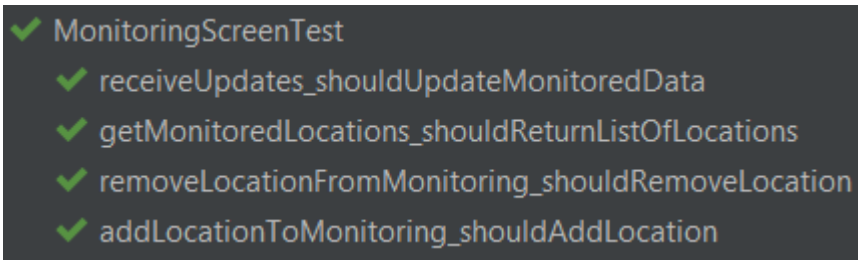


Рисунок 4.11 – Результати тестування моніторингу Android-застосунку системи

4.3.4 Сторінка запису

Тестування сторінки запису перевіряє функціональність сторінки запису, де користувачі можуть переглядати деталі конкретних записів, такі як останні виміряні показники, історичні дані та інформацію про постачальника даних.

1. loadRecord_shouldDisplayRecordDetails: Перевіряє, чи відображаються деталі запису при завантаженні сторінки.
2. displayHistoricalData_shouldShowGraph: Перевіряє, чи відображається інтерактивний графік з історичними даними.
3. navigateToDataProviderPage_shouldShowProviderDetails: Перевіряє, чи відбувається навігація на сторінку постачальника даних.
4. startMonitoring_shouldBeginMonitoringLocation: Перевіряє, чи починається моніторинг вибраної локації.
5. stopMonitoring_shouldEndMonitoringLocation: Перевіряє, чи зупиняється моніторинг вибраної локації.

Результати тестування наведено на рисунку 4.12.

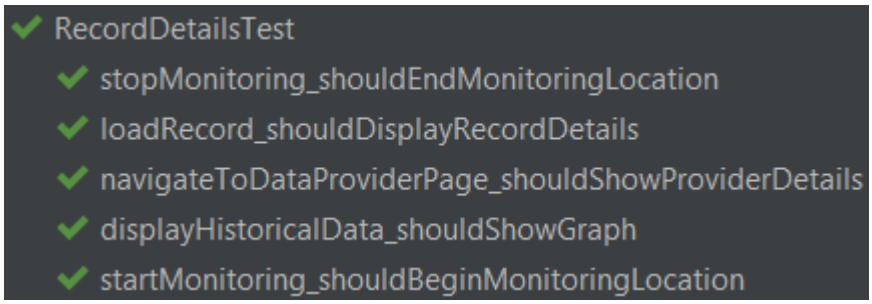


Рисунок 4.12 – Результати тестування сторінки запису Android-застосунку системи

4.3.5 Сторінка постачальника (джерела) даних

Тестування сторінки постачальника даних перевіряє функціональність сторінки постачальника даних, яка містить інформацію про джерело даних, такі як ідентифікатор, загальна кількість записів та опис.

1. loadProviderDetails_shouldDisplayProviderInfo: Перевіряє, чи відображається інформація про постачальника даних.
2. navigateToProviderWebsite_shouldOpenInBrowser: Перевіряє, чи відкривається веб-сайт постачальника даних у браузері.
3. getRecordCount_shouldReturnTotalRecords: Перевіряє, чи повертається загальна кількість зроблених записів.

Результати тестування наведено на рисунку 4.13.

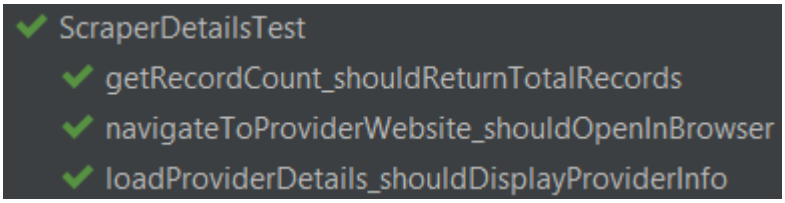


Рисунок 4.13 – Результати тестування сторінки постачальника (джерела) даних Android-застосунку системи

4.4 Результати тестування Telegram-боту системи

4.4.1 Підписка

Тестування полягає в надсиланні команди підписки з парою координат. Результати тестування наведено на рисунку 4.14.

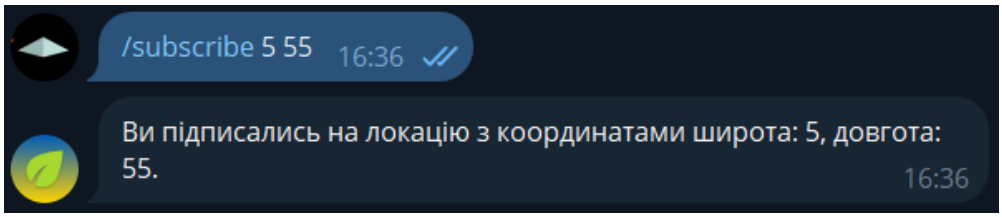


Рисунок 4.14 – Результати тестування підписки Telegram-боту системи

4.4.2 Відписка

Тестування полягає в надсиланні команди відписки з парою координат, на які раніше було здійснено підписку. Результати тестування наведено на рисунку 4.15.

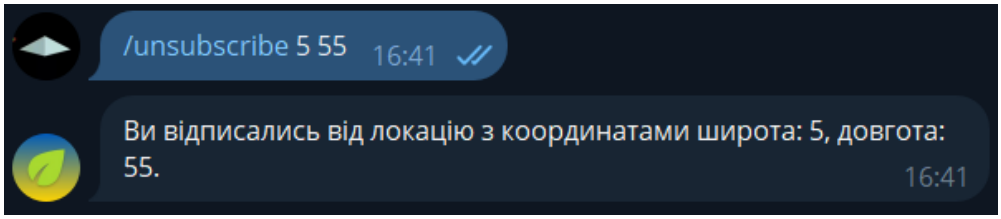


Рисунок 4.15 – Результати тестування відписки Telegram-боту системи

4.4.3 Перегляд списку підписок

Тестування полягає в надсиланні команди перегляду списку підписок. Результати тестування наведено на рисунку 4.16.

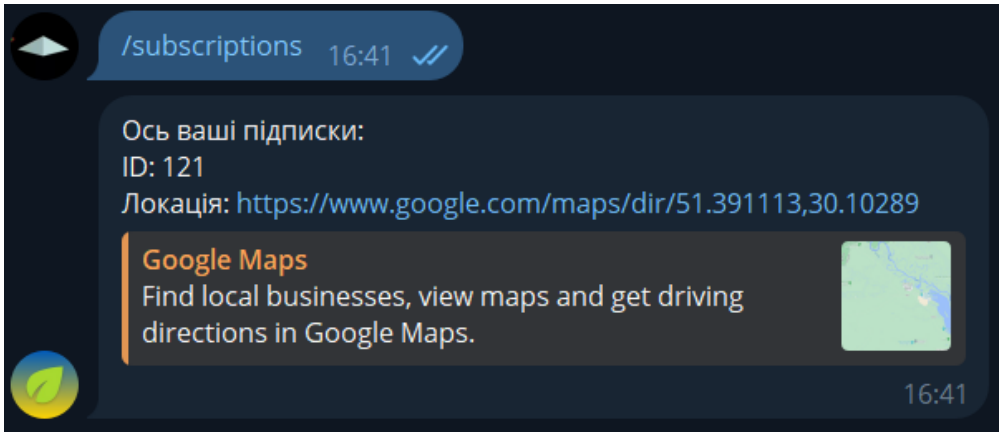


Рисунок 4.16 – Результати тестування перегляду списку підписок Telegram-боту системи

4.4.4 Отримання сповіщень

Тестування полягає в підписці та подальшому очікуванні отримання сповіщень про надходження до системи нових даних. Результати тестування наведено на рисунку 4.17.

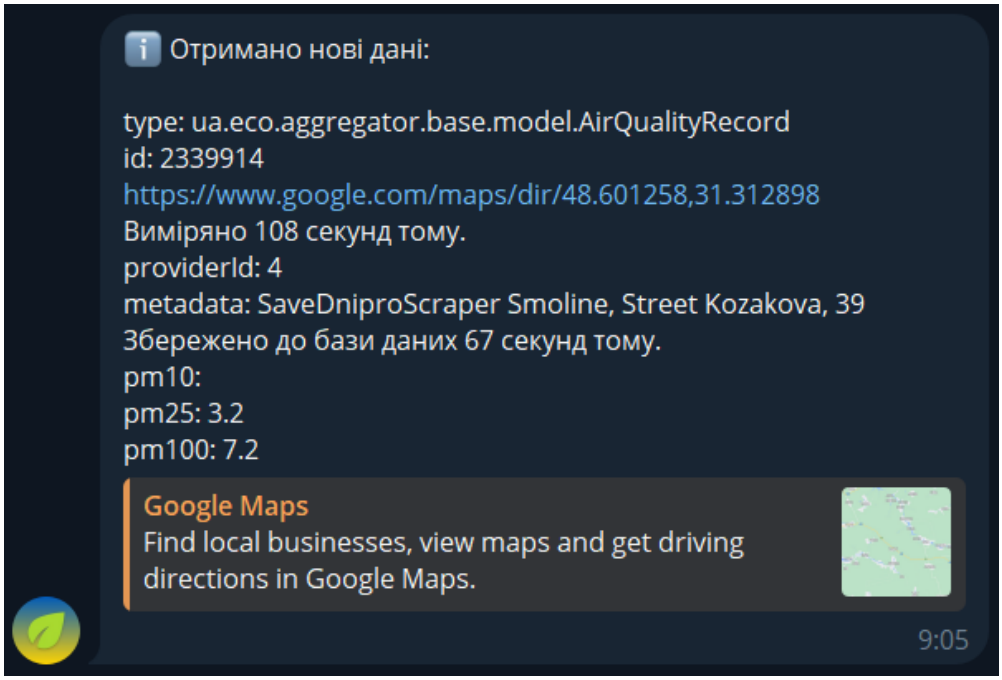


Рисунок 4.17 – Результати тестування отримання сповіщень Telegram-боту системи

ВИСНОВОК ДО РОЗДІЛУ 4

У цьому розділі дипломної роботи було представлено результати тестування розробленої системи збору та аналізу екологічних даних. Тестування охоплювало збирачі (скрейпери) даних, API системи, Android-застосунок та Telegram-бот.

Збирачі (скрейпери) даних успішно пройшли тестування. Було підтверджено, що вони збирають дані з джерел коректно та передають їх до системи.

API системи також успішно пройшли тестування. Було підтверджено, що публічний API надає доступ до даних у різних форматах, а вебхук API забезпечує механізми підписки на оновлення даних у режимі реального часу.

Android-застосунок успішно пройшов тестування. Було підтверджено, що він коректно відображає карту з даними про стан навколишнього середовища, дозволяє користувачам додавати локації до моніторингу та отримувати оновлення, а також пропонує детальну інформацію про окремі записи.

Telegram-бот успішно пройшов тестування. Було підтверджено, що він дозволяє користувачам підписуватися на оновлення даних про стан навколишнього середовища в конкретних локаціях, відписуватися від розсилок, переглядати список підписок та отримувати сповіщення про нові дані.

Загалом, результати тестування підтвердили, що розроблена система збору та аналізу екологічних даних є коректною, функціональною та зручною для користувачів.

ВИСНОВОК

В результаті виконаної дипломної роботи було розроблено агрегатор публічних даних про екологічний стан навколишнього середовища України, що включає збір, зберігання, аналіз та представлення екологічних даних у зручному форматі для користувачів.

У першому розділі було обґрунтовано актуальність розробки системи, що збирає та аналізує дані про екологічний стан України. Було проведено аналіз існуючих аналогічних систем, визначено їхні переваги та недоліки, а також окреслено основні проблеми та можливі шляхи їх вирішення. В результаті, було встановлено, що розроблювана система повинна забезпечувати зручний доступ до даних, мати механізми оцінки їхньої надійності, бути легкою у розширенні та стабільно функціонувати в умовах зростання обсягів даних.

Другий розділ був присвячений дослідженню технологій та підходів для реалізації системи. Розглянуто методи збору даних, включаючи відкриті API та вебскрейпінг, а також проаналізовано системи зберігання та обробки даних. Було оцінено методи статистичного аналізу та машинного навчання для аналізу екологічних даних. Також розглянуто архітектурні підходи для побудови масштабованої системи та методи представлення даних через різні інтерфейси, такі як веб-інтерфейси, мобільні додатки, соціальні мережі та API.

Третій розділ містив опис архітектури та компонентів розробленої системи. Було поєднано монолітний та мікросервісний підходи для досягнення незалежності та масштабованості компонентів. Описано збирачі даних, методи роботи з базою даних, кешування та використання пулу з'єднань. Було реалізовано аналітичні інструменти для створення "знімків" екологічного стану та прогнозування забруднення повітря. Також було описано методи представлення даних через публічний та вебхук API, а також клієнтські застосунки для Android та Telegram. Процес розгортання системи включав впровадження CI конвеєра та контейнеризацію компонентів за допомогою Docker та Docker Compose.

У четвертому розділі були представлені результати тестування системи. Тестування охоплювало збирачі даних, API, Android-застосунок та Telegram-бот. Всі компоненти успішно пройшли тестування, підтвердивши коректність збору та передачі даних, роботу API, функціональність мобільного застосунку та Telegram-бота. Загальні результати тестування засвідчили, що система є коректною, функціональною та зручною для користувачів.

Таким чином, дипломна робота демонструє успішну розробку комплексної системи збору та аналізу екологічних даних, що відповідає сучасним вимогам до таких систем та надає корисний інструмент для моніторингу стану навколишнього середовища в Україні.

Технічне завдання на дипломний бакалаврський проєкт виконано повністю.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Bairoch, P. International Industrialization Levels from 1750 to 1980 / P. Bairoch // Journal of European Economic History. — 1982. — Vol. 11, No. 2. — P. 269-333.
2. Petrov, M., Nikolaeva, Z., Dimitrov, A. The impact of anthropogenic activity on the global environment / M. Petrov, Z. Nikolaeva, A. Dimitrov // INTERNATIONAL SCIENTIFIC JOURNAL "SCIENCE. BUSINESS. SOCIETY". — 2023. — Vol. 8, No. 2. — P. 59-64.
3. Solomon, S., Plattner, G.-K., Knutti, R., Friedlingstein, P. Irreversible climate change due to carbon dioxide emissions / S. Solomon, G.-K. Plattner, R. Knutti, P. Friedlingstein // Proceedings of the National Academy of Sciences. — 2009. — Vol. 106, No. 6. — P. 1704-1709.
4. SaveEcoBot. SaveEcoBot — https://www.saveecobot.com.
5. ЛУН. ЛУН Місто Air — https://misto.lun.ua/air.
6. WAQI. About the World Air Quality Index project — https://aqicn.org/contact/.
7. ISO (International Organization for Standardization). ISO 14001:2015 — https://www.iso.org/standard/60857.html.
8. Open Geospatial Consortium — https://www.ogc.org/.
9. CF Community. CF Metadata Conventions — https://cfconventions.org/.
10. Open Geospatial Consortium. Sensor Model Language (SensorML) — https://www.ogc.org/standard/sensorml/.
11. The Apache Software Foundation. Apache Hadoop — https://hadoop.apache.org/.
12. The Apache Software Foundation. Apache Cassandra — https://cassandra.apache.org/_/index.html.
13. solidIT consulting & software development gmbh. DB-Engines Ranking — https://db-engines.com/en/ranking.
14. Oracle. Oracle — https://www.oracle.com/ua/.

15. MongoDB, Inc. JSON and BSON — https://www.mongodb.com/json-and-bson.
16. ScyllaDB, Inc. Cassandra, ACID and BASE — https://www.scylladb.com/learn/apache-cassandra/introduction-to-apache-cassandra/.
17. Elasticsearch B.V. Elasticsearch Architecture — https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html.
18. Carnegie Mellon Database Group. Elasticsearch ACID support — https://dbdb.io/db/elasticsearch.
19. Statistics How To. Pearson’s correlation coefficient — https://www.statisticshowto.com/probability-and-statistics/correlation-coefficient-formula/#Pearson.
20. Spearman, C. The Proof and Measurement of Association between Two Things / C. Spearman // The American Journal of Psychology. — 1904. — Vol. 15, No. 1. — P. 72-101.
21. Kendall, M. A New Measure of Rank Correlation / M. Kendall // Biometrika. — 1938. — Vol. 30, No. 1-2. — P. 81-93.
22. LibreTexts. Introduction to Nonlinear Regression — https://math.libretexts.org/Workbench/Numerical_Methods_with_Applications_(Kaw)/6%3A_Regression/6.04%3A_Nonlinear_Regression.
23. Mihaela T. Udristioiu, Y. E. M. H. Y. Prediction, modelling, and forecasting of PM and AQI using hybrid machine learning / Mihaela T. Udristioiu, Y. E. M. H. Y. // Journal of Cleaner Production. — 2023. — Vol. 421, No. 138496.
24. StatCounter. Частка ринку настільних комп'ютерів, мобільних телефонів та планшетів у світі — https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-200901-202404.
25. Український гідрометеорологічний центр Державної служби України з надзвичайних ситуацій. Ситуація на пунктах спостереження радіометричної мережі НГМС — https://www.meteo.gov.ua/ua/Situaciya-na-punktakh-spostere-zhennya.
26. JetBrains. Kotlin — https://kotlinlang.org/.
27. JetBrains. Ktor — https://ktor.io/.

28. Google. Gson — https://github.com/google/gson.
29. Міністерство захисту довкілля та природних ресурсів України. ЕкоЗагроза — https://ecozagroza.gov.ua/about.
30. ГО “SaveDnipro”. SaveDnipro. Про нас — https://www.savednipro.org/about/.
31. Sensor.Community. Sensor.Community — https://sensor.community/ua/.
32. NASA. NASA FIRMS — https://firms.modaps.eosdis.nasa.gov/.
33. NASA. NASA FIRMS Map — https://firms.modaps.eosdis.nasa.gov/map/#d:24hrs;@33.2,48.2,6.3z.
34. NASA. NASA FIRMS API — https://firms.modaps.eosdis.nasa.gov/api/.
35. Oracle. Oracle Cloud Free Tier — https://www.oracle.com/ua/cloud/free/.
36. Terracotta, Inc. Ehcache — https://www.ehcache.org/.
37. JetBrains. Exposed — https://github.com/JetBrains/Exposed.
38. JetBrains. Exposed ORM. Connection pooling — https://ktor.io/docs/db-connection-pooling-caching.html#connection-pooling.
39. Wooldridge, B. HikariCP / B. Wooldridge — https://github.com/brettwooldridge/HikariCP.
40. Facebook Open Source. Prophet — https://facebook.github.io/prophet/.
41. The Linux Foundation. OpenAPI — https://www.openapis.org/.
42. SmartBear Software. Swagger UI — https://swagger.io/tools/swagger-ui/.
43. Google. Google Maps — https://www.google.com/maps.
44. GitHub, Inc. GitHub Actions documentation — https://docs.github.com/en/actions.
45. Docker Inc. Docker Documentation — https://docs.docker.com/.
46. Docker Inc. Docker Compose Documentation — https://docs.docker.com/compose/.

ДОДАТОК 1

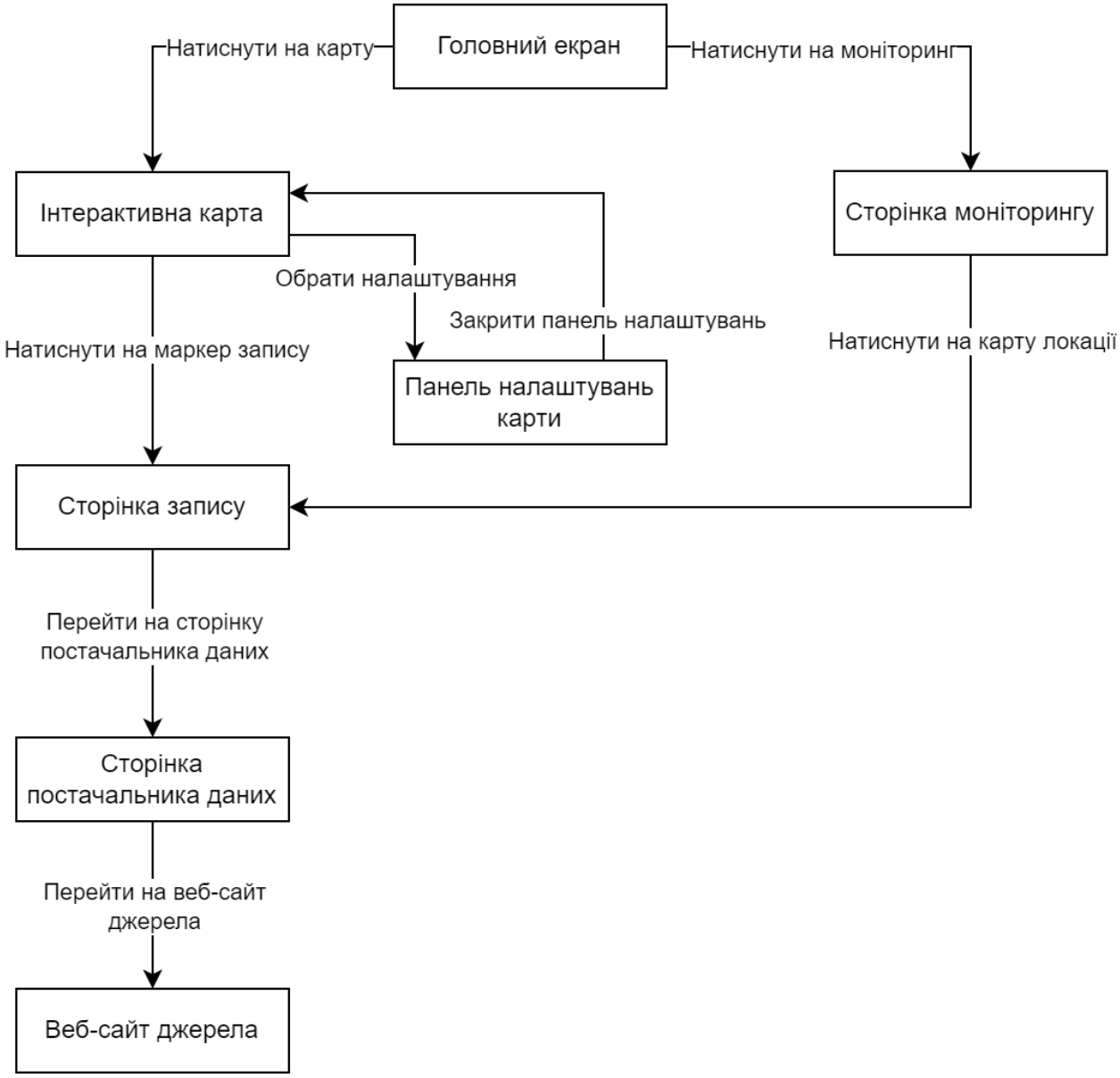
Агрегатор публічних екологічних даних України

Схема взаємодії користувача з Android-застосунком системи (структурна схема)

ІАЛЦ.467200.004 Е1

Аркушів 1

Київ - 2024 р.



					ІАЛЦ.467200.004 Е1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		*Воловик О. А.*			Агрегатор публічних екологічних даних України Схема взаємодії користувача з Android-застосунком системи (структурна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		*Ковальчук О. М.*					
Реценз.						*КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02*	
Н. Контр.		*Виноградов Ю. М.*					
Затвердив							

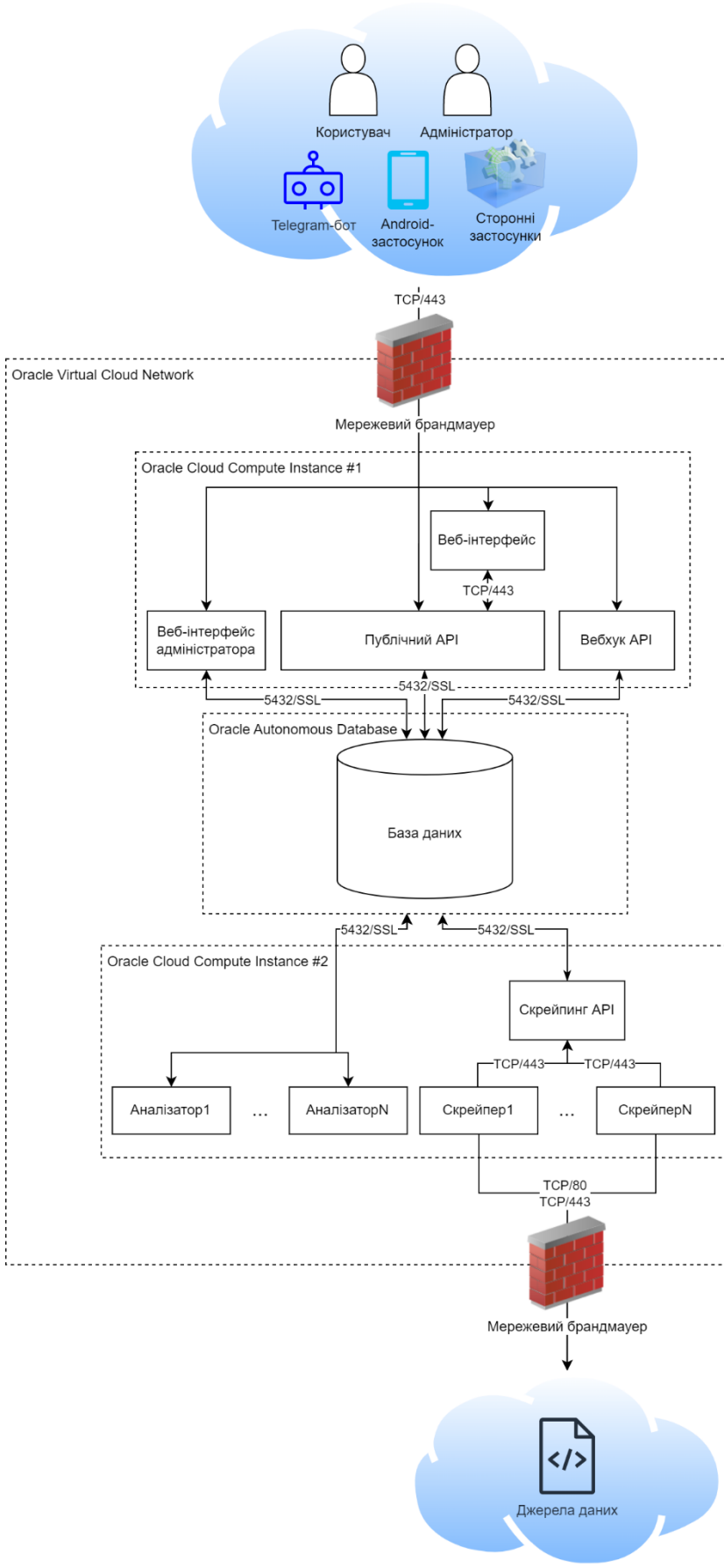
ДОДАТОК 2

Агрегатор публічних екологічних даних України

Схема інфраструктури системи (функціональна схема)
ІАЛЦ.467200.005 Д2

Аркушів 1

Київ - 2024 р.



					ІАЛЦ.467200.005 Д2	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		*Воловик О. А.*			Агрегатор публічних екологічних даних України Схема інфраструктури системи (функціональна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		*Ковальчук О. М.*				
Реценз.						*КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02*
Н. Контр.		*Виноградов Ю. М.*				
Затвердив						

ДОДАТОК З

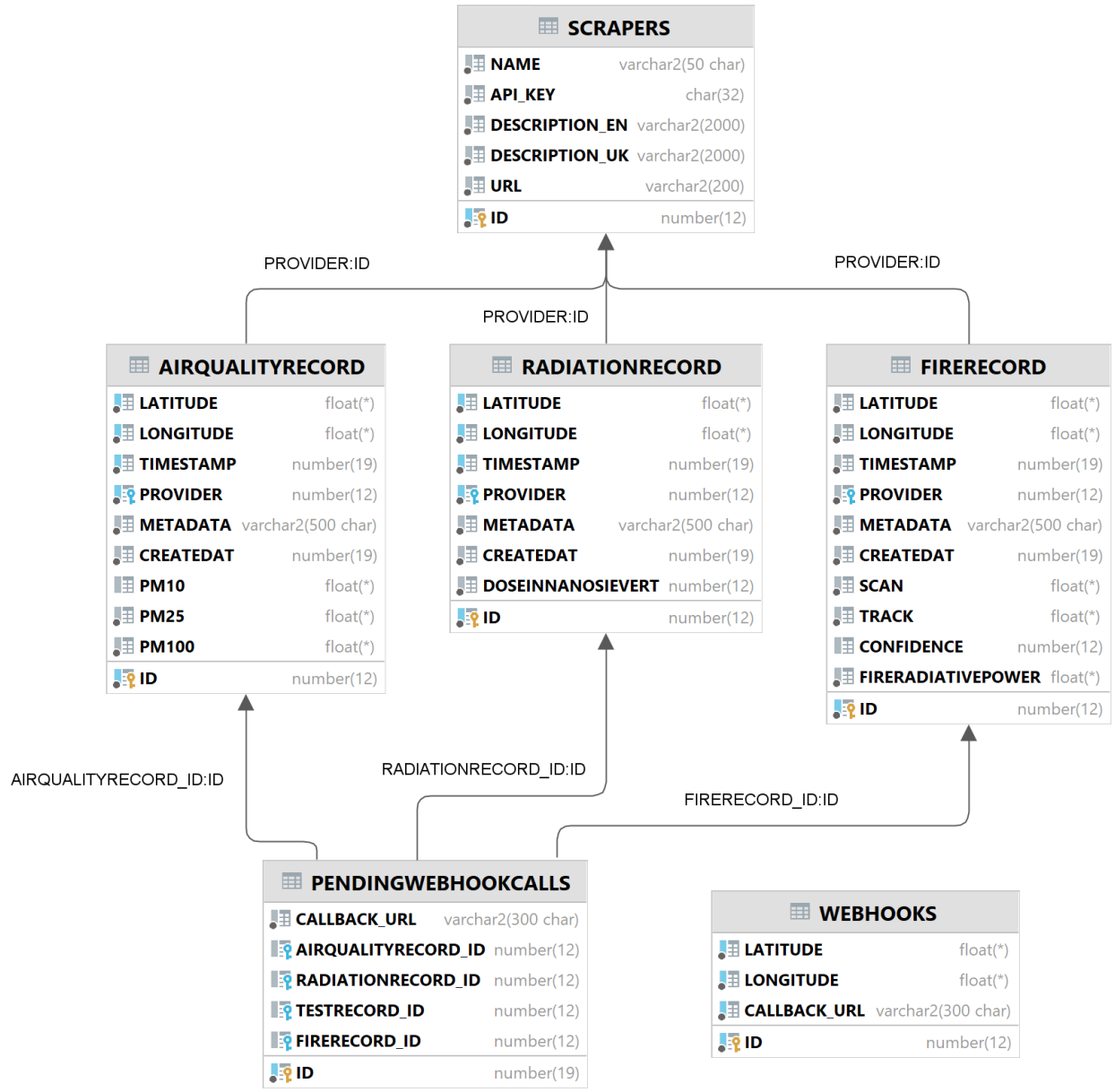
Агрегатор публічних екологічних даних України

Діаграма бази даних системи (принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ - 2024 р.



Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.006 Д3		
Розробив		*Воловик О. А.*			Агрегатор публічних екологічних даних України Діаграма бази даних системи (принципова схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		*Ковальчук О. М.*					
Реценз.						*КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02*	
Н. Контр.		*Виноградов Ю. М.*					
Затвердив							

ДОДАТОК 4

Агрегатор публічних екологічних даних України

Текст програмного коду
ІАЛЦ.467200.007 Д4

Аркушів 12

Київ - 2024 р.

RecordService.kt

```
package ua.eco.aggregator.backend

import org.jetbrains.exposed.sql.Column
import ua.eco.aggregator.base.model.AggregatedRecord
import ua.eco.aggregator.base.model.AggregatedRecordDTO
import ua.eco.aggregator.base.model.PaginatedData
import kotlin.reflect.KClass
import kotlin.reflect.KType

object SortFieldName {
    const val TIMESTAMP = "timestamp"
}

enum class SortDirection {
    ASCENDING,
    DESCENDING
}

/**
 * Record service
 *
 * @param T Record type
 * @param TDTO Record DTO type
 */
interface RecordService<T : AggregatedRecord, TDTO : AggregatedRecordDTO> {

    /**
     * Entity data property
     *
     * @property name Name of the property
     * @property type KType of the property
     */
    class EntityDataProperty(
        val name: String,
        val type: KType
    )

    val entityClass: KClass<T>
    val entityDTOClass: KClass<TDTO>
    val entityDataProperties: List<EntityDataProperty>
    val entityClassSimpleName: String
    val recordsTableIdColumn: Column<Int>
    val recordsTableName: String

    /**
     * Create new record
     *
     * @param entityDTO Entity DTOs
     * @param scraperId The ID of the scraper that collected these records. If
not specified, the scraper
     * ID will be calculated for each record separately using the apiKey
property of the entity DTO
     * @return True, if entity was created
     */
    suspend fun create(entityDTO: TDTO, scraperId: Int? = null): Boolean

    /**
     * Create a list of records
```

```
             *
             * @param entityDTOs List of entity DTOs
             * @return Number of entities created
             */
            suspend fun createMany(entityDTOs: List<TDTO>): Int

            /**
             * Read a record by ID
             *
             * @param id ID of record
             * @return Record(nullable)
             */
            suspend fun read(id: Int): T?

            /**
             * Get number of total submitted records by provider
             *
             * @param providerId Provider ID
             * @return Number of total submitted records by provider
             */
            suspend fun getTotalSubmittedRecordsByProvider(providerId: Int): Long

            /**
             * Delete record by ID
             *
             * @param id Record ID
             * @return
             */
            suspend fun delete(id: Int): Int

            /**
             * Read paginated
             *
             * @param providerId If specified, filters records by provider ID
             * @param timestampStart If specified with timestampEnd, filters records
by timestamp(timestampStart <= timestamp <= timestampEnd)
             * @param timestampEnd If specified with timestampStart, filters records
by timestamp(timestampStart <= timestamp <= timestampEnd)
             * @param latitude If specified with longitude, filters records by
coordinates
             * @param longitude If specified with latitude, filters records by
coordinates
             * @param sortFieldName The name of the record field by which the sort
will be performed
             * @param sortDirection The sort direction
             * @param page Page
             * @return Paginated data
             */
            suspend fun readPaginated(
                providerId: Int?,
                timestampStart: Long?,
                timestampEnd: Long?,
                latitude: Double?,
                longitude: Double?,
                sortFieldName: String,
                sortDirection: SortDirection,
                page: Long
            ): PaginatedData<T>

            /**
             * Read latest submitted records with distinct locations
```

```
            *
            * @param at If specified, ignores records with "timestamp" > "at"
            * @param maxAge If specified without "at" parameter, ignores records
older than "maxAge" seconds relative to current time.
            * If specified with "at" parameter, ignores records older than "maxAge"
seconds relative to "at".
            */
           suspend fun readLatestSubmittedRecordsWithDistinctLocations(
               at: Long? = null,
               maxAge: Long? = null
           ): List<T>
}
```

RecordServiceCachedImpl.kt

```
package ua.eco.aggregator.backend

import org.ehcache.config.builders.CacheConfigurationBuilder
import org.ehcache.config.builders.CacheManagerBuilder
import org.ehcache.config.builders.ExpiryPolicyBuilder
import org.ehcache.config.builders.ResourcePoolsBuilder
import org.ehcache.config.units.EntryUnit
import org.ehcache.config.units.MemoryUnit
import org.ehcache.impl.config.persistence.CacheManagerPersistenceConfiguration
import org.jetbrains.exposed.sql.Column
import ua.eco.aggregator.base.model.AggregatedRecord
import ua.eco.aggregator.base.model.AggregatedRecordDTO
import ua.eco.aggregator.base.model.PaginatedData
import java.io.File
import java.io.Serializable
import java.time.Duration
import kotlin.reflect.KClass

class RecordServiceCachedImpl<T : AggregatedRecord, TDTO : AggregatedRecordDTO>(
    private val delegate: RecordService<T, TDTO>,
    storagePath: File
) : RecordService<T, TDTO> {

    private data class PaginatedReadParameters(
        val providerId: Int?,
        val timestampStart: Long?,
        val timestampEnd: Long?,
        val latitude: Double?,
        val longitude: Double?,
        val sortFieldName: String,
        val sortDirection: SortDirection,
        val page: Long
    ) : Serializable

    private data class LatestSubmittedRecordsWithDistinctLocationsReadParameters(
        val at: Long?, val maxAge: Long?
    ) : Serializable

    override val entityClass: KClass<T> = delegate.entityClass
    override val entityDTOClass: KClass<TDTO> = delegate.entityDTOClass
    override val entityDataProperties: List<RecordService.EntityDataProperty> =
delegate.entityDataProperties
    override val entityClassSimpleName: String = delegate.entityClassSimpleName
    override val recordsTableIdColumn: Column<Int> = delegate.recordsTableIdColumn
```

```
    override val recordsTableName: String = delegate.recordsTableName

    private val cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
        .with(CacheManagerPersistenceConfiguration(storagePath))
        .withCache(
            "${entityClassSimpleName}byId",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(
                Int::class.javaObjectType,
                entityClass.java,
                ResourcePoolsBuilder.newResourcePoolsBuilder()
                    .heap(1000, EntryUnit.ENTRIES)
                    .offheap(10, MemoryUnit.MB)
                    .disk(100, MemoryUnit.MB, true)
            )
        )
        .withCache(
            "paginated${entityClassSimpleName}",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(
                PaginatedReadParameters::class.java,
                PaginatedData::class.java,
                ResourcePoolsBuilder.newResourcePoolsBuilder()
                    .heap(1000, EntryUnit.ENTRIES)
                    .offheap(10, MemoryUnit.MB)
                    .disk(100, MemoryUnit.MB, false)
            ).withExpiry(
                ExpiryPolicyBuilder.timeToLiveExpiration(Duration.ofMinutes(5))
            )
        )
        .withCache(
            "latestSubmitted${entityClassSimpleName}WithDistinctLocations",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(

LatestSubmittedRecordsWithDistinctLocationsReadParameters::class.java,
                List::class.java,
                ResourcePoolsBuilder.newResourcePoolsBuilder().heap(1000,
EntryUnit.ENTRIES)
            ).withExpiry(
                ExpiryPolicyBuilder.timeToLiveExpiration(Duration.ofMinutes(5))
            )
        )
        .withCache(
            "totalSubmitted${entityClassSimpleName}byProviderId",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(
                Int::class.javaObjectType,
                Long::class.javaObjectType,
                ResourcePoolsBuilder.newResourcePoolsBuilder()
                    .heap(1000, EntryUnit.ENTRIES)
                    .offheap(10, MemoryUnit.MB)
                    .disk(100, MemoryUnit.MB, false)
            ).withExpiry(
                ExpiryPolicyBuilder.timeToLiveExpiration(Duration.ofHours(24))
            )
        )
        .build(true)

    private val recordsByIdCache =
        cacheManager.getCache("${entityClassSimpleName}byId",
Int::class.javaObjectType, entityClass.java)

    private val paginatedReadsCache = cacheManager.getCache(
        "paginated${entityClassSimpleName}",
```

```
        PaginatedReadParameters::class.java,
        PaginatedData::class.java
    )

    private val latestSubmittedRecordsWithDistinctLocationsCache =
cacheManager.getCache(
        "latestSubmitted${entityClassSimpleName}WithDistinctLocations",
        LatestSubmittedRecordsWithDistinctLocationsReadParameters::class.java,
        List::class.java
    )

    private val totalSubmittedRecordsByProviderCache = cacheManager.getCache(
        "totalSubmitted${entityClassSimpleName}byProviderId",
        Int::class.javaObjectType,
        Long::class.javaObjectType
    )

    override suspend fun read(id: Int): T? =
        recordsByIdCache[id]
            ?: delegate.read(id)
                .also { record -> recordsByIdCache.put(id, record) }

    override suspend fun getTotalSubmittedRecordsByProvider(providerId: Int): Long =
        totalSubmittedRecordsByProviderCache[providerId]
            ?: delegate.getTotalSubmittedRecordsByProvider(providerId).also {
                totalSubmittedRecordsByProviderCache.put(providerId, it)
            }

    override suspend fun delete(id: Int): Int {
        recordsByIdCache.remove(id)
        return delegate.delete(id)
    }

    override suspend fun readPaginated(
        providerId: Int?,
        timestampStart: Long?,
        timestampEnd: Long?,
        latitude: Double?,
        longitude: Double?,
        sortFieldName: String,
        sortDirection: SortDirection,
        page: Long
    ): PaginatedData<T> {
        val parameters = PaginatedReadParameters(
            providerId = providerId,
            timestampStart = timestampStart,
            timestampEnd = timestampEnd,
            latitude = latitude,
            longitude = longitude,
            sortFieldName = sortFieldName,
            sortDirection = sortDirection,
            page = page,
        )

        return paginatedReadsCache[parameters] as PaginatedData<T>?
            ?: delegate.readPaginated(
                providerId,
                timestampStart,
                timestampEnd,
                latitude,
                longitude,
```

```
                sortFieldName,
                sortDirection,
                page
            ).also {
                paginatedReadsCache.put(parameters, it)
            }
    }

    override suspend fun readLatestSubmittedRecordsWithDistinctLocations(at: Long?,
maxAge: Long?): List<T> {
        val parameters = LatestSubmittedRecordsWithDistinctLocationsReadParameters(
            at = at,
            maxAge = maxAge
        )

        return latestSubmittedRecordsWithDistinctLocationsCache[parameters] as
List<T>?
            ?: delegate.readLatestSubmittedRecordsWithDistinctLocations(
                at, maxAge
            ).also {
                latestSubmittedRecordsWithDistinctLocationsCache.put(parameters, it)
            }
    }

    override suspend fun createMany(entityDTOs: List<TDTO>): Int =
delegate.createMany(entityDTOs)

    override suspend fun create(entityDTO: TDTO, scraperId: Int?): Boolean =
delegate.create(entityDTO, scraperId)
}
```

RecordServiceImpk.kt

```
package ua.eco.aggregator.backend

import kotlinx.coroutines.Dispatchers
import org.jetbrains.exposed.exceptions.ExposedSQLException
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.SqlExpressionBuilder.eq
import org.jetbrains.exposed.sql.Table.Dual.nullable
import org.jetbrains.exposed.sql.statements.InsertStatement
import org.jetbrains.exposed.sql.transactions.TransactionManager
import org.jetbrains.exposed.sql.transactions.experimental.newSuspendedTransaction
import org.jetbrains.exposed.sql.transactions.transaction
import org.koin.java.KoinJavaComponent.inject
import ua.eco.aggregator.base.model.AggregatedRecord
import ua.eco.aggregator.base.model.AggregatedRecordDTO
import ua.eco.aggregator.base.model.PaginatedData
import java.time.Instant
import kotlin.reflect.KClass
import kotlin.reflect.KProperty1
import kotlin.reflect.full.memberProperties
import kotlin.reflect.full.primaryConstructor
import kotlin.reflect.typeOf

private fun SortDirection.toSortOrder(): SortOrder =
    if (this == SortDirection.ASCENDING) SortOrder.ASC else SortOrder.DESC

class RecordServiceImpl<T : AggregatedRecord, TDTO : AggregatedRecordDTO>(
    override val entityClass: KClass<T>,
```

```
    override val entityDTOClass: KClass<TDTO>,
    database: Database,
    private val pageSize: Int
) : RecordService<T, TDTO> {

    override val entityDataProperties =
        entityClass.primaryConstructor!!.parameters.mapIndexedNotNull { index,
parameter ->
            if (index >= AggregatedRecord::class.memberProperties.size) {
                RecordService.EntityDataProperty(parameter.name!!, parameter.type)
            } else {
                null
            }
        }

    private val entityIntDataColumns = mutableListOf<Column<Int>>()
    private val entityNullableIntDataColumns = mutableListOf<Column<Int?>>()
    private val entityLongDataColumns = mutableListOf<Column<Long>>()
    private val entityNullableLongDataColumns = mutableListOf<Column<Long?>>()
    private val entityFloatDataColumns = mutableListOf<Column<Float>>()
    private val entityNullableFloatDataColumns = mutableListOf<Column<Float?>>()
    private val entityDoubleDataColumns = mutableListOf<Column<Double>>()
    private val entityNullableDoubleDataColumns = mutableListOf<Column<Double?>>()
    private val entityBooleanDataColumns = mutableListOf<Column<Boolean>>()
    private val entityNullableBooleanDataColumns = mutableListOf<Column<Boolean?>>()

    private val entityDataColumns = listOf(
        entityIntDataColumns,
        entityNullableIntDataColumns,
        entityLongDataColumns,
        entityNullableLongDataColumns,
        entityFloatDataColumns,
        entityNullableFloatDataColumns,
        entityDoubleDataColumns,
        entityNullableDoubleDataColumns,
        entityBooleanDataColumns,
        entityNullableBooleanDataColumns
    )

    private val scraperService by inject<ScraperService>(ScraperService::class.java)

    private val RecordsTable = object : Table(name = entityClass.simpleName!!) {
        val id = integer("id").autoIncrement()
        val latitude = double("latitude")
        val longitude = double("longitude")
        val timestamp = long("timestamp")
        val provider = reference("provider", scraperService.scrapersTableIdColumn)
        val metadata = varchar("metadata", length = 500)
        val createdAt = long("createdAt")

        override val primaryKey = PrimaryKey(id)

        init {
            uniqueIndex(latitude, longitude, timestamp, provider)
        }
    }

    override val entityClassSimpleName = entityClass.simpleName!!
    override val recordsTableName = RecordsTable.nameInDatabaseCase()
    override val recordsTableIdColumn = RecordsTable.id
```

```
    private fun Table.allFieldsAsString(tableName: String = this.tableName) =
        fields.joinToString(", ") { "$tableName.${(it as Column<*>).name}" }

    init {
        transaction(database) {
            for (property in entityDataProperties) {
                if (entityDataColumns.flatten().any { it.name == property.name }) {
                    continue
                }
                when (property.type) {
                    typeOf<Int>() ->
entityIntDataColumns.add(RecordsTable.integer(property.name))
                    typeOf<Int?>() ->
entityNullableIntDataColumns.add(RecordsTable.integer(property.name).nullable())
                    typeOf<Long>() ->
entityLongDataColumns.add(RecordsTable.long(property.name))
                    typeOf<Long?>() ->
entityNullableLongDataColumns.add(RecordsTable.long(property.name).nullable())
                    typeOf<Float>() ->
entityFloatDataColumns.add(RecordsTable.float(property.name))
                    typeOf<Float?>() ->
entityNullableFloatDataColumns.add(RecordsTable.float(property.name).nullable())
                    typeOf<Double>() ->
entityDoubleDataColumns.add(RecordsTable.double(property.name))
                    typeOf<Double?>() -> entityNullableDoubleDataColumns.add(
                        RecordsTable.double(property.name).nullable()
                    )

                    typeOf<Boolean>() ->
entityBooleanDataColumns.add(RecordsTable.bool(property.name))
                    typeOf<Boolean?>() -> entityNullableBooleanDataColumns.add(
                        RecordsTable.bool(property.name).nullable()
                    )

                    else -> throw IllegalArgumentException(
                        "${entityClass.simpleName} member property ${property.name}"
+
                                " type ${property.type} is not allowed."
                    )
                }
            }
            SchemaUtils.create(RecordsTable)
        }
    }

    private val fieldsIndex = RecordsTable.realFields.toSet().mapIndexed { index,
expression -> expression to index }
        .toMap()

    private suspend fun <T> dbQuery(block: suspend () -> T): T =
        newSuspendedTransaction(Dispatchers.IO) { block() }

    override suspend fun create(entityDTO: TDTO, scraperId: Int?): Boolean {
        val scraperId = scraperId ?:
scraperService.getByApiKey(entityDTO.apiKey)!!.id

        return try {
            RecordsTable.insert { table ->
                table[latitude] = entityDTO.latitude
                table[longitude] = entityDTO.longitude
                table[timestamp] = entityDTO.timestamp
```

```
                table[provider] = scraperId
                table[metadata] = entityDTO.metadata
                table[createdAt] = Instant.now().epochSecond

                table.setColumnValues(entityIntDataColumns, entityDTO)
                table.setColumnValues(entityNullableIntDataColumns, entityDTO)
                table.setColumnValues(entityLongDataColumns, entityDTO)
                table.setColumnValues(entityNullableLongDataColumns, entityDTO)
                table.setColumnValues(entityFloatDataColumns, entityDTO)
                table.setColumnValues(entityNullableFloatDataColumns, entityDTO)
                table.setColumnValues(entityDoubleDataColumns, entityDTO)
                table.setColumnValues(entityNullableDoubleDataColumns, entityDTO)
                table.setColumnValues(entityBooleanDataColumns, entityDTO)
                table.setColumnValues(entityNullableBooleanDataColumns, entityDTO)
            }.insertedCount != 0
        } catch (e: ExposedSQLException) {
            false
        }
    }

    override suspend fun createMany(entityDTOs: List<TDTO>): Int = dbQuery {
        var addedCounter = 0

        val allApiKeysEqual = entityDTOs.all { it.apiKey ==
entityDTOs.first().apiKey }

        val scraperId = if (allApiKeysEqual) {
            scraperService.getByApiKey(entityDTOs.first().apiKey)!!.id
        } else null

        entityDTOs.forEach { entityDTO ->
            if (create(entityDTO, scraperId)) {
                addedCounter++
            }
        }

        return@dbQuery addedCounter
    }

    private inline fun <reified T> InsertStatement<Number>.setColumnValues(columns:
List<Column<T>>, entityDTO: TDTO) {
        for (column in columns) {
            this[column] = getInstanceProperty<T>(entityDTO, column.name)
        }
    }

    override suspend fun read(id: Int): T? = dbQuery {
        RecordsTable.select { RecordsTable.id eq id }.map { it.toEntity() }
    }.singleOrNull()

    override suspend fun getTotalSubmittedRecordsByProvider(providerId: Int): Long =
dbQuery {
        RecordsTable.select { RecordsTable.provider eq providerId }.count()
    }

    override suspend fun delete(id: Int) = dbQuery {
        RecordsTable.deleteWhere { RecordsTable.id eq id }
    }

    override suspend fun readPaginated(
        providerId: Int?,
```

```
        timestampStart: Long?,
        timestampEnd: Long?,
        latitude: Double?,
        longitude: Double?,
        sortFieldName: String,
        sortDirection: SortDirection,
        page: Long
    ): PaginatedData<T> = dbQuery {
        var query = RecordsTable.selectAll()
        // 1. Apply filters
        // 1.1 Filter by providerId
        if (providerId != null) {
            query = query.andWhere { RecordsTable.provider eq providerId }
        }

        // 1.2 Filter by time period
        if (timestampStart != null && timestampEnd != null) {
            query =
                query.andWhere { (RecordsTable.timestamp greaterEq timestampStart)
and (RecordsTable.timestamp lessEq timestampEnd) }
        }

        // 1.3 Filter by location
        if (latitude != null && longitude != null) {
            query =
                query.andWhere { (RecordsTable.latitude eq latitude) and
(RecordsTable.longitude eq longitude) }
        }

        // 2. Apply sorting
        query = when (sortFieldName) {
            SortFieldName.TIMESTAMP -> query.orderBy(RecordsTable.timestamp to
sortDirection.toSortOrder())

            else -> {
                val sortFieldColumn = entityDataColumns.flatten().firstOrNull
{ it.name == sortFieldName }
                if (sortFieldColumn == null) {
                    throw IllegalArgumentException("Sort field $sortFieldName not
found in entity ${entityClass.simpleName}")
                }
                query.orderBy(sortFieldColumn to sortDirection.toSortOrder())
            }
        }

        val totalRecords = query.count()

        // 3. Apply paging
        val data = query
            .limit(pageSize, page * pageSize)
            .map { it.toEntity() }

        return@dbQuery PaginatedData(
            page = page,
            maxPageNumber = totalRecords / pageSize,
            itemsPerPage = pageSize,
            totalItemsCount = totalRecords,
            data = data
        )
    }
```

```
    override suspend fun readLatestSubmittedRecordsWithDistinctLocations(
        at: Long?,
        maxAge: Long?
    ): List<T> = dbQuery {
        val sqlWhereTimestamp =
            if (at != null && maxAge != null)
                "WHERE timestamp < $at AND timestamp > ${at - maxAge}"
            else if (at != null)
                "WHERE timestamp < $at"
            else if (maxAge != null)
                "WHERE timestamp > ${System.currentTimeMillis() / 1000 - maxAge}"
            else ""

        val sqlStatement =
            """
                SELECT ${RecordsTable.allFieldsAsString("t1")}
                FROM ${RecordsTable.nameInDatabaseCase()} t1
                JOIN (
                    SELECT latitude, longitude, MAX(timestamp) AS latest_timestamp
                    FROM ${RecordsTable.nameInDatabaseCase()}
                    $sqlWhereTimestamp
                    GROUP BY latitude, longitude
                ) t2 ON t1.latitude = t2.latitude AND t1.longitude = t2.longitude AND
t1.timestamp = t2.latest_timestamp
            """.trimIndent()

        nativeSelect(sqlStatement).map { it.toEntity() }
    }

    private fun ResultRow.toEntity(): T {
        val constructorArguments = buildList<Any?> {
            add(this@toEntity[RecordsTable.id])
            add(this@toEntity[RecordsTable.latitude])
            add(this@toEntity[RecordsTable.longitude])
            add(this@toEntity[RecordsTable.timestamp])
            add(this@toEntity[RecordsTable.provider])
            add(this@toEntity[RecordsTable.metadata])
            add(this@toEntity[RecordsTable.createdAt])

            for (entityDataProperty in entityDataProperties) {
                add(
                    when (entityDataProperty.type) {
                        typeOf<Int>() -> this@toEntity[entityIntDataColumns.first
{ it.name == entityDataProperty.name }]
                        typeOf<Int?>() ->
this@toEntity[entityNullableIntDataColumns.first { it.name ==
entityDataProperty.name }]
                        typeOf<Long>() -> this@toEntity[entityLongDataColumns.first
{ it.name == entityDataProperty.name }]
                        typeOf<Long?>() ->
this@toEntity[entityNullableLongDataColumns.first { it.name ==
entityDataProperty.name }]
                        typeOf<Float>() -> this@toEntity[entityFloatDataColumns.first
{ it.name == entityDataProperty.name }]
                        typeOf<Float?>() ->
this@toEntity[entityNullableFloatDataColumns.first { it.name ==
entityDataProperty.name }]
                        typeOf<Double>() ->
this@toEntity[entityDoubleDataColumns.first { it.name == entityDataProperty.name }]
                        typeOf<Double?>() ->
this@toEntity[entityNullableDoubleDataColumns.first { it.name ==
```

```
entityDataProperty.name }]
                        typeOf<Boolean>() ->
this@toEntity[entityBooleanDataColumns.first { it.name == entityDataProperty.name }]
                        typeOf<Boolean?>() ->
this@toEntity[entityNullableBooleanDataColumns.first { it.name ==
entityDataProperty.name }]
                        else -> throw IllegalArgumentException(
                            "${entityClass.simpleName} member property
${entityDataProperty.name}" +
                                    " type ${entityDataProperty.type} is not
allowed."
                        )
                    }
                )
            }
        }
        return
entityClass.primaryConstructor!!.call(*constructorArguments.toTypedArray())
    }

    private fun nativeSelect(query: String): List<ResultRow> {
        val resultRows = mutableListOf<ResultRow>()
        TransactionManager.current().exec(query) { resultSet ->
            while (resultSet.next()) {
                resultRows.add(ResultRow.create(resultSet, fieldsIndex))
            }
        }

        return resultRows
    }
}

@Suppress("UNCHECKED_CAST")
private fun <R> getInstanceProperty(instance: Any, propertyName: String): R {
    val property = instance::class.members
        // don't cast here to <Any, R>, it would succeed silently
        .first { it.name == propertyName } as KProperty1<Any, *>
    // force a invalid cast exception if incorrect type here
    return property.get(instance) as R
}
```