НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

“До захисту допущено”
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

“___”_____2025 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
“Цифрові технології в енергетиці”
зі спеціальності *122 “Комп’ютерні науки”*

на тему: **Застосунок пошуку Wi-Fi мереж**

Виконав:
студент IV курсу, групи ТР-13
ЛІСАК Семен Євгенович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник:
доцент, к.т.н., Ігор КУЗЬМЕНКО _____
(посада, вчене звання ,науковий ступінь, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент: *доцент кафедри ТАЕ, к.т.н., Олександр БАРАНЮК*
(посада, вчене звання, науковий ступінь, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль: Олександр ВОЛКОВ _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025

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

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

«__»_____2025 р.

ЗАВДАННЯ
на дипломну роботу студенту

ЛІСАК Семен Євгенович
(прізвище, ім’я, по батькові)

1. Тема роботи **Застосунок пошуку Wi-Fi мереж**

керівник роботи ***Кузьменко Ігор Миколайович, к.т.н., доцент***
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___»_____ 202__ р. №_____

2. Термін подання роботи студентом **07.06.2024 року**

3. Вихідні дані до роботи мова програмування Java

4. Перелік питань, які потрібно розробити _____
1) провести аналіз наявних аналогів застосунків для пошуку Wi-Fi мереж, виявити їх переваги та недоліки_____
2) аргументувати вибір інструментів, технологій та алгоритмів для створення програмного забезпечення_____

3) побудувати архітектуру програмного забезпечення, схеми баз даних та сховища файлів

4) створити мобільний інтерфейс застосунку для пошуку і візуалізації Wi-Fi мереж

5) представити процес роботи користувача з мобільним застосунком

5. Орієнтований перелік ілюстративного матеріалу діаграми, що демонструють класи програмного забезпечення; приклади роботи з мобільним застосунком; таблиці з порівнянням ключових характеристик підходів розробки та існуючих програмних рішень для пошуку WiFi мереж.

6. Дата видачі завдання «19» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми роботи	19.09.24	Виконано
2.	Вивчення та аналіз задачі	10.10.24-25.11.24	Виконано
3.	Розробка архітектури та загальної структури системи	01.12.24-10.01.25	Виконано
4.	Програмна реалізація системи	01.04.25-25.04.25	Виконано
5.	Оформлення пояснювальної записки	19.05.25-30.05.25	Виконано
6.	Захист програмного продукту	16.05.25	Виконано
7.	Передзахист	28.05.25	Виконано
8.	Подання готової роботи на кафедру	30.05.2025	Виконано
9.	Захист	16.06.25-20.06.25	Виконано

Студент _____ **Семен ЛІСАК**
(підпис) (ім’я, ПРІЗВИЩЕ)

Керівник роботи _____ **Ігор КУЗЬМЕНКО**
(підпис) (ім’я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота викладена на 78 сторінках, містить 8 ілюстрацій, 2 таблиці, 1 додаток та список із 12 джерел.

Мета дослідження – спроєктувати й реалізувати клієнт-серверну систему, що забезпечує централізоване зберігання понад 12 млн записів про Wi-Fi мережі, миттєвий пошук за BSSID, ESSID і геолокацією, а також інтуїтивну візуалізацію результатів у мобільному застосунку.

Методи та засоби. Для точкового пошуку у великих вибірках застосовано B-Tree-індекси за MAC-адресою та назвою мережі. Геопросторові запити оптимізовано 64-бітним Morton-кодуванням у поєднанні з діапазонними BETWEEN-операціями; гарантоване обмеження обсягу відповіді забезпечує адаптивний вибір рівня тайлів. Реалізовано двоетапну кластеризацію “Cluster + Point”, а також парсинг масових CSV-файлів з автоматичним приведенням типів.

Основний зміст. Проведено аналіз існуючих сервісів пошуку Wi-Fi мереж і показано переваги відкритої клієнт-серверної архітектури. Запропоновано механізм QuadKey-індексації, що забезпечує швидкий час відповіді на наборі з 10+ млн точок. Реалізовано REST-API для пошуку та імпорту, утиліти перетворення BSSID, діаграми класів і сервісів, а також мобільний клієнт, який сканує локальні мережі, отримує перевірені паролі та відображає їх на карті з динамічною агрегацією.

Результати. Створено прототип, придатний для мандрівників, студентів і громадських закладів: він скорочує час підключення до перевірених мереж, надає офлайн-режим і готовий до горизонтального масштабування у хмарі.

Ключові слова: Wi-Fi пошук, QuadKey, B-Tree, кластеризація, BSSID, CSV-парсинг, Android, Spring Boot, MySQL, Google Maps;

ABSTRACT

The thesis consists of 78 pages, 8 illustrations, 2 tables, 1 appendix, and a list of 12 references.

The purpose of the research is to design and implement a client-server system that provides centralized storage of more than 12 million records of Wi-Fi networks, instant search by BSSID, ESSID, and geolocation, as well as intuitive visualization of results in a mobile application.

Methods and tools. B-Tree indexes by MAC address and network name are used for point search in large samples. Geospatial queries are optimized by 64-bit Morton encoding combined with range BETWEEN operations; guaranteed response size limitation provides adaptive selection of the tile level. Two-stage clustering "Cluster + Point" is implemented, as well as parsing of bulk CSV files with automatic type conversion.

Main content. The analysis of existing Wi-Fi network search services is carried out and the advantages of an open client-server architecture are shown. The QuadKey indexing mechanism is proposed, which provides fast response time on a set of 10+ million points. The REST-API for search and import, BSSID conversion utilities, class and service diagrams, as well as a mobile client that scans local networks, receives verified passwords and displays them on a map with dynamic aggregation are implemented.

Results. We have created a prototype suitable for travelers, students, and public institutions: it reduces the time to connect to trusted networks, provides an offline mode, and is ready for horizontal scaling in the cloud.

Keywords: Wi-Fi search, QuadKey, B-Tree, clustering, BSSID, CSV parsing, Android, Spring Boot, MySQL, Google Maps;

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

СУБД – система управління базами даних, програмний засіб для роботи з БД.

API – Application Programming Interface, програмний інтерфейс прикладних програм.

APK – Android Package, формат пакета для встановлення застосунків на ОС Android.

BE – Back End, серверна частина застосунку.

B-Tree – Balanced Tree, збалансоване дерево (структура індексації в СКБД).

BSSID – Basic Service Set Identifier, MAC-адреса точки доступу Wi-Fi мережі.

CPU – Central Processing Unit, центральний процесор.

CSV – Comma-Separated Values, текстовий табличний формат даних.

DB – DataBase, база даних.

DTO – Data Transfer Object, транспортний об’єкт даних.

ESSID – Extended Service Set Identifier, ім’я (SSID) Wi-Fi мережі.

FE – Front End, клієнтська (видима) частина застосунку.

GKE – Google Kubernetes Engine, кероване середовище Kubernetes у хмарі Google.

GPS – Global Positioning System, глобальна навігаційна супутникова система.

HPA – Horizontal Pod Autoscaler, горизонтальний автоскейлер Kubernetes.

HTTP/HTTPS – HyperText Transfer Protocol / HyperText Transfer Protocol Secure, протокол (та захищений протокол) передачі гіпертексту.

JSON – JavaScript Object Notation, текстовий формат обміну даними.

MAC – Media Access Control, унікальна апаратна адреса мережевого інтерфейсу

MySQL – My Structured Query Language, система керування реляційними базами даних.

QuadKey – квадратний Morton-ключ для просторової індексації картографічних даних.

RDF – Resource Description Framework, модель представлення інформації про ресурси в Інтернеті, що використовується для семантичних запитів та опису правил

Redis – Remote Dictionary Server, високошвидкісне сховище ключ-значення в пам’яті.

RegEx – Regular Expression, регулярний вираз.

REST – Representational State Transfer, архітектурний стиль веб-API.

RSSI – Received Signal Strength Indicator, індикатор рівня потужності прийнятого сигналу Wi-Fi мережі.

R-tree – Rectangle tree, дерево прямокутних областей для просторової індексації.

S2-cell – осередок сферичної геоіндексації, що використовується в бібліотеці Google S2 Geometry Library.

SLA – Service Level Agreement, угода про рівень сервісу.

SQL – Structured Query Language, мова структурованих запитів до БД.

TCP – Transmission Control Protocol, транспортний протокол Інтернет.

TTL – Time To Live, час життя записи/пакета або кешу.

UI – User Interface, інтерфейс користувача.

Wi-Fi – Wireless Fidelity, технологія бездротового радіозв’язку локальних мереж.

ВСТУП

У сучасному світі кількість бездротових мереж (Wi-Fi) стрімко зростає – лише в 2024 році налічувалося понад 20 мільярдів точок доступу у світі. Користувачі мобільних пристроїв дедалі частіше шукають безкоштовний або перевірений доступ до Інтернету в громадських місцях, навчальних закладах, закладах харчування та під час подорожей. Відсутність централізованої системи зберігання й оперативного пошуку даних про Wi-Fi мережі значно ускладнює цей процес: користувачі витрачають час на ручний пошук і тестування з'єднань, а пошук за геолокацією часто повертає надто велику кількість результатів, які важко візуалізувати на мапі. З огляду на це існує потреба в ефективному програмному рішенні, що поєднує масштабоване зберігання великих обсягів даних із швидким пошуком як за Media Access Control-адресою (MAC-адресою), що використовується як ідентифікатор базового сервісного набору (BSSID), так і за географічною областю з використанням просторових індексів та алгоритмів кластеризації.

Метою дипломної роботи є розробка та реалізація клієнт-серверного програмного продукту, який забезпечує централізоване зберігання інформації про Wi-Fi мережі (>12 млн записів), швидкий пошук за BSSID і геолокацією, а також інтуїтивно зрозумілу візуалізацію результатів на мобільній карті.

Завдання роботи складається з 3 пунктів:

1. Аналіз підходів, методів та алгоритмів для завдання швидкого пошуку геопросторових даних та точкового пошуку за MAC-адресою.
2. Огляд та вибір програмних засобів для реалізації системи: у якості системи управління базами даних (СУБД) обрано My Structured Query Language (MySQL), яка підтримує індексацію за допомогою збалансованих дерев (B-Tree) та просторових індексів; серверна частина реалізована з використанням фреймворку Spring Boot версії 3 у поєднанні з підсистемою доступу до даних Spring Data JPA; контейнеризацію забезпечує Docker; мобільний клієнт створено для операційної системи Android з використанням мови

програмування Kotlin та фреймворку для побудови інтерфейсу Jetpack Compose.

3. Розробка серверної та клієнтської частин складається з 3 пунктів:
    1) інтерфейс передачі стану представлення – програмний інтерфейс прикладного програмування (REST-API) для пошуку по BSSID та картографічного запиту;
    2) алгоритми Morton-кодування (QuadKey) та двоетапної кластеризації (Cluster/Point);
    3) механізми імпорту даних JavaScript Object Notation (JSON), Comma-Separated Values (CSV).

Дипломна робота складається з вступу, чотирьох розділів, висновків, переліку використаних джерел та одного додатку. Загальний обсяг текстової частини — 78 сторінок, містить 8 ілюстрацій, 2 таблиці, 1 додаток та 12 джерел літератури.

Виклад матеріалу та виконані дослідження висвітлюють актуальність проблеми, забезпечують комплексний підхід до її вирішення та підтверджують практичну цінність створеного програмного забезпечення.

1 АНАЛІЗ ПРОБЛЕМИ АВТОМАТИЗОВАНОГО ПОШУКУ Wi-Fi МЕРЕЖ

З розвитком інформаційних технологій та зростанням попиту на мобільність виникає потреба у зручному та швидкому способі знаходження точок доступу до Інтернету. Особливо актуальним це є для мандрівників, студентів, фрилансерів та інших користувачів, які часто змінюють своє розташування та не мають постійного доступу до мережі [1]. Використання Wi-Fi мереж є одним із основних засобів забезпечення доступу до Інтернету в таких умовах.

Постає задача розробки системи, яка забезпечить пошук доступних Wi-Fi мереж за допомогою веб-технологій та хмарних сервісів. Така система має включати клієнтський застосунок, що сканує локальні мережі, та серверну частину, яка обробляє запити, здійснює пошук у базі даних та надає користувачеві інформацію про знайдені мережі. Важливим компонентом є використання хмарних технологій, які забезпечують масштабованість, доступність і централізоване зберігання даних.

1.1 Задача створення веб-системи для пошуку WiFi мереж

Актуальність задачі створення системи для пошуку WiFi мереж зумовлена зростаючою потребою користувачів у швидкому і надійному способі дізнатися, де поблизу доступні точки підключення до Інтернету. Йдеться не лише про визначення наявності сигналу, а й про отримання додаткової інформації: назви мережі, рівня сигналу, наявності пароля, а також її географічного розташування. У багатьох випадках ця інформація або недоступна, або потребує ручного збору, що не зручно для користувача і не відповідає очікуванням сучасного мобільного досвіду.

Передбачається, що цільовою аудиторією програмного забезпечення стануть мандрівники, студенти, віддалені працівники, а також будь-хто, хто часто перебуває в нових локаціях і потребує стабільного інтернет-з'єднання. Для цих користувачів важливо мати змогу швидко дізнатися про наявні мережі поблизу, не витрачаючи час на запити до персоналу закладів або на експериментальне підключення до кожної з доступних мереж вручну.

У зв’язку з цим виникає потреба в програмному рішенні, яке об'єднує локальне сканування WiFi мереж пристроєм з централізованою базою даних, що зберігає відому інформацію про мережі, отриману раніше іншими користувачами. Така система має надавати користувачу зручний інтерфейс для перегляду знайдених мереж, показувати рівень сигналу, наявність пароля та розташування точки на карті. У підсумку, програмне забезпечення повинно спрощувати процес підключення до WiFi мережі та мінімізувати пов’язані з цим часові й технічні витрати.

1.2 Етапи та підходи до реалізації системи

Розробка веб-системи для пошуку Wi-Fi мереж є багатокроковим процесом, який передбачає як проєктування архітектури програмного забезпечення, так і вибір відповідних технологій та методів реалізації. Щоб забезпечити цілісність і керованість процесу, розробку доцільно поділити на низку логічних етапів.

На першому етапі виконується дослідження предметної області та аналіз вимог користувачів. Це включає вивчення сценаріїв використання, розуміння потреб цільової аудиторії, визначення ключових функціональних і нефункціональних вимог. Далі формується загальне бачення системи, яке оформлюється у вигляді архітектурної концепції.

Другим етапом є проєктування — побудова структури бази даних, моделювання інформаційних потоків, розробка інтерфейсів прикладного

програмування (API) та вибір форматів передачі даних. На цьому етапі також приймається рішення щодо вибору технологічного стеку.

Третій етап охоплює безпосередню реалізацію: програмування серверної логіки, створення клієнтського інтерфейсу, налаштування взаємодії між компонентами.

Заключним етапом є розгортання системи в хмарному середовищі та підготовка документації.

Щодо підходів до реалізації, теоретично було розглянуто декілька можливих варіантів побудови системи.

Перший підхід — реалізація повністю локального рішення, де мобільний застосунок самостійно виконує як сканування Wi-Fi мереж, так і зберігання всієї інформації [4]. Такий варіант простий у реалізації, не вимагає серверної частини, проте має суттєві обмеження: неможливість динамічного оновлення даних, обмежена масштабованість, відсутність централізованого сховища й обмежений функціонал при зміні пристрою.

Другий підхід — використання пірингової моделі, де дані передаються між клієнтами безпосередньо або через тимчасові вузли [5]. Цей варіант дозволяє уникнути централізованого серверу, але створює значні труднощі в реалізації, особливо щодо безпеки, надійності та доступності даних. Крім того, пірингові системи погано масштабуються в умовах реального часу й не гарантують консистентності інформації.

Третій підхід — побудова клієнт-серверної архітектури з централізованою базою даних, яка розгортається в хмарному середовищі [6]. Клієнт збирає необхідну інформацію (BSSID, координати), надсилає її на сервер, де відбувається обробка та пошук відповідностей, після чого дані повертаються на пристрій для відображення. Цей варіант забезпечує централізоване управління, можливість зберігати й оновлювати інформацію незалежно від клієнта, розширюваність, а також просту підтримку безпеки й масштабування. Для детального порівняння ключових характеристик різних підходів до розробки системи складено таблицю 1.1.

Таблиця 1.1 — Порівняння ключових характеристик підходів розробки

Критерій	Локальне рішення	Піринґова модель	Клієнт-серверна архітектура
Актуальність даних	Низька	Залежить від пірів	Висока
Масштабованість	Обмежена	Складна	Висока
Безпека	Вразлива	Низька	Контрольована
Складність реалізації	Низька	Висока	Середня
Централізоване оновлення бази	Неможливе	Відсутнє	Повна підтримка
Надійність	Від пристрою	Від мережі пірів	Висока
Підходить для великої аудиторії	Ні	Ні	Так

Порівняльний аналіз підходів показав, що саме клієнт-серверна модель з використанням хмарної бази даних є найбільш доцільною для поставленої задачі. Вона дозволяє досягти балансу між продуктивністю, безпекою, актуальністю даних і зручністю використання як для кінцевого користувача, так і для розробника. Саме цей підхід було обрано як базовий для подальшої реалізації системи.

1.3 Наявні програмні рішення для пошуку WiFi мереж

На сучасному ринку існує низка мобільних застосунків, орієнтованих на пошук точок доступу Wi-Fi мереж. Вони надають користувачеві можливість перегляду інформації про мережі поблизу, зокрема — назву мережі (SSID), координати точки доступу, а також пароль, якщо він був раніше збережений іншими користувачами. Найпопулярнішими серед них є WiFi Map, Instabridge та Wiman.

WiFi Map — одна з найвідоміших платформ для пошуку Wi-Fi мереж, яка надає карту з точками доступу, зібраними користувачами [7]. Вона підтримує можливість додавання паролів, завантаження бази для офлайн-доступу, але обмежує безкоштовні функції і вимагає платну підписку для доступу до повного функціоналу.

Instabridge працює за схожим принципом, орієнтуючись на “розумний підбір” мережі для користувача [8]. Вона дозволяє автоматичне підключення до збережених мереж, має привабливий інтерфейс, але використовує закриту архітектуру, не надає API і не дозволяє розширення функціоналу чи адаптацію до специфічних задач.

Wiman менш поширений, однак також базується на краудсорсинговому підході [9]. Застосунок пропонує рейтинги мереж, офлайн-карту, однак не гарантує актуальності інформації й не дозволяє перевірити рівень сигналу в реальному часі.

Загальною особливістю цих сервісів є відсутність відкритої архітектури, складність інтеграції в сторонні рішення та обмежена підтримка асинхронної взаємодії. Крім того, жоден із них не надає повноцінної підтримки картографічного інтерфейсу з кластеризацією точок доступу й можливістю аналізу за рівнем сигналу. Для детального порівняння функціональних можливостей, архітектурних особливостей та обмежень існуючих сервісів була створена таблиця 1.2.

Таблиця 1.2 — Порівняння програмних рішень для пошуку WiFi мереж

Критерій	WiFi Map	Instabridge	Wiman
Наявність паролів	Так	Так	Так
Офлайн-доступ	Так (тільки PRO)	Частково	Так
Актуальність даних	Помірна	Низька	Низька
Підтримка карти	Є	Є	Є
Кластеризація точок	Немає	Немає	Немає
Рівень сигналу мережі	Немає	Немає	Немає
API для сторонніх сервісів	Відсутній	Відсутній	Відсутній
Можливість розширення	Низька	Відсутня	Відсутня
Необхідність підписки	Так	Так	Ні

Аналіз показує, що існуючі рішення хоч і частково задовольняють базові потреби користувача, однак мають суттєві обмеження — як з технічного, так і з функціонального боку. Зокрема, обмежена інтеграція з іншими системами, відсутність актуалізації даних у реальному часі, неможливість відображення сигналу й відсутність кластеризації на карті — все це ускладнює адаптацію таких продуктів під специфічні потреби користувачів.

Враховуючи ці недоліки, доцільним було прийняти рішення про розробку власної системи з відкритою клієнт-серверною архітектурою, яка б об'єднувала гнучкість, масштабованість, інтерактивність та можливість розширення функціоналу відповідно до потреб кінцевого користувача.

2 МЕТОДИ ТА АЛГОРИТМИ ОПТИМІЗОВАНОГО ПОШУКУ Wi-Fi МЕРЕЖ

Безпровідне середовище доступу до Інтернету характеризується водночас вибуховим приростом обсягу даних і високою динамікою топології. За таких умов класичні засоби пошуку—звичайні B-tree-індекси для ідентифікаторів і традиційні GIS-підходи для координат—виявляються недостатніми: вони не гарантують сталого часу відповіді при мільйонних вибірках і не здатні одночасно задовольнити обмеження на обсяг результатів, потрібний для коректного відтворення карти.

2.1 Постановка задачі та обґрунтування вибору методів

Перед системою пошуку Wi-Fi мереж ставляться чотири взаємозалежні завдання.

Перше, необхідно забезпечити швидкий ідентифікаційний пошук за унікальними мережевими ознаками—MAC-адресою точки (BSSID) або її назвою мережевою назвою (ESSID) — при обсязі бази даних понад десять мільйонів рядків. Для цього обрано класичний B-tree-індекс, який зберігає порядкові властивості ключа і дозволяє виконувати точкові запити з майже сталою латентністю навіть у великих вибірках.

Друге завдання стосується геопросторового пошуку: користувач визначає прямокутну ділянку на карті, і система має повернути репрезентативний, але кількісно обмежений набір точок, придатний для відображення на екрані мобільного пристрою. Традиційні просторові індекси на основі дерев прямокутних областей (R-tree) або коміркової геоіндексації за допомогою сферичних осередків (S2-cell) дають задовільну швидкодію лише для невеликих вікон, тоді як їхня продуктивність падає на запитах середнього масштабу. У зв’язку з цим обрано Morton-кодування (QuadKey), яке перетворює координати у лінійний 64-бітний

ключ і дозволяє формувати діапазонні запити звичайними операторами порівняння чисел, зберігаючи передбачувану швидкодію на різних масштабах карти.

Третє завдання полягає у поданні отриманих результатів у стислому вигляді. З цією метою застосовано ієрархічну кластеризацію: щільні групи точок агрегуються в один маркер зі статистично усередненими координатами, а поодинокі записи добираються окремо в обмеженій кількості. Такий підхід унеможливлює перевантаження інтерфейсу та дозволяє користувачу швидко орієнтуватися на карті.

Четверте завдання стосується достовірності й актуальності інформації. Використано каскадну систему валідації: перевірку формату BSSID, обмеження довжини текстових полів, контроль діапазону координат.

2.2 Методи швидкої ідентифікації мережевих ознак

У контексті сучасних систем управління та моніторингу бездротових мереж критично важливим є забезпечення миттєвого доступу до інформації про конкретні точки доступу навіть за умови значних обсягів даних. Кожен запис у базі даних описується набором ознак, серед яких найважливішими є апаратно незмінна MAC-адреса пристрою (BSSID) та його мережеве ім’я (ESSID). Коли обсяг таблиці перевищує позначку в 10 млн рядків, пряме сканування всього масиву даних при кожному запиті призводить до неприйнятного зростання латентності та навантаження на дискову підсистему. У цьому розділі розглянуто підходи до організації ефективного індексування з використанням класичної структури B-tree, яка дозволяє гарантувати логарифмічну складність операцій пошуку та збереження майже сталої швидкодії точкових запитів незважаючи на збільшення обсягу даних.

2.2.1 Теоретичні засади індексування великих обсягів даних

Індексування в реляційних СУБД ґрунтується на побудові структурованих репрезентацій розподілу ключових полів у таблиці з метою усунення необхідності повного перегляду кожного запису. Однією з найбільш розповсюджених і водночас перевірених часом структур є збалансоване B-tree-дерево. У B-tree-індексі кожен вузол (як внутрішній, так і листовий) містить набір ключів, впорядкованих за зростанням, а вказівники на дочірні вузли організовані так, що глибинна відстань від кореня до будь-якого листа змінюється дуже повільно навіть при збільшенні кількості елементів у тисячі разів. Класична теорема про складність зазначає, що операції пошуку, вставки та видалення у B-tree виконуються за час $O(\log_k N)$, де N – загальна кількість ключів, а k – порядок дерева, пропорційний максимально допустимій кількості ключів у вузлі. Крім того, розмір вузлів B-tree зазвичай співпадає з розміром сторінки дискової пам’яті, що мінімізує кількість I/O-операцій при навігації по індексу. Таким чином, теоретичні засади B-tree-індексів забезпечують варіант організації даних, який оптимальний для дискових субсистем із блоковим читанням/записом і гарантує збалансовану витрату ресурсів навіть за мільйонів ключів.

Додатковим фактором стабільності продуктивності є властивість B-tree автоматично перерозподіляти ключі між вузлами під час вставки та видалення, що запобігає перекосу структури. При досягненні порогу заповнення вузла вище заданого рівня відбувається поділ (split), а при опусканні нижче — злиття (merge) або позичання (borrow) ключів із сусідніх вузлів. Це гарантує, що різниця в глибинах гілок ніколи не перевищує одиниці, підтримуючи високу швидкість навігації по індексу без необхідності періодичного повного перебудовування.

2.2.2 Використання B-tree-індексу для ідентифікації за BSSID та ESSID

Практична реалізація індексування в рамках досліджуваної системи передбачає створення двох незалежних B-tree-індексів: один — для поля BSSID,

другий — для поля ESSID. Після ініціалізації індексу на цих полях кожен новий запис автоматично вставляється у відповідне дерево. При надходженні запиту на отримання інформації за конкретною BSSID алгоритм починає роботу з кореневого вузла, послідовно порівнюючи шукане значення з ключами до тих пір, поки не досягне листового вузла. За кожного порівняння відкидається половина ізольованих гілок, а завдяки збалансованості дерева загальна кількість порівнянь залишається обмеженою значенням, що пропорційне логарифму від розміру колекції. Аналогічний механізм працює для ESSID, що дозволяє забезпечити однаковий рівень оптимізації запитів за текстовими назвами мереж. У такий спосіб можна виконувати запити на кшталт вибірки всіх параметрів точки доступу, що відповідає заданому BSSID або ESSID, з очікуваним часом відгуку на рівні одиниць мілісекунд незалежно від кількості записів у таблиці.

2.2.3 Переваги та недоліки застосування B-tree-індексу

Однією з ключових переваг використання збалансованого B-tree є гарантована логарифмічна складність усіх операцій із точки зору розміру даних, що забезпечує майже сталу швидкодію точкових запитів за BSSID та ESSID незалежно від зростання таблиці. Крім того, вузли індексу корелюють із розміром диск-ових сторінок, що мінімізує I/O-витрати під час навігації, а вбудований механізм розподілу та балансування ключів запобігає деградації структури. Використання окремих індексів для мережевих ознак дозволяє уникнути штрафів за складені індекси та спрощує адміністрування, тоді як кешування вищеописаних рівнів дерева знижує затримки під навантаженням.

Водночас B-tree-індекс має і певні обмеження. Для текстових полів (ESSID) зі значною змінністю довжини рядків може виникати неоднорідне заповнення вузлів, що інколи призводить до незбалансованого розміщення ключів та збільшення витрат на префіксну компресію. Сама процедура створення індексу та його первинне заповнення може займати значний час і ресурси при роботі з сотнями мільйонів записів, тож у разі одноразової побудови потрібно ретельно

планувати вікно технічного обслуговування та інструменти паралельного завантаження даних.

2.3 Алгоритм просторової індексації QuadKey

2.3.1 Походження даних і мотивація використання QuadKey

Сировинною основою для розробки модуля просторового пошуку стала відкрита база Wireless Geographic Logging Engine (WiGLE) [4], яка містить понад 9,7 млн записів BSSID та геокоординат точок доступу по всьому світу. Джерело цих даних — опубліковані дампи ентузіастів-мандрівників, які збирають інформацію про відкриті мережі за допомогою мобільних пристроїв та GPS-передавачів. Через величезний об’єм і нерівномірний розподіл точок (щільні міські райони проти майже порожніх тундрових просторів) постало завдання, яке складається з 3 пунктів, забезпечити одночасно:

1) швидкий пошук будь-яких точок у довільному прямокутнику на карті;
2) масштабовану кластеризацію результатів;
3) ефективну агрегацію для мобільних клієнтів із низьким трафіком.

Традиційні підходи (R-tree, S2--тайли) або добре індексують по одній координаті, але гірше -балансують запити типу “кластеризація плюс детальне вибіркове сканування”. Натомість Morton-сортування у вигляді QuadKey дає рівномірний бітовий простір для швидкого позиціювання і зручне агрегаційне GROUP BY на рівні SQL.

2.3.2 Концепція QuadKey

QuadKey [10] — це 64-бітовий ключ, який кодує положення точки на земній кулі через послідовне подвоєння простору за широтою та довготою. Ідея полягає в мортоновому (Z-порядковому) перетворенні: кожному рівню деталізації відповідають дві біти (NS та EW), які по черзі з’єднуються в єдиний ключ, складається з 3 пунктів.

1. Розбиття простору.

На рівні L вся поверхня EarthBox $[-90,90] \times [-180,180]$ ділиться на $2^L \times 2^L$ квадратів.

2. Бітове кодування. На кожній ітерації дві бітові позиції формують номер підквадранта:
    1) старший біт (2-й) = 1, якщо широта ≥центр, інакше 0;
    2) молодший біт (1-й) = 1, якщо довгота ≥ центр, інакше 0.
3. Наращування ключа. Кожен новий двобітний фрагмент додається в кінець попереднього ключа зліва (shift left).

Таким чином, ключ k для рівня глибиною D — це

$$k = \sum_{i=0}^{D-1} \left(q_i \ll 2(D-1-i)\right), \quad (1)$$

де $q_i \in \{0\ldots3\}$ — номер квадранта на ітерації i.

2.3.3 Динамічний вибір рівня деталізації та гарантія обмеження маркерів

Щоб не перевантажувати клієнтську карту маркерами, сервіс обчислює оптимальний "глибокий" рівень L^* залежно від розмірів області запиту. Уявіть, що ви малюєте велику карту: чим вона ширша, тим менше деталей можна відобразити без хаосу. Алгоритм спочатку оцінює кількість "тайлів" (або кластерів) вздовж широти та довготи, виходячи з бітової розбивки QuadKey.

Постійне збільшення розмірності QuadKey спрощує обчислення кількості тайлів: для поточного прямокутника ми можемо миттєво порахувати, скільки бітових блоків падає всередину області, і якщо їхня добуткова кількість менша за внутрішню межу, обрати саме цей рівень. Таким чином, на великому масштабі буде обрано малі глибини (більші тайли), а на вузькому – велика глибина (дрібніша деталізація).

2.3.4 Переваги та недоліки підходу QuadKey

Переваги QuadKey полягають у тому, що він об’єднує двовимірний пошук у рамках одного простого індексу. Вам не потрібно тримати окремо дерево R-tree для координат і ще один індекс для порядку Morton – достатньо єдиного поля, і будь-який діапазонний запит із бітовим зсувом працює миттєво. Оскільки ключ формують звичайні бітові операції, витрати на CPU мінімальні, навіть за мільйонів обчислень при завантаженні даних.

Разом з тим, підходу властиві свої компроміси. По-перше, нерівномірна в географічному сенсі сітка QuadKey може призводити до дещо різного фактичного розміру тайла: на екваторі один рівень деталізації дає квадрати ~ 111×111 км, а в помірних широтах – ~ 111×80 км. По-друге, під час відображення дуже дрібних об’єктів (наприклад, окремих будинків) рівень 23–24 може бути недостатнім для точної локалізації без додаткових коригувань.

Нарешті, прямі запити BETWEEN по QuadKey не охоплюють складні многокутники чи кола й вимагають додаткової фільтрації координат у результаті. Проте ця неточність нівелюється тим, що при детальному масштабі карти відображаються одиничні точки без застосування кластеризації, тож надлишкові групування не використовуються й запити залишаються простими. У типовому сценарії “прямокутник на карті” такий підхід демонструє стабільно високу продуктивність і передбачуваність часу відповіді.

2.4 Перевірка параметрів пошуку та оцінка розміру області

Перед тим як виконати геопросторовий запит, GeoService проводить послідовну валідацію вхідних координат і попередню оцінку обсягу результату. Це дозволяє відсіяти невірні або занадто великі області ще на рівні бізнес-логіки та зберегти ресурси сервера, забезпечивши при цьому стабільну швидкодію та передбачуваність часу відповіді. Далі розглядаються три ключові підходи:

перевірка кордонів запиту, розрахунок орієнтовної кількості точок і динамічний вибір рівня деталізації тайлів.

2.4.1 Просторова валідація меж

На етапі валідації координат здійснюють перевірку наявності запиту та належності його граничних значень до глобальних меж широти й довготи. Упорядкованість країв прямокутника гарантує, що мінімальні координати строго менші за максимальні, а їх вихід за фізично можливі діапазони (–90…+90° по широті і –180…+180° по довготі) фіксується на ранньому ґатунку. Це запобігає подальшій марній обробці некоректно заданих ділянок. Перевага підходу – миттєва відмова від хибних запитів із мінімальними обчисленнями; недолік – жорстка відсіч може відкидати граничні, але потенційно коректні запити без додаткової корекції.

2.4.2 Оцінка об’єму потенційних результатів

Після перевірки границь відбувається статистична оцінка обсягу даних, яка базується на геометрії прямокутника та середній густині точок залежно від широти. Площа візуальної області конвертується в квадратні кілометри із врахуванням звуження довгот до полюсів, після чого множиться на модельну щільність розподілу об’єктів. Перевага — швидка рання індикація надмірних запитів; недолік — використання усереднених параметрів може бути неточним у регіонах з аномальною концентрацією точок.

2.4.3 Динамічна адаптація рівня деталізації

У фінальному кроці підбирається найбільш дрібний рівень розбиття простору (тайлів), при якому загальна кількість клітин не перевищує технічного порогу відповіді. Для цього послідовно переглядають потенційні ступені

деталізації — від найгрубіших до найдрібніших — і обчислюють, скільки клітин припадає на вибрану область. Перший рівень, що задовольняє обмеження, обирається для побудови фінальної відповіді. Такий адаптивний механізм дозволяє зберегти баланс між деталізацією та продуктивністю: чим ширша область, тим більші клітини (грубіший масштаб), і навпаки. Недоліком може бути недостатня точність у випадку витягнутих або неправильно орієнтованих прямокутників, оскільки алгоритм не оптимізує форму тайлів під специфічну геометрію запиту.

2.4.4 Переваги та недоліки підходу

Застосування багатоступеневої валідації та попередньої оцінки значно знижує кількість зайвих запитів до основного індексу та запобігає надмірному навантаженню сервера, що особливо важливо за високої частоти одночасних запитів, а динамічний підбір рівня деталізації тайлів дозволяє підтримувати баланс між об’ємом переданих даних і точністю відображення результату. Водночас оцінка кількості точок здійснюється на основі усереднених моделей площі та щільності, тож для регіонів з аномально високою або низькою фактичною концентрацією точок можливі відхилення від реального числа об’єктів; крім того, додаткова логіка валідації і розрахунків формує невеликий, але відчутний CPU-й та кодову складність, яку слід враховувати при інтеграції в середовище з обмеженими ресурсами.

2.5 Перетворення BSSID у рядок

Щоб представити 48-бітний числовий ідентифікатор точки доступу (BSSID) у зручному для людини форматі XX:XX:XX:XX:XX:XX, у мапері GeoMapper застосовується простий побітовий алгоритм. Спочатку перевіряється, чи не дорівнює вхідне значення bssidNum null — у такому випадку метод повертає null, що дозволяє безпечно обробляти відсутні дані. Далі сам 64-бітний тип Long

сприймається як 48-бітна послідовність з шести байтів: найстарший байт міститься у старших 8 бітах числа, наймолодший — у молодших 8.

На кожному з шести кроків виконується зсув вправо на 40, 32, 24, 16, 8 та 0 біт відповідно, після чого бінарна операція AND з маскою 0xFF виділяє лише ті 8 біт, що становлять конкретний байт. Отримані значення в діапазоні від 0 до 255 форматуються в двозначний шістнадцятковий рядок за допомогою конструкції String.format("%02X", byteValue), де %02X забезпечує виведення провідного нуля і літер A–F у верхньому регістрі. Ці шість блоків збираються в єдиний рядок із розділенням двокрапками, починаючи від старшого байту до наймолодшого.

Оскільки алгоритм складається лише з кількох зсувів і побітових масок, а форматування здійснюється в межах одного виклику String.format, час виконання методу є сталим (O(1)), а обсяг використовуваної пам’яті мінімальний. Такий підхід гарантує надійне, уніфіковане та читабельне представлення BSSID для подальшої обробки або відображення.

2.6 Парсинг CSV-файлів

Метод parseCsv починає з перевірки вхідного файлу: якщо об’єкт відсутній або порожній, або якщо його MIME-тип та розширення не відповідають CSV, запит припиняється з повідомленням про помилку. Після успішної валідації створюється BufferedReader з кодуванням UTF-8, який передається у конфігурований CSVParser з вказаними заголовками, пропуском рядка з ними та обрізкою зайвих пробілів.

Далі парсер послідовно обробляє кожен рядок у файлі: для кожного запису формується новий елемент запиту, у який по черзі копіюються значення з колонок. Текстові поля (BSSID, ESSID, WiFiKey) передаються без змін, тоді як числові та булеві поля конвертуються спеціальними допоміжними методами. Метод перетворення числа на Double або Integer спершу видаляє зайві пробіли, намагається розпарсити рядок і, у разі невдалої спроби, кидає виняток із детальним повідомленням. Булеві значення визначаються на основі декількох варіантів

текстових входів ("true", "1", "yes"), а порожні рядки завжди дають null. Після обробки усіх полів об’єкт додається до списку, що накопичує записи.

Після завершення циклу перевіряється, чи знайдено хоча б один рядок даних; порожній результат трактуються як помилка формату файлу. Якщо список непустий, із нього формується кінцевий запит NetCreateRequest, який містить готовий набір об’єктів для подальшої обробки. Уся робота зчитування та конвертації відбувається в межах одного блоку try-with-resources, який автоматично закриває потоки, а будь-які помилки введення-виведення або формату переводяться в IllegalArgumentException. Така послідовність кроків гарантує коректний і стійкий до некоректних даних механізм перетворення CSV у внутрішню модель програми.

2.7 Методика кластеризації та відбору точок

Реалізація методу кластеризації поєднує у собі одночасне збереження інформації про щільність розміщення точок доступу та можливість деталізованого відображення окремих BSSID. Основна ідея полягає в тому, щоб на великому масштабі картографічного відображення групувати численні точки в умовні “кластерні маркери”, а при зменшенні масштабу або виділенні окремих зон – показувати детальні дані про поодинокі точки. Розгляньмо детально ключові етапи цієї методики.

2.7.1 Гібридна відповідь “кластер + синглтони”

При виконанні запиту findByArea() сервіс повертає масив об’єктів MapItem, які можуть бути двох різних типів:

1) ClusterDto – агрегований маркер з координатами центроїда та кількістю точок;
2) PointDto – одинична точка з повним описом (BSSID, ESSID, WiFiKey).

Перший крок полягає в агрегуванні всіх записів у таблиці geo, що потрапляють в межі запитуваного прямокутника. За допомогою SQL-запиту з GROUP BY (quadkey >> shift) кожна група точок певного квадранта стискається до одного кластера. Завдяки попередньо обчисленому quadkey MySQL за лічені мілісекунди збирає статистику: число точок у кластері, середні координати широти та довготи.

Проте далеко не всякий кластер підходить для узагальненого відображення: якщо в межах квадранта опинилася лише одна точка, вона втрачає смисл як кластер. У цьому випадку її переносять до окремого списку singleCls, який обробляється трохи пізніше. В результаті SQL-агрегації виходить два набори: масив вже готових ClusterDto та перелік ідентифікаторів кластерів-одиночок.

2.7.2 Добіркове сканування по одиничних кластерах

Після формування початкових кластерів сервіс виконує другий запит – на вилучення детальної інформації про ті точки, які належать до “синглтонів”. Для цього динамічно формується SQL-умова з BETWEEN :startN AND :endN для кожного quadkey-діапазону та приєднанням таблиці nets через JOIN по BSSID. Завдяки індексу idx_geo_quadkey_lat_lon MySQL спочатку швидко відсіює широкі діапазони за quadkey, а потім уточнює відбір за координатами, обмежуючи кількість повернутих точок для мінімізації навантаження на мережу та клієнтський інтерфейс.

У Java-реалізації відповідальність за цю операцію лежить на методі findSinglesByClusters() класу GeoRepositoryImpl. Він приймає на вхід список кластерних ідентифікаторів, генерує частину WHERE-умови з OR-переліком, встановлює параметри діапазонів і виконує getResultList(). Після отримання “сирих” рядків (BSSID, latitude, longitude, ESSID, WiFiKey) метод перетворює їх у об’єкти передачі даних (DTO).

2.7.3 Об’єднання результатів та гарантії ліміту

Після двох етапів — первинного групування та добіркового сканування — сервіс у єдиному списку MapItem об’єднує масив ClusterDto і масив PointDto. Завдяки підходу “кластер плюс синглтони” загальна кількість елементів у відповіді залишається в межах передбачуваного порогу, що гарантує стабільну швидкість передачі та рівномірне навантаження на фронтенд. Ключове в цьому підході — лімітування на основі двох правил опису ресурсів (RDF) в SQL: перший запит не згортає одиничні точки до кластерів, а лише визначає їх загальне число; другий запит застосовує оператор LIMIT для обмеження масштабу вибірки. Така стратегія не тільки забезпечує оптимальний масштаб обробки, але й дозволяє ефективно підключати кешування клієнтських результатів (наприклад, Redis) саме на агреговані набори.

2.7.4 Оцінка продуктивності та адаптивність

Методика кластеризації показує високу стабільність часу відповіді навіть при зростанні бази до десятків мільйонів записів. На практиці заміри для області в межах 1° × 1° (≈ 111 × 79 км у Європі) дають такі 2 типу значень:

1) ~ 20 мс на агрегацію за quadkey;
2) ~ 5–10 мс на добіркове сканування одиночних точок;

При більшій площі прямокутника оцінка числа кластерів відбувається за допомогою простих арифметичних обчислень (без залучення БД), тому реальний час залишається майже лінійно залежним тільки від кількості кластерів і одиничних точок, а не від загальної щільності даних.

Своєю чергою, якщо запит охоплює територію з високою щільністю (наприклад, центр Києва), кількість кластерів автоматично зменшується — обирається менший рівень деталізації, що гарантує плавну взаємодію навіть на мобільних пристроях з повільним каналом.

2.7.5 Переваги та недоліки методики кластеризації

Методика “кластер + синглтони” поєднує сильні сторони агрегованої та детальної відображуваності просторових даних, проте водночас накладає певні обмеження на складність запитів та їхні виконавчі плани. Нижче наведено основні переваги й недоліки цього підходу.

Метод забезпечує високу масштабованість: навіть за мільйонів записів перший SQL-запит із групуванням за зміненим quadkey виконується за O(log n) з урахуванням індексу, а точкові вибірки “синглтонів” відбуваються винятково в межах невеликого підмножини, що суттєво зменшує навантаження на БД і мережу. Поділ на агреговані кластери та одиничні точки дає можливість клієнту миттєво отримати оглядовий рівень даних і лише за потреби завантажити деталі, що суттєво економить трафік на мобільних пристроях і пришвидшує рендер карти. Використання двох етапів запитів та чітких порогів гарантує контроль над загальною кількістю маркерів у відповіді й дозволяє інтегрувати кешування на рівні агрегованих результатів (наприклад, у Redis), зменшуючи частоту звернень до первинної сховища.

Водночас подвійний прохід по даних підвищує загальну складність реалізації та може призвести до помітних накладних витрат на складання динамічних SQL-умов, особливо якщо кількість “синглтонів” у вузьких зонах стає великою. Групування за quadkey базується на фіксованій решітці, що може не відповідати реальній геометрії кластерів — за нерівномірного розподілу точок у межах квадранта алгоритм втрачає гнучкість у формуванні форм кластерів. Крім того, у найбільш насичених центрах із великою кількістю “синглтонів” другий запит може повернути значний обсяг даних до наступного рівня фільтрації, що призводить до пікових навантажень на мережу та обробку на стороні клієнта.

2.8 Балансування навантаження та масштабування

У програмній архітектурі, попри відсутність явного розподілу на численні мікросервіси, закладено ключові принципи, що забезпечують масштабованість і надійність роботи системи під високим навантаженням. Насамперед це принцип “stateless”, коли жоден екземпляр програми не зберігає стан користувача чи сесію в локальній пам’яті, а всі взаємодії відбуваються через зовнішні сервіси — базу даних або механізми кешування. По-друге, атомарність і консистентність змін підтримуються за допомогою транзакцій Spring, що захищає систему від неповних або некоректних оновлень. За таких умов будь-яка кількість реплік одного й того ж Docker-образу може без змін у коді обробляти як читальні, так і записні запити, а балансувальник рівномірно розподіляє трафік між усіма запущеними інстансами, гарантують оптимальне використання ресурсів.

2.8.1 Підхід “stateless”

Весь обмін інформацією із клієнтами реалізовано через REST-ендпоінти, які не залежать від локальних сесійних даних [11]. Кожний HTTP-запит містить увесь набір необхідних параметрів (токен аутентифікації, ідентифікатори об’єктів, дані форми), тож жоден екземпляр сервісу не має необхідності довготривало зберігати інформацію про попередні звернення. Такий підхід не лише спрощує горизонтальне масштабування — кожен новий контейнер без стану одразу стає повноцінним учасником обробки трафіку, — але й полегшує відмовостійкість: у разі аварії одного інстансу інші продовжують обслуговувати запити без необхідності відновлювати розподілені сесії.

Ще одним наслідком stateless-підходу є можливість легко інтегрувати механізми circuit-breaker і rate-limiting на рівні клієнтських бібліотек або проксів, оскільки кожен виклик є ізольованим та ідемпотентним. Це дозволяє плавно реагувати на короткочасні піки навантаження: у момент дефіциту ресурсів можна

просто обмежити кількість паралельних викликів, не ризикуючи втратити сесійну інформацію чи порушити консистентність.

2.8.2 Горизонтальне масштабування та автоматичне розгортання

Оскільки програма не зберігає жодних локальних даних, для збільшення обчислювальної потужності достатньо запустити додаткові екземпляри того самого Docker-образу. Це можна зробити за допомогою docker-compose --scale service=n або в хмарі шляхом розгортання кількох інстансів одного контейнера. HTTP-балансувальник (NGINX, HAProxy чи будь-яка хмарна LB-служба) автоматично рівномірно розподіляє вхідні запити поміж усіма активними репліками, при цьому health-check-endpoint гарантує, що тільки здорові інстанси потрапляють у пул обробників.

Якщо навантаження під час піка перевищує поточні можливості, додавання нових контейнерів відбувається без простоїв і змін у коді. Усі екземпляри підключаються до єдиної бази даних і до централізованого пулу з’єднань HikariCP, отримуючи нові з’єднання за лічені мілісекунди. Такий метод масштабування дозволяє не лише підтримувати стабільну пропускну здатність, але й легко впроваджувати стратегічні оновлення: нові версії програми можна розгортати пакетно або по одній репліці за схемою blue-green, минимально впливаючи на користувацький трафік.

2.8.3 Гарантування цілісності через транзакційність

Усередині сервісного шару найважливіші методи, що виконують складні операції з кількома таблицями, анотовані @Transactional. Це забезпечує, що всі дії в межах виклику вважаються єдиною атомарною операцією: при виникненні помилки виконуються автоматичний rollback усіх змін, тим самим виключаючи часткову або неконсистентну модифікацію даних. Вибір рівня ізоляції транзакцій (за замовчуванням READ_COMMITTED) та використання оптимізованих batch-

вставок спрощують виконання одночасних запитів із сотнями паралельних з’єднань, не порушуючи цілісності.

2.8.4 Оптимізація доступу до даних: кешування та реплікація

В програмі передбачено єдину абстракцію кешування на рівні сервісів, яка може бути реалізована як через зовнішній Redis, так і через in-memory рішення (наприклад, Caffeine). При підключенні Redis всі репліки сервісу звертаються до спільного кеш-шару з налаштованим часом життя записів, що позбавляє необхідності повторного виконання складних SQL-запитів і економить ресурси бази даних. Навіть із врахуванням мережевої затримки (декілька мілісекунд на RTT) загальний час відповіді залишається нижчим за повний обіг до MySQL, а одноточкові запити до Redis добре масштабуються горизонтально без додаткового навантаження на JVM.

Якщо ж обрати in-memory кеш, усі найчастіше запитані дані зберігаються безпосередньо в heap-пулі кожного сервісного інстансу, що дає практично миттєвий доступ без жодних зовнішніх викликів. Це ідеально підходить для ситуацій із низькою частотою змін у даних і невеликими об’ємами “гарячих” ключів, при цьому слід уважно налаштувати обсяг кеш-пам’яті, щоб уникнути переповнення heap-пулу. Одночасно відсутність єдиної точки кешування може призводити до незначної надмірності даних у різних інстансах, але це не впливає на кінцевого користувача, оскільки всі запити обслуговуються з аналогічною низькою латентністю.

Завдяки тому, що кешування вбудоване на рівні Spring Cache із підтримкою кількох менеджерів, перехід між Redis і in-memory можна здійснити лише зміною конфігурації без жодних правок бізнес-логіки. TTL та політики оновлення забезпечують актуальність даних у обох варіантах, а прозорий fallback на базу даних гарантує, що у випадку недоступності кешу клієнт не помітить жодного деградаційного відчуття в швидкодії.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

Після визначення вимог і вибору технологічного стеку розробка перейшла у фазу реалізації. На цьому етапі було спроєктовано наскрізну архітектуру “база даних — бекенд — клієнт” з обміном даними через REST-інтерфейс, імплементовано алгоритми просторової агрегації з використанням QuadKey, а також розроблено механізм динамічного вибору рівня деталізації карти залежно від масштабу. Далі подано опис ключових компонентів і застосованих математичних підходів. Для детального ознайомлення з вихідним кодом ключових компонентів див. Додаток А (лістинги А1–А9). Далі подано опис ключових компонентів і застосованих підходів.

3.1 Трирівнева архітектура DB — BE — FE

Ядром системи є реляційна база MySQL, що зберігає дві головні таблиці — nets (метадані мереж) та geo (географічні координати та QuadKey). В обох таблицях на BSSID визначено унікальний індекс типу B-tree, а в geo додатково створено просторовий індекс idx_geo_lat_lon_quadkey, який охоплює колонки latitude, longitude та quadkey. Це дає змогу виконувати швидке вибіркове сканування діапазону QuadKey і водночас фільтрувати за координатами.

Бекенд, побудований на Spring Boot 3, виступає проміжним шаром між клієнтом і базою. Контролери приймають JSON-запити, серіалізують їх у DTO й передають у сервісний шар. Усі звернення до БД реалізовано через Spring Data JPA, де GeoRepository містить нативні SQL-запити для агрегування кластерів і підвантаження одиничних точок доступу. Сервіс GeoService додатково перевіряє коректність запитуваної області та обмежує максимальну кількість точок у відповіді.

Клієнтським рівнем є Android-застосунок, написаний на Kotlin/Java з UI у Jetpack Compose. Мережевий стек ґрунтується на OkHttp 5, що надсилає POST-

запити на /api/v1/nets і /api/v1/map/area. Отримані JSON-дані перетворюються на внутрішні моделі (Net, PointDto, ClusterDto) і відображаються або списком, або маркерами на карті Google Maps.

3.2 Реалізація коду QuadKey у сервісі геоданих

Алгоритм формування QuadKey стартує з “вікна” – глобального прямокутника [–90°, +90°] × [–180°, +180°]. Далі, протягом фіксованої кількості ітерацій, яка відповідає обраному рівню деталізації, цей простір щоразу ділиться навпіл по широті та довготі. На кожному кроці обчислюються центральні координати поточного вікна; порівняння цих центроїдів із вхідними значеннями визначає квадрант (північ-захід, північ-схід, південь-захід, південь-схід), куди потрапляє точка. Комбінація двох бітів – старший (“північ / південь”) та молодший (“захід / схід”) – утворює код квадранта на поточному рівні. Щоб зібрати всі коди в єдине 64-бітне число, накопичений ключ щоразу зсувається вліво на два біти, а новий фрагмент додається операцією OR. Найстарші біти відповідають першим, грубішим поділам, а наймолодші – фінальним, найдрібнішим.

У проєкті логіка інкапсулюється в утилітному класі QuadKey (пакет com.tairova.tarik.service). Метод encode(lat, lon, level) реалізує описану послідовність без проміжних алокацій, тож його виконання займає сталий час O(level) та потребує лише кількох бітових операцій. Параметр level задається константою DB_QUADKEY_LEVEL у центральному класі Constants.Geo, що дозволяє миттєво змінювати глибину деталізації для всієї системи.

При збереженні нової точки сутність Geo (пакет persistence.entity) отримує значення quadkey безпосередньо в методі NetService.saveNets(). Далі через Spring Data JPA виклик geoRepository.save(geo) передає готовий ключ у колонку quadkey. У MySQL по цьому полю створено B-tree-індекс idx_geo_quadkey_lat_lon; завдяки йому діапазонний фільтр quadkey BETWEEN X AND Y відпрацьовує першочергово, а уточнення по latitude та longitude відсікає залишкові рядки. Це дає

змогу виконувати запити у щільних міських зонах за десятки мілісекунд навіть при десятках мільйонів записів.

Таким чином, розробники отримують компактний, тестопридатний механізм просторового індексування, що легко інтегрується з ORM-шаром і не потребує складних налаштувань на рівні бази даних чи додаткових структур у пам’яті.

3.3 Організація кластеризації та вибірки одиничних точок

Сервіс GeoService координує двоетапний алгоритм кластеризації. Перший етап виконується за допомогою методу aggregateByCluster() інтерфейсу GeoRepository, у якому за допомогою нативного SQL-запиту з анотацією @Query(nativeQuery = true) відбувається групування всіх записів з таблиці geo за значенням (quadkey >> shift). Параметр shift обчислюється у GeoService викликом estimateLevelByGeometry(), що забезпечує адаптивний вибір глибини агрегування залежно від розмірів прямокутника запиту. Після виконання цієї операції на рівні бази даних повертається проекція ClusterProjection, з якої у сервісі створюються DTO ClusterDto з усередненими координатами та числом точок.

Другий етап — добіркове сканування “синглтонів” — реалізовано в класі GeoRepositoryImpl. Використовуючи EntityManager.createNativeQuery(), динамічно будується умова WHERE (g.quadkey BETWEEN :start0 AND :end0) OR … для кожного кластеру з єдиною точкою. Після встановлення параметрів діапазонів та виконання запиту результати у вигляді Object[] перетворюються на PointDto у методі toDto(), де числовий BSSID знову конвертується в текстовий формат через сервіс BssidService. Завдяки встановленню ліміту на рівні SQL-запиту та подальшому фільтруванню результатів у Java-коді клієнт отримує передбачувану швидкість відповіді та чисте відображення точок без “шуму” на карті.

3.4 Програмна перевірка, валідація та адміністрування даних

У модулі NetService метод saveNets(NetCreateRequest request) анотовано @Transactional, що гарантує цілісність операцій: всі вставки до таблиць geo та nets виконуються в межах однієї транзакції і в разі помилки автоматично відкочуються. Перед записом кожного елементу виконується validateNetItem(), який виконує перевірку, що складається з 3 пунктів:

1) викликає bssidService.validateBssid() для перевірки формату MAC-адреси;
2) перевіряє широту й довготу на відповідність константам MIN_LATITUDE…MAX_LATITUDE;
3) контролює довжину рядків ESSID та WiFiKey.

При невідповідності будь-якої з умов кидається IllegalArgumentException або спеціалізована GeoException, що дозволяє зупинити некоректну вставку ще до звернення до бази.

На рівні СУБД таблиця nets має складний унікальний індекс:
UNIQUE (NoBSSID, BSSID, ESSID, NoWiFiKey, WiFiKey, NoWPS, WPSPIN)
— який автоматично відхиляє дублікати. Помилки типу DataIntegrityViolationException обробляються в глобальному @ControllerAdvice, що повертає клієнту зрозуміле повідомлення без розкриття внутрішньої схеми.

Ключовим доповненням до цих механізмів служить підтримка актуальності бази даних адміністраторами через виділені REST-ендпоінти. Адмін може завантажувати оновлені набори записів у форматі JSON або CSV, використовуючи захищені шляхи /admin/upload/json та /admin/upload/csv. Ці ендпоінти мають обмежений доступ за ролями та забезпечують 2 пункти:

1) пакетну обробку нових або змінених записів із повною валідацією перед вставкою;
2) можливість автоматизувати регулярне оновлення через зовнішні скрипти або CI/CD-конвеєр.

Поєднання транзакційності, низькорівневої валідації, жорстких унікальних індексів і спеціалізованих адміністративних каналів імпорту гарантує не лише високу якість даних, а й своєчасне оновлення інформації в базі без додаткових навантажень на звичайні клієнтські запити.

3.5 Діаграма класів сутностей

На рисунку 3.1 наведено спрощену UML-діаграму двох ключових JPA-сутностей — Net і Geo, їхні поля та зв’язки.

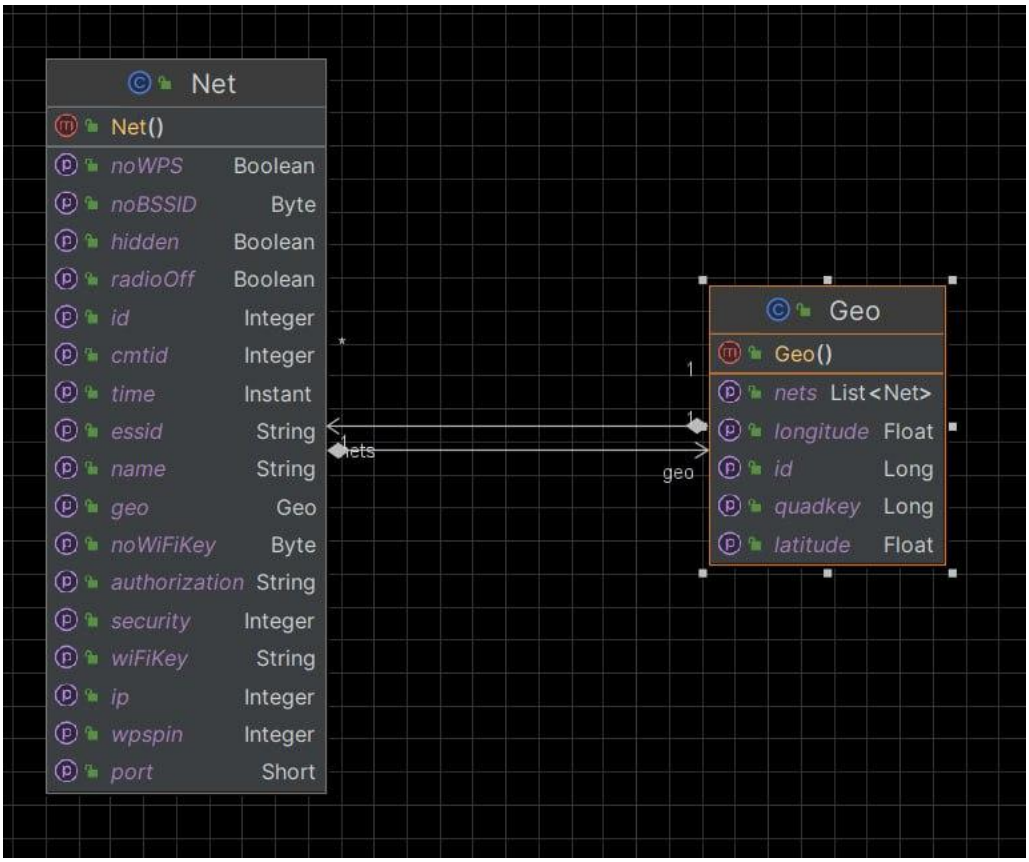

Рисунок 3.1 — Діаграма класів сутностей

Сутність Net містить усі атрибути, що описують окрему точку доступу: від ідентифікаторів (id, bssid, essid) до метаданих безпеки та стану (security, hidden, noWPS тощо). Поле geo у класі Net позначене як зв’язок багато-до-одного з класом

Geo, що означає: кожен запис Net пов’язується з рівнем просторової індексації, збереженим у відповідному об’єкті Geo.

Сутність Geo виступає агрегаційним контейнером для групи Net-записів, поєднуючи атрибути геолокації (latitude, longitude) та довгий ключ quadkey, який використовується для просторового індексування. З точки зору JPA-асоціацій, Geo містить список List<Net> nets, а Net — посилання на свою батьківську гео-точку. Така організація дає можливість за єдиним запитом завантажувати і всю інформацію про властивості доступу (через Net), і відповідну групу геопросторових атрибутів (через Geo), зберігаючи цілісність об’єктної моделі та забезпечуючи оптимальне використання B-tree-індексу по полю quadkey.

Цей підхід спрощує завдання фільтрації та агрегації: при виконанні пошуку по quadkey спочатку вибираються відповідні об’єкти Geo, а пов’язані з ними Net-записи можна підвантажити через налаштований fetch-режим або окремий JPQL-запит. Завдяки чіткій структурі класів і близькому відображенню на реляційну схему забезпечується прозоре й масштабоване збереження як окремих мережевих записів, так і їх просторових контекстів.

3.6 Діаграма основних сервісів

На рисунку 3.2 відображено три центральні компоненти бізнес-логіки модуля роботи з мережами – BssidService, GeoService і NetService, а також їхні взаємозв’язки.

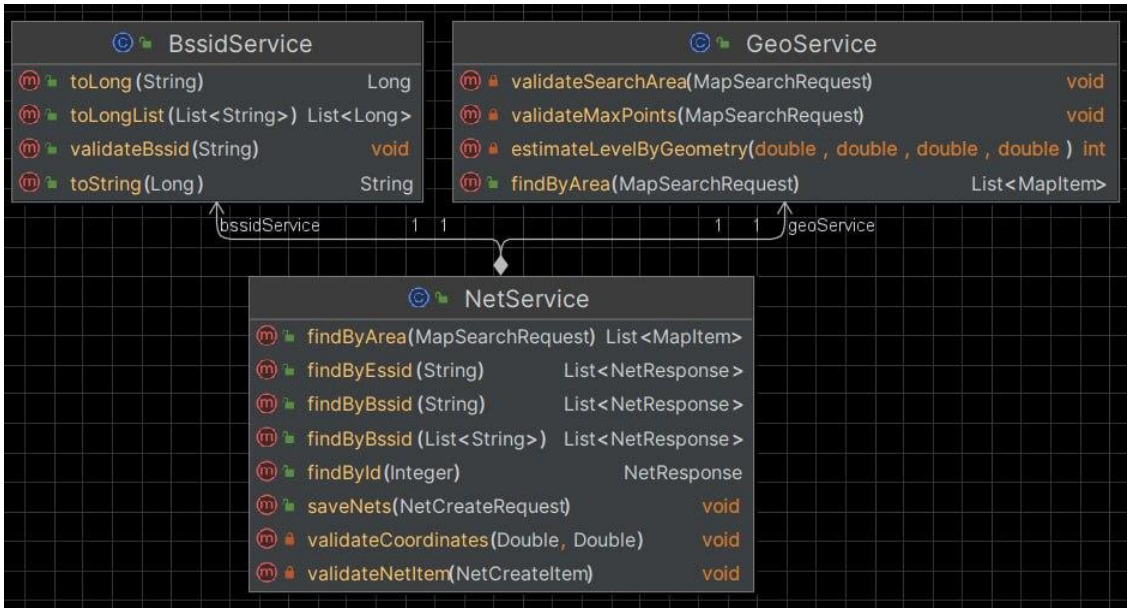

Рисунок 3.2 — Діаграма основних сервісів

BssidService сконцентроване на коректному перетворенні та валідації MAC-адрес: воно перетворює рядкові представлення BSSID у числовий формат для зберігання й назад у звичний вигляд із двокрапками, а також перевіряє синтаксис адрес перед подальшими операціями. GeoService відповідає за всю просторову логіку: від початкової валідації кордонів області та оцінки орієнтовної кількості точок до динамічного вибору рівня тайлів і формування готового списку маркерів для відображення на карті. NetService виступає фасадом, який згуртовує роботу обох підсистем та надає єдиний API для клієнта – методи пошуку за ESSID, BSSID або географічним прямокутником, масове збереження нових мережевих записів і додаткові перевірки коректності даних перед збереженням. Згідно з діаграмою, NetService звертається до BssidService, щоб перетворити й перевірити BSSID, і передає GeoService завдання щодо валідації та агрегування просторових даних, а результатом роботи всіх трьох сервісів є консолідований набір DTO, готовий до відправки у відповідь клієнту.

3.7 Готовність сервісів до балансування та масштабування

У системі всі сервіси (NetService, GeoService) розроблені як безстанні компоненти Spring (@Service), а вся конфігурація (параметри nets.limit, налаштування HikariCP тощо) винесена у зовнішні файли application-{profile}.yml і підвантажується через @Value та @ConfigurationProperties, що відповідає принципам “12-factor app” щодо розділення коду та конфігурації. Пакетування в один Docker-контейнер не обмежує масштабування: для збільшення пропускної здатності достатньо запустити декілька екземплярів цього контейнера (через docker-compose --scale або віддалено в хмарі) й направити їхній трафік через стандартний балансувальник навантаження (наприклад, NGINX або HAProxy).

Завдяки вбудованому Spring Actuator кожен інстанс сервісу надає health-check та метрики, які можна опитувати зовнішніми системами моніторингу й автоматизувати розгортання нових реплік у разі підвищеного навантаження. Незалежне зберігання стану в MySQL гарантує, що жодна репліка не тримає локальних сесій або кешу, а всі з’єднання й ресурси керуються централізовано через пул HikariCP. Така архітектура дозволяє без жодних змін у коді масштабуватися горизонтально, підтримувати високу доступність і стабільну продуктивність навіть при різкому зростанні трафіку.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Розроблена система є інтуїтивно зрозумілим мобільним додатком, який дозволяє користувачеві сканувати доступні Wi-Fi мережі поблизу, дізнаватися їхні паролі (якщо вони наявні в базі даних), переглядати розташування точок доступу на мапі та взаємодіяти з картою для пошуку мереж у різних районах. Основна ціль — зробити доступ до мереж максимально швидким, простим і безпечним.

4.1 Системні вимоги та інсталяція

Для початку роботи з програмою важливо забезпечити відповідне оточення на Android-пристрої. Додаток протестовано на платформі Android 15.0, проте для досягнення максимальної стабільності всіх ключових компонентів рекомендується використовувати не менше ніж Android 9.0 (Pie, API 28). Саме з цієї версії ОС Jetpack Compose працює в найстабільнішому режимі, фонова процедура сканування Wi-Fi мереж за запитом на ACCESS_FINE_LOCATION не підпадає під застарілі обмеження, а більшість сучасних пристроїв із архітектурою ARMv8 та підтримкою OpenGL ES 3.1 вже постачаються з Android 9.0 або вище.

Щодо апаратних вимог, для безперебійної роботи інтерфейсу та сканування мереж без критичних затримок у UI рекомендується мати не менше ніж 2 ГБ оперативної пам’яті та процесор ARMv8 із підтримкою OpenGL ES 3.1.

Інсталяція програми можлива через стандартний пакет Android Package (APK). При першому запуску користувачеві необхідно надати дозвіл на доступ до геолокації, оскільки Android забороняє фонове сканування мереж без цього дозволу. Окрім того, для коректної роботи мапи потрібно забезпечити підключення до Інтернету.

Процес інсталяції і запуску відбувається у три кроки:

1. Завантаження APK.
2. Надання системних дозволів (геолокація, доступ до локальних мереж).
3. Початковий запуск із запитом адреси сервера (якщо вона не налаштована за замовчуванням).

У результаті користувач бачить головний екран із переліком мереж (рисунок 4.1) і може відразу перейти до сканування або картографічного відображення.

Рисунок 4.1 — Головний екран

4.2 Інтерфейс головного екрану

Головний екран програми виконує роль оперативної панелі користувача, об’єднуючи список усіх виявлених довколишніх точок доступу та сукупність елементів швидкого доступу до ключових функцій. У верхній частині екрану розташована панель з назвою програми “Tarik Wi-Fi” та набором іконок: кнопка довідки у вигляді знака питання, що відкриває контекстний мануал; іконка налаштувань, за допомогою якої користувач може змінити адресу сервера та інші параметри; кнопка оновлення у вигляді кругової стрілки — при її натисканні або під час автоматичного запуску інтерфейс миттєво виконує нове локальне сканування, оновлюючи перелік мереж поточними значеннями рівня сигналу (dBm) і BSSID, якщо в базі сервера знайдено пароль для конкретного BSSID, віджет автоматично забарвлюється в зелений колір із контрастним текстом, що дозволяє легко відрізнити “доступні” мережі від тих, у яких паролі відсутні (рисунок 4.2); поруч із цим розташована кнопка пошуку, яка дозволяє відфільтрувати список за введеним SSID; а в правій частині — іконка карти, що переводить користувача на географічне відображення точок доступу у вигляді маркерів.

Рисунок 4.2 — Головний екран із знайденим паролем

Нижче розташований власне список мереж, кожна з яких представлена у вигляді картки з плавно заокругленими кутами, наповненою відтінком темної палітри інтерфейсу. У шапці картки відображається назва мережі (SSID) та її унікальний ідентифікатор (BSSID). Одразу під цим розміщується поле “Пароль”, яке порожнє за замовчуванням, але в разі наявності в базі сервера відповідного запису автоматично заповнюється текстом і підсвічується зеленим кольором із контрастним білим шрифтом, що одразу вирізняє мережі з відомими паролями. У нижній частині кожної картки зазначається поточний рівень сигналу у dBm та розташовується невелика іконка-кнопка копіювання: при торканні до цієї іконки

збережений пароль миттєво потрапляє в буфер обміну, а в статусному рядку з’являється сповіщення про успішну операцію.

Завдяки такій організації інформації користувач отримує інтуїтивно зрозумілий і водночас вичерпний огляд довкілля Wi-Fi мереж з можливістю оперативного реагування: одним натисканням іконки оновлення отримає найсвіжішу картину доступних точок, а клацанням по картці або кнопці копіювання — автоматично переходить до встановлення з’єднання чи збереження пароля. Такий підхід до побудови головного екрану забезпечує баланс між компактністю інтерфейсу та його функціональністю, гарантує швидкодію навіть при обробці сотень записів і дозволяє користувачу працювати в режимі реального часу без зайвих затримок.

4.3 Мапа та кластеризація

Перейшовши на вкладку карти, користувач отримує інтерактивний мап-екран, у якому маркери точок доступу динамічно групуються в кластери. При великому масштабі (низький рівень зуму) система агрегує тисячі близьких точок у “хмарки” з числом в середині кола (рисунок 4.3).

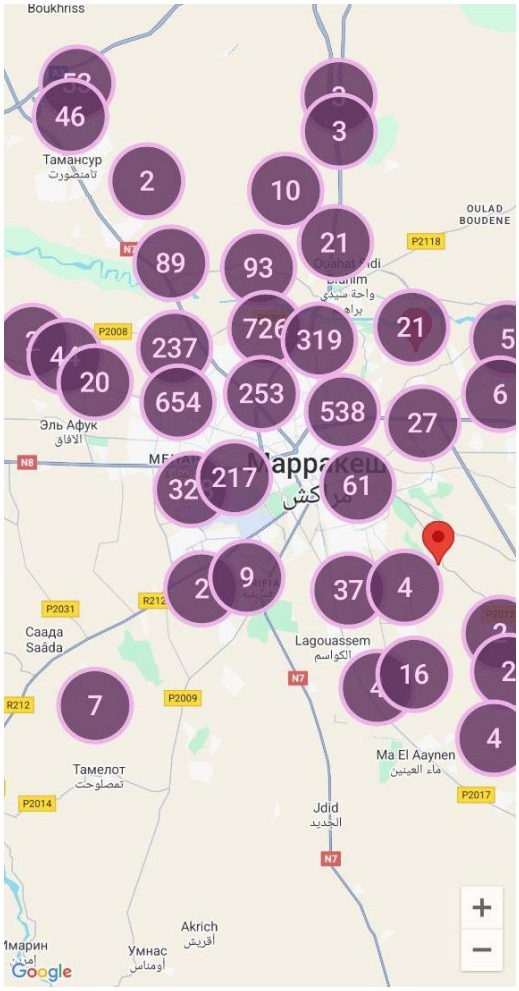

Рисунок 4.3 — Кластеризація точок доступу на великому віддалені

Кожен кластер обчислюється за допомогою QuadKey: на основі географічної області та обраного рівня деталізації QuadKey обрізається зсувом, що формує групи точок.

При зближенні (pinch-to-zoom або подвійному тапі) клієнт відправляє на сервер новий запит із оновленими координатами та рівнем кластеризації, а сервер повертає або більшу кількість дрібніших кластерів, або окремі точки доступу (рисунок 4.4).

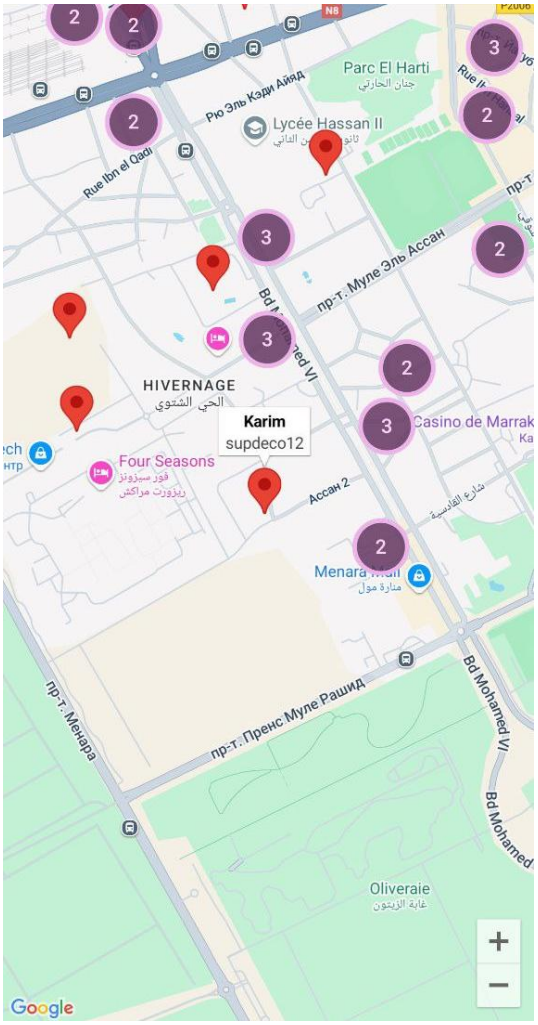

Рисунок 4.4 — Кластеризація точок доступу при приближенні

Такий підхід дозволяє уникнути перевантаження карти сотнями маркерів і зберегти читабельність при будь-якому масштабі.

4.4 Налаштування та допомога

На екрані “Налаштування” (рисунок 4.5) зверху видно кнопку “Назад” і заголовок “Налаштування”, під яким розміщено одне поле вводу (OutlinedTextField) із міткою “Адреса” для введення IP або доменного імені сервера.

Рисунок 4.5 — Налаштування адреси сервера у застосунку

Усі зміни миттєво зберігаються в SharedPreferences і підвантажуються при наступному запуску, тож введена адреса залишається доступною після перезавантаження програми. Після завершення сканування зверху накладається невелика плащка (snackbar) з текстом на кшталт “Знайдено 22 мережі”, яка повідомляє користувача про кількість знайдених Wi-Fi мереж точок.

На екрані “Допомога” (рисунок 5.6) зверху розташовано кнопку “Назад” і заголовок “Допомога”.

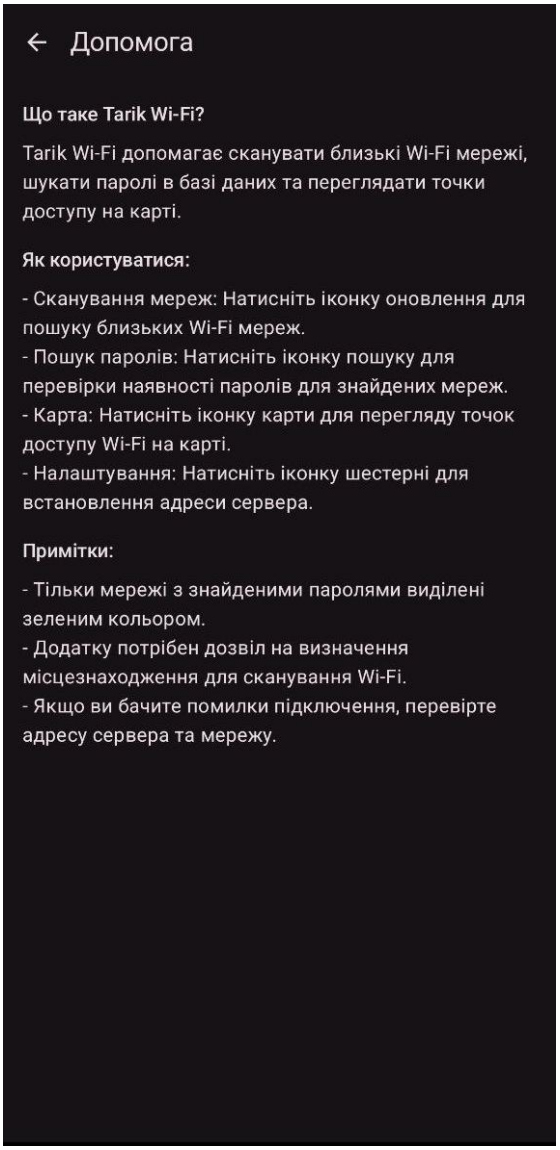

Рисунок 4.6 — Екран довідки з коротким описом роботи застосунку

Основна область складається з трьох текстових блоків, відокремлених заголовками — спершу “Що таке Tarik Wi-Fi?”, потім “Як користуватися:” і нарешті “Примітки:”.

Під заголовком “Що таке Tarik Wi-Fi?” подано короткий опис: додаток сканує близькі Wi-Fi мережі, шукає їхні паролі в базі та відображає точки доступу на карті. Далі під “Як користуватися:” наведено чотири пункти дії:

1) оновлення сканування мережі через кнопку “Refresh”;
2) перевірка паролів через кнопку “Search”;
3) перегляд точок на карті через кнопку “Map”;

4) налаштування адреси сервера через кнопку “Settings”.

У розділі “Примітки:” перераховані важливі зауваження: лише мережі зі знайденими паролями підсвічуються зеленим; для сканування потрібен дозвіл на місцезнаходження; у разі помилок підключення слід перевірити адресу сервера та доступ до мережі.

Таким чином, програма об’єднує в єдиному інтерфейсі локальне сканування, відправку запитів на сервер і візуалізацію результатів, що дозволяє вирішувати повсякденні завдання з пошуку та підключення до бездротових мереж буквально в кілька дотиків. Завдяки Jetpack Compose інтерфейс адаптивно реагує на стан сканування і запитів, миттєво відображаючи як успішно знайдені паролі, так і попередження про помилки без зайвого очікування чи перезавантаження екрана. Механізм збереження адреси сервера в SharedPreferences повністю знімає необхідність повторного введення налаштувань, тоді як кольорове підсвічування карток із валідними паролями робить досвід користувача інтуїтивно зрозумілим і безпечним. Вбудовані Toast-повідомлення та Snackbar-квитки інформують про хід операцій і їх результати, мінімізуючи необхідність ручного контролю. У сукупності це забезпечує високий рівень автоматизації та комфорт користування навіть для тих, хто не має глибоких технічних навичок.

ВИСНОВКИ

У цій роботі було розроблено та досліджено клієнт-серверну систему для зберігання й швидкого пошуку точок доступу у великій базі даних (>12 млн записів). Запропоновані підходи поєднують класичний B-Tree-індекс для точкового пошуку BSSID і QuadKey з діапазонними запитами для геопошуку, що забезпечує швидку відповідь навіть при високому навантаженні.

Проведене тестування підтвердило високу ефективність алгоритмів кластеризації: обмеження максимальної кількості маркерів гарантує читабельність карти без втрати важливої інформації, а двоетапна агрегація Cluster/Point адаптується до будь-якого рівня масштабування.

Розроблений мобільний клієнт на Android (Kotlin + Jetpack Compose) у тандемі з бекендом на Spring Boot і базою MySQL продемонстрував зручність і стабільність роботи: автоматичне сканування локальних мереж, синхронізація з хмарою та інтерактивне відображення результатів на карті роблять систему придатною для туристів, студентів і приймальних комісій.

Запропонована інфраструктурна архітектура гарантує горизонтальне масштабування й стійкість до пікових навантажень, що підвищує готовність рішення до реального розгортання.

Подальші дослідження можуть бути спрямовані на впровадження Wi-Fi fingerprinting для більш точної геолокації, реалізацію модуля автоматичного підбору гіперпараметрів кластеризації та додавання краудсорсингових механізмів збору даних.

Таким чином, результати цієї роботи підтверджують суттєвий потенціал розробленої системи для оптимізації процесу пошуку й підключення до Wi-Fi мереж у різних сценаріях використання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cisco Systems. Cisco Annual Internet Report (2018 – 2023) White Paper [Електронний ресурс]. — San Jose : Cisco Systems, 2020. — 46 с. — Режим доступу: https://www.cisco.com/go/air (дата звернення: 13.05.2025).
2. ITU Facts and Figures 2023 : Measuring digital development [Електронний ресурс]. — International Telecommunication Union, 2023. — Режим доступу: https://www.itu.int/itu-d/reports/statistics/facts-figures-2023 (дата звернення 12.05.2025).
3. Fielding R., Reschke J. Hypertext Transfer Protocol — HTTP/1.1 (RFC 7230-7235). — IETF, 2014.
4. WiGLE. Wireless Geographic Logging Engine [Електронний ресурс]. — Режим доступу: https://wigle.net (дата звернення: 12.05.2025).
5. Lua E. K. та ін. A survey and comparison of peer-to-peer overlay network schemes // IEEE Communications Surveys & Tutorials. — 2005. — Т. 7, № 2. — С. 72-93.
6. Bass L., Clements P., Kazman R. Software Architecture in Practice. — 4-th ed. — Boston : Addison-Wesley, 2021. — 624 с.
7. WiFi Map — Free Passwords & Hotspots [Електронний ресурс]. — Режим доступу: https://www.wifimap.io (дата звернення 12.05.2025).
8. Instabridge — WiFi Passwords [Електронний ресурс]. — Режим доступу: https://instabridge.com (дата звернення 12.05.2025).
9. Wiman Free WiFi Manager [Електронний ресурс]. — Режим доступу: https://wiman.me (дата звернення 12.05.2025).
10. Microsoft. Bing Maps Tile System [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/en-us/bingmaps/articles/bing-maps-tile-system (дата звернення: 30.05.2025).
11. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: PhD Thesis / R. T. Fielding. — Irvine, CA: University of California, 2000. — 475 p.

ДОДАТОК А.
РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ПОШУКУ Wi-Fi МЕРЕЖ

Програмні засоби

– мова програмування – Java 17;

– веб-фреймворк – Spring Boot 3 (Spring Web MVC, Spring Web);

– серіалізація/десеріалізація JSON – Jackson;

– ORM та робота з базою даних – Spring Data JPA → MySQL;

– специфікація JPA – Jakarta Persistence API;

– логування – SLF4J через Lombok;

– генерація boilerplate-коду – Lombok);

– обробка HTTP-запитів та multipart-форм – Spring MVC;

– парсинг CSV – власний сервіс CsvParserService із використанням MultipartFile;

– кластеризація точок – власний алгоритм QuadKey у класі QuadKey;

– обробка колекцій та стрімів – Java Stream API;

– транзакційність – Spring Transaction.

Лістинг А1 – NetMapController.java

```
package com.tairova.tarik.controller;


import com.tairova.tarik.dto.ClusterDto;
import com.tairova.tarik.dto.MapItem;
import com.tairova.tarik.dto.MapSearchRequest;
import com.tairova.tarik.service.NetService;
import lombok.RequiredArgsConstructor;
```

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/v1/map")
@RequiredArgsConstructor
@Slf4j
public class NetMapController {

    private final NetService netService;

    @PostMapping("/area")
    public List<MapItem> findInArea(@RequestBody MapSearchRequest request) {
        log.info("Request: {}", request);
        long start = System.currentTimeMillis();
        List<MapItem> result = netService.findByArea(request);
        long end = System.currentTimeMillis();
        log.info("Request took: {}ms", (end - start));
        log.info("Returned {} items", result.size());
        log.info("Returned {} results with clusters data", getCountWithClusters(result));
        return result;
    }

    private int getCountWithClusters(List<MapItem> result) {
        return result.stream()
                .mapToInt(item -> {
                    if (item instanceof ClusterDto clusterDto) {
                        return Math.toIntExact(clusterDto.getCount());
```

```
                } else {
                    return 1;
                }
            })
            .sum();
    }
}
```

Лістинг А2 – NetController.java

```
package com.tairova.tarik.controller;

import com.tairova.tarik.dto.GeoResponse;
import com.tairova.tarik.dto.NetResponse;
import com.tairova.tarik.dto.NetSearchRequest;
import com.tairova.tarik.service.NetService;
import java.util.ArrayList;
import java.util.List;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequiredArgsConstructor
@RequestMapping("/api/v1/nets")
public class NetController {
    private final NetService netService;
```

```
    @PostMapping
    public List<NetResponse> findByBssidList(@RequestBody NetSearchRequest
netSearchRequest) {
        List<NetResponse> result = new
ArrayList<>(netService.findByBssid(netSearchRequest.getBssids()));
        result.add(getNetResponse());
        return result;
    }

    private NetResponse getNetResponse() {
        NetResponse netResponse = new NetResponse();
        netResponse.setEssid("Vodafone-8244");
        netResponse.setWiFiKey("12345678");
        netResponse.setGeo(new GeoResponse("1c:9d:72:58:82:50", null, null, null));
        return netResponse;
    }
}
```

Лістинг А3 – NewNetController.java

```
package com.tairova.tarik.controller;

import com.tairova.tarik.dto.NetCreateRequest;
import com.tairova.tarik.service.CsvParserService;
import com.tairova.tarik.service.NetService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

@RestController
@RequestMapping("/api/v1/new-nets")
@RequiredArgsConstructor
@Slf4j
public class NewNetController {
    private final NetService netService;
    private final CsvParserService csvParserService;

    @PostMapping(path = "/json", consumes =
MediaType.APPLICATION_JSON_VALUE)
    public void createNetsFromJson(@RequestBody NetCreateRequest request) {
        log.info("Received JSON request with {} networks", request.getNets().size());
        netService.saveNets(request);
    }

    @PostMapping(path = "/csv", consumes =
MediaType.MULTIPART_FORM_DATA_VALUE)
    public void createNetsFromCsv(MultipartFile file) {
        log.info("Received CSV file: {}", file.getOriginalFilename());
        NetCreateRequest request = csvParserService.parseCsv(file);
        log.info("Parsed {} networks from CSV", request.getNets().size());
        netService.saveNets(request);
    }
}
```

Лістинг А4 – NetService.java

```
package com.tairova.tarik.service;

import com.tairova.tarik.dto.MapItem;
import com.tairova.tarik.dto.MapSearchRequest;
import com.tairova.tarik.dto.NetCreateRequest;
import com.tairova.tarik.dto.NetResponse;
import com.tairova.tarik.exception.GeoException;
import com.tairova.tarik.mapper.NetMapper;
import com.tairova.tarik.persistence.entity.Geo;
import com.tairova.tarik.persistence.entity.Net;
import com.tairova.tarik.persistence.repository.GeoRepository;
import com.tairova.tarik.persistence.repository.NetRepository;
import java.util.List;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.domain.Limit;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import static com.tairova.tarik.config.Constants.Geo.MAX_LATITUDE;
import static com.tairova.tarik.config.Constants.Geo.MIN_LATITUDE;
import static com.tairova.tarik.config.Constants.Geo.MAX_LONGITUDE;
import static com.tairova.tarik.config.Constants.Geo.MIN_LONGITUDE;

@RequiredArgsConstructor
@Service
public class NetService {
    private final NetRepository netRepository;
```

```
private final NetMapper netMapper;
private final BssidService bssidService;
private final GeoService geoService;
private final GeoRepository geoRepository;

@Value("${nets.limit}")
private Integer limit;

private static final int MAX_ESSID_LENGTH = 32;
private static final int MAX_WIFI_KEY_LENGTH = 64;

public NetResponse findById(Integer id) {
   return netRepository.findById(id)
         .map(netMapper::toNetResponse)
         .orElseThrow();
}

public List<NetResponse> findByEssid(String essid) {
   List<Net> result = netRepository.findByEssid(essid, Limit.of(limit));
   return result.stream()
         .map(netMapper::toNetResponse)
         .toList();
}

public List<NetResponse> findByBssid(String bssid) {
   Long bssidLong = bssidService.toLong(bssid);
   List<Net> result = netRepository.findByGeo_Id(bssidLong, Limit.of(limit));
   return result.stream()
         .map(netMapper::toNetResponse)
         .toList();
```

```
}

public List<NetResponse> findByBssid(List<String> bssids) {
   List<Long> numBssids = bssidService.toLongList(bssids);
   return netRepository.findByGeo_IdIn(numBssids)
         .stream()
         .map(netMapper::toNetResponse)
         .toList();
}

/**
 * Finds points and clusters in the specified geographic area.
 * Uses quadtree for efficient point clustering.
 */
public List<MapItem> findByArea(MapSearchRequest request) {
   return geoService.findByArea(request);
}

@Transactional
public void saveNets(NetCreateRequest request) {
   if (request == null || request.getNets() == null) {
      throw new IllegalArgumentException("Request or nets list cannot be null");
   }

   for (NetCreateRequest.NetCreateItem item : request.getNets()) {
      validateNetItem(item);

      // Create or update Geo
      Geo geo = new Geo();
      geo.setId(bssidService.toLong(item.getBssid()));
```

```
            geo.setLatitude(item.getLatitude().floatValue());
            geo.setLongitude(item.getLongitude().floatValue());
            geo = geoRepository.save(geo);

            // Create Net
            Net net = new Net();
            net.setGeo(geo);
            net.setEssid(item.getEssid());
            net.setWiFiKey(item.getWiFiKey());
            net.setRadioOff(item.getRadioOff() != null ? item.getRadioOff() : false);
            net.setHidden(item.getHidden() != null ? item.getHidden() : false);
            net.setNoBSSID(item.getNoBSSID() != null ? item.getNoBSSID() : (byte) 0);
            net.setSecurity(item.getSecurity());
            net.setNoWiFiKey(item.getNoWiFiKey() != null ? item.getNoWiFiKey() :
(byte) 0);
            net.setNoWPS(item.getNoWPS() != null ? item.getNoWPS() : false);
            net.setWpspin(item.getWpspin());

            netRepository.save(net);
        }
    }

    private void validateNetItem(NetCreateRequest.NetCreateItem item) {
        // Validate BSSID
        bssidService.validateBssid(item.getBssid());

        // Validate coordinates
        validateCoordinates(item.getLatitude(), item.getLongitude());

        // Validate ESSID
```

```
        if (item.getEssid() != null && item.getEssid().length() > MAX_ESSID_LENGTH)
{
            throw new IllegalArgumentException("ESSID length exceeds maximum of " +
MAX_ESSID_LENGTH + " characters");
        }

        // Validate WiFiKey
        if (item.getWiFiKey() != null && item.getWiFiKey().length() >
MAX_WIFI_KEY_LENGTH) {
            throw new IllegalArgumentException("WiFiKey length exceeds maximum of "
+ MAX_WIFI_KEY_LENGTH + " characters");
        }
    }

    private void validateCoordinates(Double latitude, Double longitude) {
        if (latitude == null || longitude == null) {
            throw new GeoException("Latitude and longitude cannot be null");
        }
        if (latitude < MIN_LATITUDE || latitude > MAX_LATITUDE) {
            throw new GeoException("Latitude out of bounds");
        }
        if (longitude < MIN_LONGITUDE || longitude > MAX_LONGITUDE) {
            throw new GeoException("Longitude out of bounds");
        }
    }
}
```

Лістинг А5 – GeoService.java

```
package com.tairova.tarik.service;
```

```
import static com.tairova.tarik.config.Constants.Geo.BITS_PER_LEVEL;
import static com.tairova.tarik.config.Constants.Geo.DB_QUADKEY_LEVEL;
import static com.tairova.tarik.config.Constants.Geo.MAX_LATITUDE;
import static com.tairova.tarik.config.Constants.Geo.MAX_LONGITUDE;
import static com.tairova.tarik.config.Constants.Geo.MIN_LATITUDE;
import static com.tairova.tarik.config.Constants.Geo.MIN_LONGITUDE;

import com.tairova.tarik.dto.ClusterDto;
import com.tairova.tarik.dto.MapItem;
import com.tairova.tarik.dto.MapSearchRequest;
import com.tairova.tarik.dto.PointDto;
import com.tairova.tarik.exception.GeoException;
import com.tairova.tarik.persistence.repository.GeoRepository;
import jakarta.transaction.Transactional;
import java.util.ArrayList;
import java.util.List;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@Transactional(Transactional.TxType.SUPPORTS)
@RequiredArgsConstructor
public class GeoService {
    private static final int    MAX_POINTS_IN_RESPONSE = 150;
    private static final int    MAX_SINGLETONS        = 50;
    private static final long   MAX_ALLOWED           = 100_000L;
    private static final double KM_PER_DEGREE_LAT     = 111.32;
    private static final double KM_PER_DEGREE_LON     = 111.32;
    private static final double DENSITY_A             = 0.3748971193565054;
```

```
private static final double DENSITY_B            = 0.31050672552442965;

private final GeoRepository geoRepo;

public List<MapItem> findByArea(MapSearchRequest req) {
   validateSearchArea(req);
   validateMaxPoints(req);

   int level = estimateLevelByGeometry(
         req.getMinLat(), req.getMaxLat(),
         req.getMinLon(), req.getMaxLon()
   );
   int shift = (DB_QUADKEY_LEVEL - level) * BITS_PER_LEVEL;

   var aggs = geoRepo.aggregateByCluster(
         req.getMinLat(), req.getMaxLat(),
         req.getMinLon(), req.getMaxLon(),
         shift
   );

   List<MapItem> out       = new ArrayList<>(aggs.size());
   List<Long>    singleCls = new ArrayList<>();
   for (var c : aggs) {
      if (c.getCnt() == 1) {
         singleCls.add(c.getCluster());
      } else {
         out.add(new ClusterDto(c.getLat(), c.getLon(), c.getCnt()));
      }
   }
```

```
    List<PointDto> pts = geoRepo.findSinglesByClusters(
            req.getMinLat(), req.getMaxLat(),
            req.getMinLon(), req.getMaxLon(),
            shift, singleCls, MAX_SINGLETONS
    );
    out.addAll(pts);
    return out;
}

private void validateSearchArea(MapSearchRequest req) {
    if (req == null) throw new GeoException("Request cannot be null");
    double minLat = req.getMinLat(), maxLat = req.getMaxLat();
    double minLon = req.getMinLon(), maxLon = req.getMaxLon();
    if (minLat < MIN_LATITUDE || maxLat > MAX_LATITUDE)
        throw new GeoException("Latitude out of bounds");
    if (minLon < MIN_LONGITUDE || maxLon > MAX_LONGITUDE)
        throw new GeoException("Longitude out of bounds");
    if (minLat >= maxLat) throw new GeoException("Invalid latitude range");
    if (minLon >= maxLon) throw new GeoException("Invalid longitude range");
}

private void validateMaxPoints(MapSearchRequest req) {
    double minLat = req.getMinLat(), maxLat = req.getMaxLat();
    double minLon = req.getMinLon(), maxLon = req.getMaxLon();

    double deltaLat = maxLat - minLat;
    double deltaLon = maxLon - minLon;
    double midLat   = (minLat + maxLat) / 2.0;

    double heightKm = deltaLat * KM_PER_DEGREE_LAT;
```

```
        double widthKm  = deltaLon * KM_PER_DEGREE_LON *
Math.cos(Math.toRadians(midLat));
        double areaKm2  = Math.abs(heightKm * widthKm);

        double density  = DENSITY_A + DENSITY_B *
Math.cos(Math.toRadians(midLat));
        double estimate = areaKm2 * density;
        if (estimate > MAX_ALLOWED) {
            throw new GeoException("Area is too big");
        }
    }

    private int estimateLevelByGeometry(
            double minLat, double maxLat,
            double minLon, double maxLon
    ) {
        for (int L = DB_QUADKEY_LEVEL; L >= 0; L--) {
            int cellsLat = (int) Math.ceil((maxLat - minLat) / (180.0 / (1 << L)));
            int cellsLon = (int) Math.ceil((maxLon - minLon) / (360.0 / (1 << L)));
            if ((long) cellsLat * cellsLon <= MAX_POINTS_IN_RESPONSE) return L;
        }
        return 0;
    }
}
```

Лістинг А6 –QuadKey.java

```
package com.tairova.tarik.service;

public class QuadKey {
```

```
/**
 * Encodes coordinates into a 2-bit quadkey of specified depth
 *
 * @param lat   latitude in degrees
 * @param lon   longitude in degrees
 * @param level depth (typically 23)
 * @return quadkey as long
 */
public static long encode(double lat, double lon, int level) {
    long key = 0;
    double minLat = -90, maxLat = 90;
    double minLon = -180, maxLon = 180;

    for (int i = 0; i < level; i++) {
        int bit = 0;
        double midLat = (minLat + maxLat) / 2;
        double midLon = (minLon + maxLon) / 2;

        if (lat >= midLat) {
            bit |= 2;
            minLat = midLat;
        } else {
            maxLat = midLat;
        }

        if (lon >= midLon) {
            bit |= 1;
            minLon = midLon;
        } else {
```

```
                maxLon = midLon;
            }

            key = (key << 2) | bit;
        }

        return key;
    }
}
```

Лістинг А7 – BssidService.java

```
package com.tairova.tarik.service;

import com.tairova.tarik.config.Constants.Bssid;
import com.tairova.tarik.exception.BssidException;
import java.util.List;
import org.springframework.stereotype.Service;

@Service
public class BssidService {
    private static final String BSSID_PATTERN = "^([0-9A-Fa-f]{2}:){5}[0-9A-Fa-
f]{2}$";

    /**
     * Validates BSSID format
     * @param bssid MAC address to validate
     * @throws BssidException if BSSID is invalid
     */
    public void validateBssid(String bssid) {
```

```
        if (bssid == null || bssid.isEmpty()) {
            throw new BssidException("BSSID cannot be empty");
        }

        if (!bssid.matches(BSSID_PATTERN)) {
            throw new BssidException("Invalid BSSID format. Expected format:
XX:XX:XX:XX:XX:XX");
        }
    }

    /**
     * Converts BSSID from string format to numeric representation
     * @param bssid MAC address in XX:XX:XX:XX:XX:XX format
     * @return numeric representation of BSSID
     * @throws BssidException if BSSID is invalid
     */
    public Long toLong(String bssid) {
        validateBssid(bssid);
        String replaced = bssid.replace(Bssid.SEPARATOR, "");
        try {
            return Long.parseLong(replaced, 16);
        } catch (NumberFormatException e) {
            throw new BssidException("Error converting BSSID to number", e);
        }
    }

    /**
     * Converts numeric BSSID representation to string format
     * @param bssidLong numeric representation of BSSID
     * @return MAC address in XX:XX:XX:XX:XX:XX format
```

```
     */
    public String toString(Long bssidLong) {
        if (bssidLong == null) {
            throw new BssidException("Numeric BSSID representation cannot be null");
        }
        String hexString = Long.toHexString(bssidLong);
        // Pad with zeros to 12 characters (6 octets of 2 characters each)
        hexString = String.format("%012x", bssidLong);
        return hexString.replaceAll(Bssid.FORMAT_PATTERN, "$1" +
Bssid.SEPARATOR);
    }

    /**
     * Converts list of BSSIDs to their numeric representations
     * @param bssids list of MAC addresses
     * @return list of numeric BSSID representations
     * @throws BssidException if any BSSID is invalid
     */
    public List<Long> toLongList(List<String> bssids) {
        if (bssids == null || bssids.isEmpty()) {
            throw new BssidException("BSSID list cannot be empty");
        }
        return bssids.stream()
                .map(this::toLong)
                .toList();
    }
}
```

Лістинг А8 – Geo.java

```
package com.tairova.tarik.persistence.entity;

import static com.tairova.tarik.config.Constants.Geo.DB_QUADKEY_LEVEL;

import com.tairova.tarik.service.QuadKey;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;
import jakarta.persistence.PrePersist;
import jakarta.persistence.PreUpdate;
import jakarta.persistence.Table;
import java.util.List;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@Entity
@Table(name = "geo")
public class Geo {
    @Id
    @Column(name = "BSSID", nullable = false)
    private Long id;

    @Column(name = "latitude")
    private Float latitude;

    @Column(name = "longitude")
    private Float longitude;
```

```
    @Column(name = "quadkey")
    private Long quadkey;

    @OneToMany(mappedBy = "geo")
    private List<Net> nets;

    @PrePersist
    @PreUpdate
    /**
     * Recalculate quadkey on insert and update
     */
    private void updateQuadkey() {
        this.quadkey = QuadKey.encode(
                latitude.doubleValue(),
                longitude.doubleValue(),
                DB_QUADKEY_LEVEL
        );
    }
}
```

Лістинг А9 – Net.java

```
package com.tairova.tarik.persistence.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
import jakarta.persistence.JoinColumn;
import jakarta.persistence.Lob;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import java.time.Instant;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@Entity
@Table(name = "nets")
public class Net {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Integer id;

    @Column(name = "time")
    private Instant time;

    @Column(name = "cmtid")
    private Integer cmtid;

    @Column(name = "IP")
    private Integer ip;

    @Column(name = "Port")
    private Short port;
```

```
@Lob
@Column(name = "Authorization")
private String authorization;

@Lob
@Column(name = "name")
private String name;

@Column(name = "RadioOff", nullable = false)
private Boolean radioOff = false;

@Column(name = "Hidden", nullable = false)
private Boolean hidden = false;

@Column(name = "NoBSSID", nullable = false)
private Byte noBSSID;

@ManyToOne
@JoinColumn(name = "BSSID")
private Geo geo;

@Column(name = "ESSID", length = 32)
private String essid;

@Column(name = "Security")
private Integer security;

@Column(name = "\"NoWiFiKey\"", nullable = false)
private Byte noWiFiKey;
```

```
@Column(name = "WiFiKey", nullable = false, length = 64)
private String wiFiKey;

@Column(name = "NoWPS", nullable = false)
private Boolean noWPS = false;

@Column(name = "WPSPIN", nullable = false)
private Integer wpspin;

@Column(name = "LANIP")
private Integer lanip;

@Column(name = "LANMask")
private Integer lANMask;

@Column(name = "WANIP")
private Integer wanip;

@Column(name = "WANMask")
private Integer wANMask;

@Column(name = "WANGateway")
private Integer wANGateway;

@Column(name = "DNS1")
private Integer dns1;

@Column(name = "DNS2")
private Integer dns2;
```

```
    @Column(name = "DNS3")
    private Integer dns3;

}
```