КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп’ютерних наук та кібернетики
Кафедра теорії та технології програмування

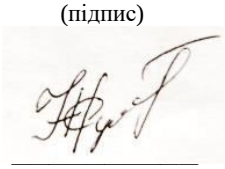
Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

РОЗРОБКА ІНТЕРАКТИВНОЇ БАЗИ ТОЧОК ГЕОКООРДИНАТ. БЕКЕНД ЧАСТИНА

Виконав студент 4-го курсу
Сирота Сергій Вячеславович

Науковий керівник:
доцент, канд.пед.н.
Русіна Наталія Геннадіївна

(підпис)



(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент 

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«____» _____ 2023 р., протокол № ____

Завідувач кафедри
Микола НІКІТІЧЕНКО
_____

Київ – 2023

РЕФЕРАТ

Обсяг роботи: 39 сторінок, 10 ілюстрацій, 1 додаток, 38 використаних джерел.

АВТОМАТИЗАЦІЯ ОБЛІКУ ГЕОПОЗИЦІЙ, ВЕБ-ЗАСТОСУНОК, РОЗГОРТАННЯ СИСТЕМИ КОНТЕЙНЕРІВ, РЕВЕРС ГЕОКОДИНГ, РОЗГОРТАННЯ АВТОНОМНОГО TILE СЕРВЕРУ, ЕКСПОРТУВАННЯ ТА ІМПОРТ ДОКЕР ТОМІВ, ПОСТУПОВА ІНТЕГРАЦІЯ, ПАГІНАЦІЯ КУРСОРОМ.

Об’єкт дослідження – процес обліку геопозицій.

Предмет роботи - бекенд частини автономної системи для інтерактивної роботи з геопозиціями.

Метою кваліфікаційної роботи є проектування, розробка, розгортання та підтримка бекенд частини автономної системи для інтерактивної роботи з геопозиціями та підтримкою інтеграції з існуючими системами обліку.

Інструменти розроблення: docker для контейнеризації, WSL для підтримки docker на Windows, docker-compose плагін для налаштування розгортання та встановлення зв’язків з контейнерів цілої системи, Golang, C#, bash скрипти, GitHub actions, Nominatim, OpenStreetMap Tile Server, Goose, Plantuml.

Результат роботи: виконано загальний огляд систем, призначених для автоматизації обліку геопозицій, розроблено та проверифіковано в умовах розгортання на сервері та автономності власний продукт.

Створені інструменти для автономного розгортання усієї системи, у тому числі tile серверу та реверс геокодингу для окремих частин світу було розроблено в ході виконання кваліфікаційної роботи.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API – Application Programming Interface, прикладний програмний інтерфейс

CI/CD – Continuous integration/continuous delivery, безперервна інтеграція/безперервна доставка

DevOps – Development & operations, розробка й операції

Env змінні – Environment variable, змінна оточення

NATO – North Atlantic Treaty Organization, міжурядовий військовий альянс

WGS – World Geodetic System, всесвітня геодезична система

WSL – Windows subsystem for Linux, Підсистема Windows для Linux

БПЛА – Безпілотний літальний апарат

БД – База даних

ГІС – Геоінформаційна система

Гб – Гігабайт

ПЗ – Програмне забезпечення

ВСТУП

Оцінка сучасного стану об’єкта розробки. На сьогоднішній день існують комплексні системи управління військами, з комплексним функціоналом та сервісами, наприклад, система Delta [1], яка використовується Силами безпеки та оборони України. Delta добре інтегрується з системами NATO та дає змогу працювати виключно онлайн. Артилерійські частини також користуються системою ГІС «АРТА»[2], з причини терміновості отримання інформації про результати бойових дій. Combat vision [3]– надає можливості розподіленої realtime системи, інтегрованої з пристроями з локальної мережі.

Актуальність роботи та підстави для її виконання. Беручи до уваги існування та розробку згаданих систем, вони, як правило мають достатньо вузько-направлене призначення, яке враховує специфіку конкретних задач, і відтак, навіть маючи широкий функціонал, потенційно складніше перевикористовуються для більш загальних потреб. Тому, було вирішено створити систему, яка дала би змогу працювати як онлайн, так і офлайн, а також забезпечуватиме гнучкість у визначенні категорій облікових об’єктів. Також, беручи до уваги сучасну стрімку цифровізацію України, важливо мати багато інструментів загального призначення.

Мета та завдання роботи. Метою роботи є створення бекенд частини системи для обліку геопозицій, а також створення конфігурації та файлів для автономного розгортання.

Для досягнення мети були поставлені завдання:

- здійснити огляд існуючих на ринку систем;
- встановити вимоги та розробити систему обліку геопозицій;
- проаналізувати та розробити автономні модулі для роботи з картами;
- окреслити шляхи для покращення системи.

Об’єкт, методи та засоби розроблення. Об’єктом дослідження є процес обліку геопозицій. Методи розроблення: визначення та оцінка вимог, бекенд-розробка, проектування баз даних та компонентів взаємодії, розробка хмарних додатків, технологій поступово. Розробка продукту базувалась на гнучкій методології AGILE [22] з етапами планування, невеликих релізів та приймальним тестуванням. Інструменти для розробки: docker для контейнеризації, WSL для підтримки docker на Windows [23]; docker-compose плагін для налаштування розгортання та встановлення зв’язків з контейнерів цілої системи; Golang як основна мова для реалізації бекенду розроблюваної системи [24]; C# як допоміжна [25]; bash скрипти для автоматизації задач розгортання та запуску процесів обробки даних [26]; GitHub actions – як платформа для реалізації поступової інтеграції [27]; Nominatim – як основа для вирішення реверс геокодингу [20]; OpenStreetMap Tile Server – набір інструментів для створення та запуску автономного tile серверу [19]. Goose – для міграцій бази даних [18], Plantuml – для генерації UML діаграм з тексту[17].

Можливі сфери застосування. Кінцевий продукт може використовуватися для практичних цілей ведення обліку замінованих та забруднених, а також територій, на яких знаходиться розбита техніка, зруйновані, потенційно небезпечні будівлі для подальшого відновлення в Україні, з причини повномасштабного вторгнення росії. Окрім військової тематики, систему можливо використовувати в аграрному секторі для обліку засіяних територій, можливо застосування гібридного варіанту, де одночасно ведеться облік замінованих та засіяних територій, для розуміння, коли території стають придатними до засівання. Усі ці процеси мають на меті збільшити ефективність бізнесу та зберегти життя людей.

Зв’язок з іншими роботами. У даній роботі використано розроблену в ході виконання спільного проєктного завдання для студентів другого та

четвертого років навчання, поєднаного з практичних частин дисциплін «Методи специфікації програм», «Коректність програм та логіки програмування» та «Інструментальні середовища та технології програмування» систему обліку з веб-сайтом.

РОЗДІЛ 1. ОГЛЯД РИНКУ ІСНУЮЧИХ СИСТЕМ

1.1 Delta

Delta – національна система ситуаційної обізнаності, яку використовують Сили безпеки і оборони України, спроектована за стандартами NATO, що потенційно дає можливість вести мережево-центричну війну [1].

Мережево-центрична війна (Network centric warfare) – воєнна доктрина, орієнтована на забезпечення інформаційної переваги шляхом обладнання техніки та об’єднання військових об’єктів у інформаційну мережу. Система Delta дозволяє в режимі реального часу відстежувати положення військ противника та планувати операцій бойових дій та приймати рішення, щодо подальшого вогневого ураження [8]. Інформація надходить в систему шляхом перетворення даних з різних джерел-постачальників:

1) Радари
2) Супутникові знімки
3) БПЛА
4) Сенсори
5) GPS трекери
6) Радіоперехоплення

Система складається з захищеного месенджеру та великої кількості сервісів, які забезпечують координацію підрозділів. Важливо зазначити, що оскільки система спроектована за стандартом НАТО і підтримує специфікацію Multilateral Interoperability Programme, то уся західна техніка, яка надається західними партнерами інтегрується в Delta майже автоматично. Також інтегровані чат-боти «єВорог» [28] та «STOP Russian

War» [21], що дає змогу отримувати та обробляти інформацію від звичайних громадян [9]. Внаслідок побудованої інфраструктури, рішення базується на використанні хмарних технологій, що знімає відповідальність менеджменту серверами, мережевим обладнанням та іншими ресурсами з команди розробки.

Очевидно, враховуючи усю специфіку, з якою розроблялась і проектувалась система, усі деталі реалізації та окремі сервіси знаходяться під суворими механізмами авторизації та не можуть бути використані у сфері, окремій від військової, наприклад, в аграрному секторі. Проте, така система потенційно може фільтрувати і видавати окремі дані у стандартизованому форматі, щоб забезпечити володіння інформацією застосунку, який використовується в аграрному секторі.

Система оснащена автоматизованим моніторингом спроб зламу та підозрілої активності, реалізована рольова авторизація та двух факторна автентифікація. Структурами кібербезпеки здійснюється постійна перевірка системи на вразливості, витоки даних та спроби несанкціонованого проникнення [10].

Як вказано на офіційні сторінці продукту Delta – система працює з мережевими технологіями [10], що виправдано розмаїттям функціоналу та засобів інтеграції та збору інформації, проте режим офлайн роботи недоступний.

1.2 Система ГІС «АРТА»

ГІС «АРТА» - автоматизована система управління військами, з 2014 року використовується Збройними Силами України та продемонструвала високу ефективність в порівнянні з традиційними підходами до

управління та контролю [2]. Система добре зарекомендувала себе як інструмент планування, контролю, обробки та поширення результатів бойових та розвідувальних операцій. Застосунок має доволі швидкий час наведення на ціль(одна хвилина) та невибагливий до спеціалізованих пристроїв(може використовуватись на смартфоні), проте інформація про автономність, зручність у використанні та інтегрованість з іншими системами не представлена у відкритих джерелах.

1.3 Combat vision

Combat vision 4.0 – розподілена система розвідки та координації на полі бою [3]. Це real-time застосунок, який працює в інтернет мережі з розподіленою базою даних, яка зберігається на кожному пристрої окремо та стандартизована за NATO STANAG 4677, що конкретно спроектовано для умов з нестабільною та повільною комунікацією. Також реалізований окремий модуль візуального контролю з нанесенням на карту усієї або відфільтрованої інформації з поступовим оновленням відображення. Наразі у відкритому доступі відсутня інформація, чи може така система працювати виключно офлайн та накопичувати та аналізувати дані згодом. Застосунок може бути використаний у цивільній сфері підрозділу ДСНС та Поліції для швидкого аналізу обстановки та реагування у разі виникнення надзвичайних ситуацій, дорожніх подій. Ще одним варіантом використання є спостереження за ділянками кордону за допомогою комплексу спостереження, який включає в себе БПЛА, камери наземного спостереження, патрульні літаки, тощо.

Проведений огляд наявних на ринку систем дозволяє констатувати, про недостатню кількість незалежних від сфери застосунків щодо функціоналу одночасного накопичення даних, можливості робити звіти,

інтегрування з іншими системами за загально прийнятим стандартом та роботи з картами, не втрачаючи функціоналу, знаходячись поза Інтернет мережею.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ

2.1 Огляд функціональних вимог

Існує декілька форматів вимог до ПЗ: системні, функціональні та нефункціональні вимоги.

Функціональні вимоги – це перелік функцій або сервісів, які повинна надавати система, а також обмежень на дані і поводження системи при їхньому виконанні. Специфікація функціональних вимог (software requirements specification) – опис функцій та їхніх властивостей, які не містять у собі протиріч і виключень [6].

Розроблювана система має задовольняти наступні вимоги:

Робота з даними та аналітика:

1) Уведення даних по одній точці – вручну:

a. Можливість вводити три типи координат: wgs84 – сучасні, широко розповсюджені [5], sk42 – радянські та застарілі, mgrs – з військової ніші. Автоматичне переведення з першої введеної системи у інші.

b. Населений пункт, район та область заповнюються автоматично з уведених координат

c. Дата та час виявлення

d. Опис позиції

e. Категорія об’єкту – може виводитись автоматично з короткого опису, або обиратись вручну з переліку

f. Код значка app6d – автоматично виводиться з обраної категорії

g. Посилання на файл джерело

h. Коментар, синтезований з уведених даних

i. Ідентифікатор ворожості

j. Джерело інформації

k. Категорія надійності інформації

2) Комплексна фільтрація:

a. За певним радіусом від координати

b. За попередньо визначеними категоріями

c. За текстом, з толерантністю до друкарських помилок

d. За часом виявлення

e. За активністю у певний період часу

3) Забезпечення ефективної пагінації

4) Перегляд даних на інтерактивній мапі з можливістю архівування окремої позиції

5) Ефективні запити, мінімізація передавання зайвої інформації

6) Можливість змінювати вже існуючі дані

7) Автоматичне архівування «застарілих» даних

Базова авторизація

Інтеграція з Excel:

1) Імпортування набору даних з Excel:

a. Заповнення пропущених рядків

b. Повідомлення помилок валідації

c. Перевірка на унікальність

d. Візуальний контроль драфт таблиці

e. Можливість вносити зміни в драфт таблицю одразу в застосунку

2) Можливість змінити назви колонок

3) Формування звітів профільтрованих даних в форматі Excel

Для полегшення розуміння, функціональні вимоги відобразимо Use Case діаграмою (рисунок 2.1).

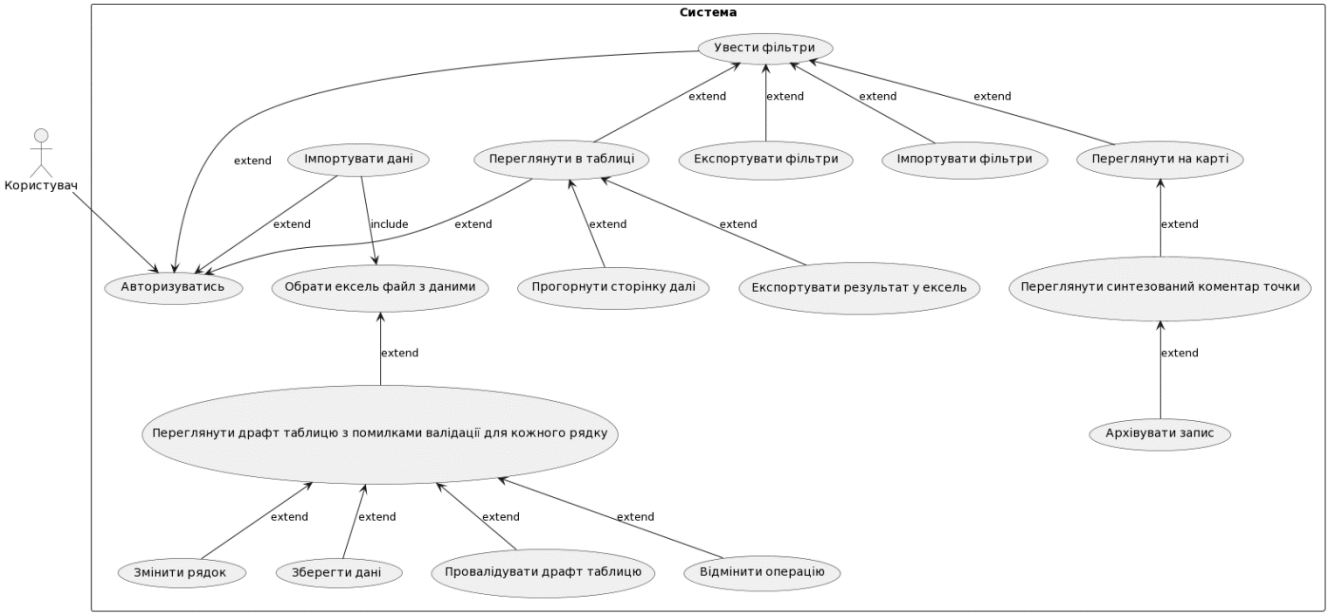


Рисунок 2.1 – Use Case діаграма системи

Враховуючи аспект командної розробки проекту, варто зазначити власну роль у реалізації функціональних вимог, це насамперед реалізація серверної частини проекту, а саме: робота з даними та аналітика (окрім автоматичного виведення систем координат та категорій), базова авторизація та робота з імпортованими драфт таблицями та їх збереженням.

2.2 Огляд нефункціональних вимог

Нефункціональні вимоги – визначають умови виконання операцій у середовищі, що безпосередньо не пов’язані з функціями, проте відображають потреби користувачів щодо їх виконання [6].

Такі умови для застосунку:

1) швидкодія фільтрації – до 1 секунди при 1Гб даних;
2) швидкодія запису – до 1 секунди для 50 тисяч позицій;
3) надійність виконання – система має бути толерантною до відмови сервісу реверс геокодингу або інтерактивних мап;

4) важливим аспектом є зручність інтерфейсу та документування API бекенду.

2.3 Огляд системних вимог

Системні вимоги – визначають зовнішні умови виконання системних функцій і обмежень на створення продукту, а також вимоги до опису програмно-апаратних підсистем [6].

Мінімальні вимоги для роботи системи:

- операційна система – Windows 10+, MacOS, Linux;
- процесор – 2Гц, 4 фізичні ядра(6, якщо запускається на віртуальні машині);
- дисковий простір – 25Гб + розрахунок на збереження даних у форматі 1 мільйон позицій ~ 2Гб простору;
- браузер – Google Chrome [14], Safari [15], Firefox [16];
- інтернет – у онлайн серверному режимі 10Мб/с, можна працювати офлайн без втрати функціоналу;
- додаткове ПЗ – docker, WSL якщо Windows.

РОЗДІЛ 3. ПРОЕКТУВАННЯ Й РОЗРОБКА

3.1 Архітектура системи

Архітектура – форма системи, надана їй тим, хто її створює. Ця форма розділяє систему на компоненти, визначає розташування цих компонентів та спосіб за яким вони ведуть комунікацію між собою. Первинна ціль архітектури полягає в забезпеченні життєвого циклу системи. Гарна архітектура робить систему простою для розуміння, розробки, підтримки та розгортання. Остаточною ціллю є мінімізація вартості терміну експлуатації та збільшення продуктивності розробників[37]. Для повного розуміння поняття архітектури визначимо детальніше властивості.

Розробка – архітектура повинна забезпечувати можливість командам організовуватись за власним бажанням, не заважаючи один одному. Якісне горизонтальне розщеплення системи, в якій бізнес правила не залежать від коду візуального відображення дозволяє різним командам(розробники бізнес правил та відображення) працювати окремо, не впливаючи один на одного. Також існує аспект вертикального розщеплення, такий тип дозволяє розділити команди розробки за юз кейсами системами, наприклад, команда, працююча над додаванням замовлень ніколи не має заважати команді, яка працює над видаленням замовлень[22].

Розгортання – простота розгортання всієї системи однією командою в терміналі.

Підтримка – найдорожча частина розробки системи. Головні компоненти вартості підтримки це складність визначення, яку частину системи потрібно змінити щоб додати новий функціонал або виправити баг. Друга частина вартості це підвищення ризику додавання нового багу при додаванні нового

функціоналу, або виправленні старого в заплутаному програмному забезпеченні.

Працюючи в команді, яка складалась з розробників з експертизою в різних сферах та мовах програмування, необхідно було чітко визначити кордони відповідальностей компонентів для забезпечення ефективності розробників.

Загалом проект складається з 6 компонентів, які розгортаються як окремі контейнери та використовують допоміжні бібліотеки (рисунок 3.1) для візуалізації карти (LeafLet), переведення координатних (GDAL) систем та полегшення розробки візуальних інтерфейсів - React.

1) Frontend – візуальне представлення в браузері
2) Core – серверний застосунок, в якому зосереджена основна частина бізнес логіки
3) Excel adapter – адаптер для формування звітів у ексель форматі
4) MySql – як база даних
5) TileServer – для автономного візуального представлення мапи
6) Nominatim – як сервіс для використання можливостей реверс геокодингу.

Термін геокодинг означає трансляцію людино-читабельної адреси в локацію на мапі. Процес реверерс геокодингу навпаки, переводить локацію на мапі в людино-читабельну адресу [38].

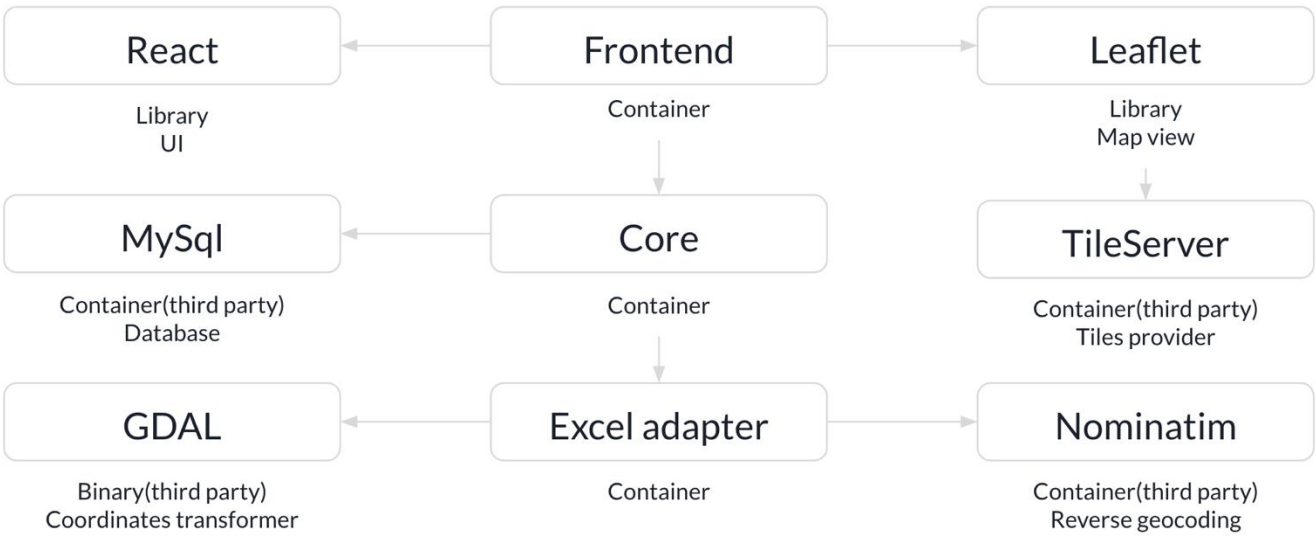


Рисунок 3.1 – Діаграма компонентів системи та зв’язків між ними

3.2 Моделювання даних

Розглянемо аспекти збереження даних, почнемо з масштабування. CAP теорема дає чіткий список трьох властивостей бази даних при розподіленому збереженні – узгодженість, доступність та стійкість до розподілення. Стверджується [12], що одночасно неможливо забезпечити виконання більше двох із перелічених властивостей, де:

- узгодженість – стан даних є одночасно узгоджений між усіма вузлами в системі;
- доступність – гарантія отримання відповіді кожному вхідному запиту;
- стійкість до розподілення – система здатна працювати при розподіленні та втраті зв’язку з окремими вузлами.

Максимальний очікуваний об’єм даних в системі - декілька десятків Гб, має забезпечуватись висока доступність та узгодженість, не має вимоги працювати в розподіленому режимі. В такому випадку доцільно використати базу даних з підтримкою узгодженості та доступності, оскільки структура даних є визначеною, хоч і потенційно розширюваною. Ідеальним кандидатом є

система управління реляційними базами даних. Вимогами до такої БД є безкоштовна ліцензія, підтримка роботи з географічним координатним типом даних. Проаналізувавши ринок, визначено два кандидати – MySQL [29] та PostgreSQL [30]. Обрано MySQL в силу вбудованої підтримки типів координат, в рішенні PostgreSQL підтримка забезпечується підключенням окремого плагіну, що потенційно збільшує складність системи.

Структура даних складається з головної таблиці – positions, в якій зосереджена більшість полів для зберігання геопозиції. Таблиця складається з декількох головних наборів полів:

1. Локація – до цього типу відносяться поля різних систем координат(lat, lon, height, sk42_x, sk42_y, mgrs.), посилань на населений пункт, область та район - most_close_settlement_id, region_id та district_id відповідно.

2. Тип інформації – час виявлення (detection_time), посилання на джерело (link), категорія надійності інформації (reliability_category) та джерело (source).

3. Тип точки – ідентифікація (identity), наприклад «Ворог» або «Друг», короткий опис позиції (OIVT), категорія зі списку існуючих (category) та позначка на карті(app6d_code). Також представлене поле статусу (status), наприклад, «АРХІВОВАНА» або «АКТИВНА».

4. Утилітарні – коментар від оператора та синтезований коментар по точці – comment та summary_comment відповідно. Унікальний ідентифікатор (id), порядковий номер (o_id), створено та оновлено (created_at, updated_at). Час, коли точка перестала бути активною (active_end_time), використовується як закешоване поле для пришвидшення пошуку типу «узяти всі точки активні в N час). Infinite_lifetime – як позначка, що не слід архівувати поле, навіть якщо його термін життя вичерпано.

Таблиця places має просту структуру, яка відповідає за збереження назви локації та типу локації, наприклад «Черкаси, Населений пункт». Таблиця categories має на меті також зменшити дублювання даних та дозволити робити швидке та консистентне оновлення категорії.

Реляційну модель даних (рисунок 3.2). розроблено з використанням третьої нормальної форми, де кожна таблиця має основний ключ, будь-яке поле залежить лише від первинного ключа та дані не дублюються в рядках [13].

В даному випадку для кожної таблиці первинний ключ це id в форматі VARCHAR(36), що є рядком фіксованої довжини. Сам застосунок використовує поле id для зберігання UUID (Universally unique identifier) [31] четвертої версії, яке генерується випадково з 2^128 комбінацій, що є цілком прийнятним для очікуваної кількості формату даних.

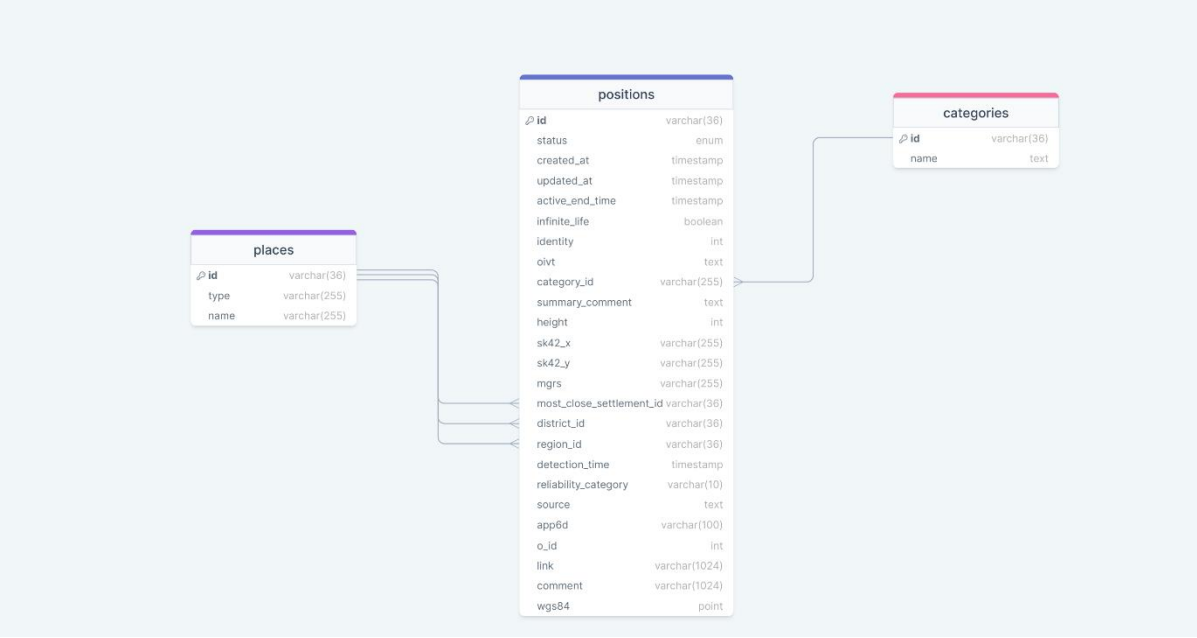


Рисунок 3.2 - UML Entity Relations діаграма

3.3 Імплементація обробки та збереження даних

Для того, щоб система забезпечувала можливість пошуку та фільтрації даних створимо REST [32] сервер. В умовах клієнт серверної взаємодії важливо встановити контракт, за яким клієнт може звертатись до серверу та отримувати

відповідь в очікуваному форматі. Оскільки Golang статично типізована мова, для такої цілі було написано невелику бібліотеку з автогенерацією специфікації OpenAPI [33] за допомогою рефлексії структур, які очікуватимуть обробники запитів. Візуальне представлення згенерованої документації представлено на рисунку 3.3. Варто зазначити, що OpenAPI складається з двух частин: специфікації та візуального відображення, розроблена бібліотека генерує специфікацію.

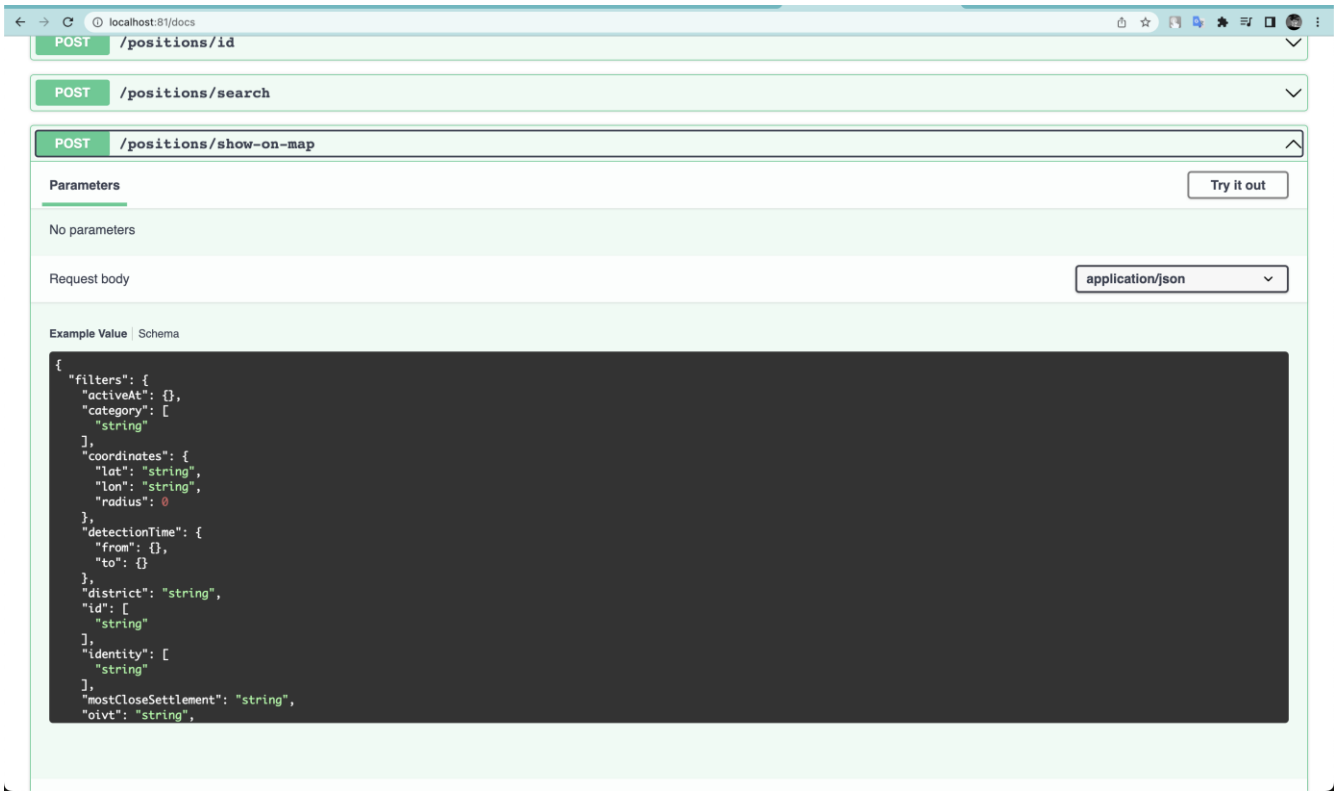


Рисунок 3.3 – Згенерована документація OpenAPI

Маючи підготовлене середовище для ефективної розробки серверу було розроблено перші 4 контролери HTTP запитів для функціоналу фільтрації: для експортування звіту в Excel форматі(1), для відображення даних в таблиці (2), для відображення даних на карті (3,4). Варто зазначити, що останні 2 запити мають різну сигнатуру і працюють по різному. Для відображення даних на карті необхідно дістати ідентифікатор, координати та значок app6d для усієї вибірки

за вказаними фільтрами і потім за потреби діставати детальну інформацію по позиції.

Остаточна сигнатура перших трьох контролерів наступна:

1) excelExport (filters: Filters): ExcelFile, де Filters – тип об’єкту для передачі фільтрів;
2) search (filters: Filters, cursor: string | null, pageSize: number): Array<Position>, де Position – тип об’єкту для представлення позиції;
3) showOnMap (filters: Filters): Array< PartialPosition >, де PartialPosition – тип об’єкту для представлення необхідних для відображення на карті даних позиції;
4) findById (id: UUID): Position | null.

Для відображення в таблиці необхідно дістати всі можливі поля об’єкта, що потенційно негативно впливає на швидкодію, тому для такого пошуку використаємо пагінацію курсором.

Пагінація (від англ. paging – нумерування сторінки) – спосіб завантаження великої кількості даних, шляхом послідовного надсилання запитів на отримання вибірки фіксованої довжини, таким чином забезпечується пришвидшення початкового завантаження та перегляду даних [36].

Пагінація курсором – тип пагінації, яка уникає більшості проблем, які виникають з пагінацією «відступом». Курсор вказує на певне положення або позицію в наборі даних і дозволяє отримувати додаткові результати зі зсувом або попередні результати. В нашому випадку курсор та складається з полів композитного індексу created_at та id. Таким чином фільтрація курсором відбувається за індексом та усі дані сортуються спочатку за created_at, час може повторюватись, цю проблему вирішує вторинне сортування за індексом id (рисунок 3.4).

Оскільки пагінація «відступом» спочатку фільтрує дані, а потім повертає частину, такий підхід доволі неефективний і при зростанні кількості даних у вибірці таблиці запит буде помітно повільнішим, часова складність O(N). Натомість пагінація курсором фільтрує дані спираючись на композитний індекс, отже часова складність складає O(log(N)).

Композитний індекс – індекс, який використовує декілька колонок даних з таблиці, таким чином пришвидшуючи запити, які містять фільтрацію, або сортування з використання колонок з індексу.

```
189  func (repo SqlRepo) Search(ctx context.Context, params *core.SearchParams) (*core.SearchRes, error) {
190    positionsRes := []model.Position{}
191    searchQuery := NewSearchQuery(repo.db.WithContext(ctx), params.Filters)
192    query := repo.db.
193      Debug().
194      Preload("MostCloseSettlement").
195      Preload("District").
196      Preload("Region").
197      Preload("Category").
198      Scopes(searchQuery.Scopes...).
199      Order("created_at ASC").
200      Order("id ASC")
201    if params.Cursor != "" {      You, 2 months ago • implement pagination
202      createdCursor, ID, errCsr := decodeCursor(params.Cursor)
203      if errCsr != nil {
204        return nil, errors.New("неправильного формату курсор, надсилайте порожній або той, який повертається з запиту")
205      }
206      query.Where("created_at>? OR (created_at = ? AND id > ?)", createdCursor, createdCursor, ID)
207    }
208    if err := query.Limit(int(params.PageSize)).Find(&positionsRes).Error; err != nil {
209      return nil, wraperr.Wrap(err, "search query failed")
210    }
211    nextCursor := ""
212    if len(positionsRes) > 0 {
213      nextCursor = encodeCursor(positionsRes[len(positionsRes)-1].CreatedAt, positionsRes[len(positionsRes)-1].ID)
214    }
```

Рисунок 3.4 – Код відповідальний за використання пагінації курсором

Для взаємодії з базою даних в сутності репозиторію інкапсульовано використання бібліотеки об’єктно-реляційного перетворення GORM [35],

Функціонал збереження даних умовно розділено на декілька частин:

1) створення драфт таблиці;
2) перевірка відповідності правилам даних;
3) збереження в БД.

Розглянемо безпосередньо збереження даних в сховище. Сигнатура REST контролеру для розробленого юз-кейсу виглядає наступним чином: uploadData(position: Array<Position>): Error | null. Оскільки система повинна перевіряти дані перед збереженням, у випадку якщо перевірка відповідності

вимогам провалиться – клієнт має отримати помилку з відповідним повідомленням.

Очікуване завантаження даних у об’ємі декількох тисяч записів за один виклик, отже першим кроком до швидкодії системи буде уникання великої кількості надлишкових запитів на встановлення з’єднання з MySQL сервером та затримок запитів в обидві сторони. Цю проблему вирішено шляхом використання завантаження партіями (batch uploading) в 100 одиниць позицій за один SQL запит (рисунок 3.5). Важливим моментом є створення партіями прямих залежностей позиції, в даному випадку неіснуючих категорій та локацій. Алгоритм наступний – спочатку завантажуємо залежності та отримуємо унікальні ідентифікатори записів, перетворюємо позиції на такі, в яких замість значень залежностей, наприклад назв, записані посилання на новостворені записи та зберігаємо позиції по 100 одиниць.

```
func (repo SqlRepo) SaveMany(ctx context.Context, data *core.ValidatedUploadManyParams) error {
  positionsToSave := make([]model.Position, len(data.Positions))      You, 2 months ago • add insert mapping …
  settlementIDs, err := repo.findOrCreatePlaceInBatches(ctx, lo.Map(data.Positions, func(pos core.PositionInput, _
  if err != nil {
    return err
  }
  districtIDs, err := repo.findOrCreatePlaceInBatches(ctx, lo.Map(data.Positions, func(pos core.PositionInput, _ in
  if err != nil {
    return err
  }
  regionIDs, err := repo.findOrCreatePlaceInBatches(ctx, lo.Map(data.Positions, func(pos core.PositionInput, _ int)
  if err != nil {
    return err
  }
  categoryIDs, err := repo.findOrCreateCategoriesInBatches(ctx, lo.Map(data.Positions, func(pos core.PositionInput,
  if err != nil {
    return err
  }
  for i, pos := range data.Positions {
    positionsToSave[i] = mapPositionBaseToDb(pos.PositionBase, categoryIDs, settlementIDs, districtIDs, regionIDs)
  }
  return repo.db.WithContext(ctx).CreateInBatches(positionsToSave, 100).Error
}
```

Рисунок 3.5 – Код відповідальний за завантаження партіями позицій та залежностей – найближчих населених пунктів, регіонів, областей та категорій

Робота з картами в застосунку розділена на три основні частини, це:

1) переведення в різні системи координат;

2) візуальне відображення;
3) реверс геокодинг.

3.4 Робота з картами

З точки зору серверної частини візуального відображення визначимо, що для того, щоб використовуючи на клієнті бібліотеку відображення, побачити перед собою мапу необхідно звертатись до джерела мап.

Tile server – сервер, який надає зображення карти частинами у форматі растрових плиток – квадратними малюнками, у форматі png 256 x 256 пікселів, які відображаються у вигляді сітки і формують мапу [19], як реалізацію такого застосунку використано OpenStreetTile Server.

Для операційної діяльності такому застосунку необхідно мати джерело мапи, таким джерелом в OpenStreetTile Server є база даних PostgreSQL. Проте, враховуючи, що розмір дискового простору, необхідного для відображення є доволі великим, для відображення топографічної карти України необхідно 23.6 Гб дискового простору, розробники розробили протокол pbf (Protocolbuffer Binary Format) – який дозволяє стиснути мапу України до розміру в 1.5 ГБ. Значне стиснення досягається шляхом нормалізації даних та видалення індексів, згенерованих для пришвидшення пошуку та генерації готових зображень.

Щоб згенерувати файли для забезпечення роботи tile серверу необхідно провести процес імпорту, який займає значний час – для України займає 3 години на системі MacOs Ventura 13.1, CPU: 2.6 GHz 6-core, Memory: 16Гб DDR4.

Оскільки така значна затримка в умовах першого запуску програми на потенційно непродуктивних ноутбуках неприпустима, а можливості просто скопіювати готову базу даних та запустити її – немає, прийнято рішення розробити рішення, яке дозволить один раз зробити імпорт і дозволити

передавати згенеровані файли. Для цього потрібно було виконати наступні кроки:

1) завантажити pbf файл для мапи України;
2) підготувати середовище для запуску контейнерів, використано docker;
3) запустити команду import в openstreet-tile-server контейнері, попередньо створивши віртуальний диск і вказавши, що імпортувати треба саме в директорію, в яку було змонтовано віртуальний диск (рисунок 3.6);
4) по завершенню імпорту створити новий контейнер, змонтувати диск з імпортованими даними та зібрати усі файли в один файл за допомогою архівування без стиснення – tar командою (рисунок 3.7);
5) експортувати запаковані дані до сховища.

Таким чином оптимізована значна частина часу для розгортання системи з нуля, 24ГБ можливо передати носієм дискового простору (флешкою) або завантажити з мережі Інтернет – при швидкості інтернету 10 МБ/с знадобиться близько 40 хвилин.

```
docker run -d -e "OSM2PGSQL_EXTRA_ARGS=-C 4096" -e THREADS=7 --shm-size="2000m" \
-v "/Users/s.syrota/Projects/sandbox/infrastructure/ukraine-latest.osm.pbf:/data/region.osm.pbf" \
-v osm-data-ua:/data/database/ \
overv/openstreetmap-tile-server \
import
```

Рисунок 3.6 – Код відповідальний за імпортування карти України в tile сервер

```
» docker run --rm --volumes-from osm-data-ua-ru-rb-final -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata
```

Рисунок 3.7 – Код відповідальний за упакування імпортованих даних в один файл

Для того, щоб запустити сервер, маючи запаковані експортовані дані, необхідно їх розпакувати в диск (рисунок 3.8) та змонтувати цей диск в контейнер tile серверу при запусканні.

```
14  docker run --entrypoint /bin/bash
15  -e "OSM2PGSQL_EXTRA_ARGS=-C 4096"
16  -e THREADS=7 --shm-size="2000m"
17  -v osm-data-ua:/data/database/ overv/openstreetmap-tile-server:2.1.0
18  -c "apt-get update; apt-get install -qq curl -y; \      You, now • Uncommitted change
19      cd /; curl https://tbs-osm-pbf.s3.eu-west-1.amazonaws.com/ua_tile.tar | tar -xv"
20
```

Рисунок 3.8 – Код відповідальний за завантаження експортованих джерел карти України для tile серверу

Аналогічна ситуація з підсистемою, використаною для потреб реверс геокодингу – Nominatim. Хоча час на імпортування даних з формату pbf в такій системі менший, його можна оптимізувати шляхом передчасного імпорту та підготовки усі необхідних даних для так званого «прогрітого старту».

3.5 Розробка модулю реверс геокодеру

Однією з необхідних функціональних вимог системи є виведення найближчого населеного пункту, району та області на основі заданих координат, такий процес більш загально називається реверс геокодингом. Для того, щоб такий функціонал працював необхідно мати джерело даних та програмне забезпечення, яке, проводячи операції над цим джерелом здатне визначити адресу. Офіційна сторінка OpenStreetMap працює з застосунком Nominatim для забезпечення операцій геокодингу. Провівши аналіз, була визначення модель розгортання системи у docker контейнері:

1) Завантажити сирі дані у форматі pbf

2) Запустити контейнер Nominatim з командою “import”, вказавши при цьому шлях до сирих даних та змонтувати том, в який будуть записані оброблені дані
3) Після успішного завершення імпорту запустити контейнер з командою “run”, змонтувавши до нього том, використаний для імпорту

Кроки подібні до запуску tile серверу, що однозначно є плюсом з причини полегшення когнітивного навантаження при розгортанні системи.

Для роботи з геокодингом в Nominatim представлено два REST методи:

- Search – призначений як для геокодингу, так і для реверс геокодингу, має гнучки інтерфейс і може повертати результат, коли “Reverse” метод не повертає.
- Reverse – призначений виключно для реверс геокодингу, може повертати дані в різному форматі. Інтерфейс представлений у вигляді GET запитів за посиланням:

“http://nominatim.org/reverse?lat=<value>&lon=<value>&params”

Варто зазначити, що система здатна працювати з різними мовами, для того щоб використовувати саме українську мову, необхідно передати в параметрах посилання “accept-language=uk”.

В розроблюваній системі є необхідність працювати з реверс геокодингом у форматі: “reverseGeocode(lat, lon: float64): { mostCloseSettlement, region, district }“ та віддавати результат на будь-який запит, проте сервіс Nominatim не завжди повертає у відповіді необхідну нам інформацію, було помічено, що при запитах до сервісу з поступовими невеликими змінами координат дані все ж таки надходили.

Відповідно, прийнято рішення реалізувати алгоритм, який використовуючи Nominatim, буде здатен завжди повертати необхідну інформацію, при цьому враховуючи аспект продуктивності самого сервісу в

обмежених ресурсах. Для цього імплементацію розбито на дві частини: забезпечення ефективності запитів до сервісу геокодингу, реалізація надбудови над геокодером.

Для алгоритми надбудови над геокодером розроблено наступні кроки:

1) З початковими вхідними координатам зробити одночасно запити на search і reverse до Nominatim
2) Якщо необхідна інформація отримана – завершити виконання і повернути результат
3) Виконати пошук по колу – здвинути початкові координати за 8 напрямками – 0, 45, 90, 135, 180, 225, 270 та 315 градусів на «500 + (кількість ітерацій - 1)*3» метрів та виконати для кожної нової координати запити в search і reverse, перейти до кроку 2.

За допомогою такого алгоритму надбудови досягається більша стійскість до випадків, коли точно за координатою неможливо визначити адресу та балансується швидкодія системи геокодингу, приклад використання розробленого методу зображений на рисунку 3.9.

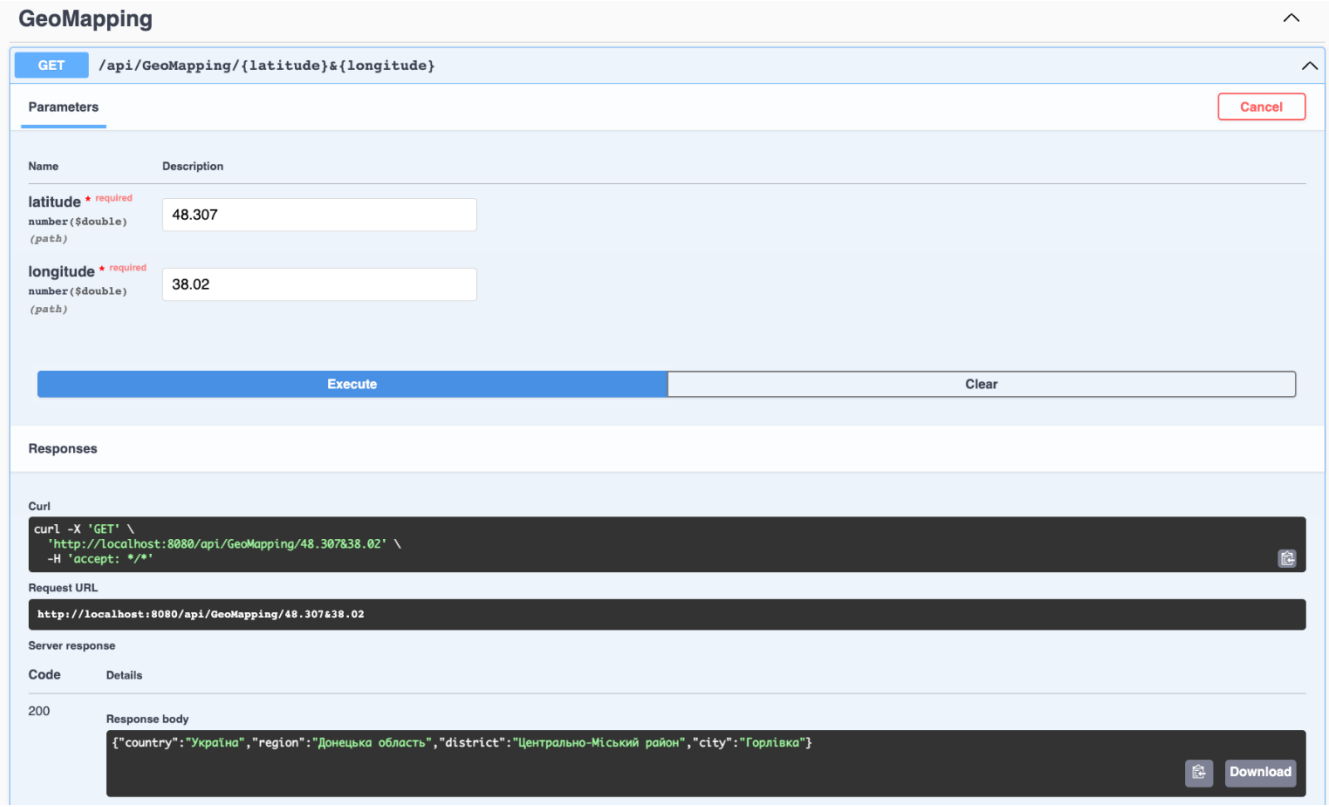


Рисунок 3.9 – Використання розробленої надбудови реверс геокодингу на прикладі міста Горлівка, Донецької області

Запущений контейнер Nominatim з увімкненим обмеженням ресурсів пам’яті 4Гб та процесорного часу 4 CPU оптимальна максимальна кількість паралельних запитів сягає 40 одиниць, при такій кількості в середньому запит повертається за 300 мілісекунд, при збільшенні кількості паралельних запитів затримка значно зростає. Щоб виміряти продуктивність в такому випадку треба було визначити середню кількість оброблених успішних запитів за секунду протягом хвилини активного бомбардування запитами при різній кількості паралельних запитів. Отже, для того щоб ефективно користуватись сервісом Nominatim при заданій конфігурації, необхідно обмежити кількість паралельних запитів, наприклад, використовуючи лічильний семафор.

Оскільки в розроблюваній системі передбачаються випадки, коли в короткий проміжок часу потрібно працювати з координатами, які знаходяться

близько, для покращення швидкодії додано кешування відповідей сервісу у розмірі 1000 записів.

В результаті отримано швидкодію алгоритму надбудови на рівні 10 паралельних запитів з затримкою 500 мілісекунд, що є гарним результатом, враховуючи досить обмежені ресурси, зону покриття геокодингом усієї України та стійкість до знаходження необхідної інформації, навіть, якщо точка знаходиться за населеним пунктом.

ВИСНОВКИ

Цифровізація в Україні стає популярнішою з кожним роком і кількість сфер, які потребуються застосування її методів збільшується. Саме тому швидко розвивається цифровізація обліку геокоординат та може бути впроваджена будь-де: у оборонних потребах, у аграрному секторі в логістиці і так далі.

В згаданих вище сферах є декілька аспектів вимог до систем обліку, безпекові та щодо швидкодії системи, інтегрованості з іншими системами, можливості переглядати дані у реальному часі, розгортання офлайн і масштабованості у серверному режимі.

Відповідно до поставленої мети, були виконані наступні завдання:

- здійснено огляд існуючих на ринку систем;
- встановлено вимоги та розроблену систему обліку геопозицій;
- проаналізовано та розроблено автономні модулі для роботи з картами;
- окреслено шляхи для покращення системи.

В даній роботі було спроектовано та розроблено серверну частину інтерактивної бази точок геокоординат, а саме забезпечення роботи офлайн карт, імплементація надбудови над реверс геокодером та реалізація бізнес-логіки роботи з даними в серверній частині. Процес розгортання офлайн полягає в тому, щоб мати можливість запустити систему на єдиному пристрої, не маючи доступ до інтернету та не втратити функціонал.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Delta – національна військова система ситуаційної обізнаності [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://gur.gov.ua/content/viiskovi-rozvidnyky-opanovuiut-systemu-delta.html
2. ГІС «АРТА» - автоматизована система управління військами [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://gisarta.org/uk/index.html
3. Combat Vision – система розвідки та координації на полі бою [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://combat.vision/
4. Російське вторгнення в Україну [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Russian_invasion_of_Ukraine
5. WGS84 – World Geodetic System 1984 Reference System [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://earth-info.nga.mil/index.php?dir=wgs84&action=wgs84#:~:text=WGS%2084%20is%20the%20standard,Terrestrial%20Reference%20System%20(ITRS).
6. Supplement 4-A, A procedure for requirement analysis [Електронний ресурс– Режим доступу до ресурсу: https://web.archive.org/web/20170131231503/http://www.dau.mil/publications/publicationsdocs/sefguide%2001-01.pdf
7. А.Д Тевяшев, В.П. Ткаченко, М.І. Губа та ін. Геоінформаційні системи. Вступний курс. – Харків. 2017. С. 392
8. Мережево центрична війна [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу: https://mil.in.ua/uk/blogs/136712/
9. Військовий софт Delta тепер офіційно у ЗСУ. [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу: https://forbes.ua/innovations/twitter-dlya-zsu-viyskoviy-soft-delta-dopomagav-u-vsikh-velikikh-operatsiyakh-vid-potoplennya-moskvi-do-zvilnennya-zmiinogo-chomu-z-nim-zsu-voyuyut-shvidshe-07122022-10318
10. Відкрита документація Delta. [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу: https://delta.mil.gov.ua/wiki/info/
11. Як працює система управління військами Delta [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://mil.in.ua/uk/articles/yak-pratsyuye-systema-upravlinnya-vijskamy-delta-interv-yu-zi-spivzasnovnykom-go-aerorozvidka-yaroslavom-goncharom/
12. CAP theorem [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу http://blog.thislongrun.com/2015/03/the-confusing-cap-and-acid-wording.html

13. Normal forms [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу http://www.bkent.net/Doc/simple5.htm
14. Google chrome [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.google.com/chrome/
15. Safary [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.apple.com/safari/
16. Firefox [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.mozilla.org/en-US/firefox/new/
17. Plantuml [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://plantuml.com/
18. Goose [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://github.com/pressly/goose
19. Openstreetmap tile server [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://github.com/Overv/openstreetmap-tile-server
20. Nominatim [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://nominatim.org/
21. Stop Russian War бот [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://ms.detector.media/trendi/post/29347/2022-04-15-sbu-onovyla-bot-stop-russian-war-mozhna-povidomlyaty-pro-vybukhivku-ta-pokynutu-rosiysku-tekhniku/
22. AGILE [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.atlassian.com/agile
23. Що таке WSL [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу https://learn.microsoft.com/en-us/windows/wsl/about
24. Golang [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://go.dev/
25. C# [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
26. GNU Bash [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.gnu.org/software/bash/
27. GitHub Actions [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://docs.github.com/en/actions
28. єВорог [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://armyinform.com.ua/tag/yevorog/
29. MySQL [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.mysql.com/
30. PostgreSQL [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.postgresql.org/docs/current/index.html
31. UUID [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://datatracker.ietf.org/doc/html/rfc4122
32. REST [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.codecademy.com/article/what-is-rest
33. OpenAPI [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://www.openapis.org/

34. Cursor vs LIMIT OFFSET pagination [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://stackoverflow.com/questions/55744926/offset-pagination-vs-cursor-pagination
35. Gorm [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://gorm.io/
36. Pagination, incremental page loading. Google search docs. [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://developers.google.com/search/docs/specialty/ecommerce/pagination-and-incremental-page-loading
37. Robert C. Martin. Clean Architecture – NJ. 2017. 139C.
38. Reverse geocoding. . [Електронний ресурс]: [Веб-сайт] – Режим доступу до ресурсу https://developers.google.com/maps/documentation/geocoding/requests-reverse-geocoding

ДОДАТКИ

Додаток А. Код надбудови над реверс геокодером

```
using Microsoft.Extensions.Caching.Memory;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using WebApi.Model;

namespace WebApi.Services
{
   public class GeoMappingService
   {
      private string reverse_url;
      private string search_url;
      private MemoryCache cache;
      private SemaphoreSlim semaphore = new SemaphoreSlim(40);

      public GeoMappingService()
      {
         cache = new MemoryCache(new MemoryCacheOptions
         {
            SizeLimit = 1000
         });
      }

      public async Task<(bool, string)> GetNearestPlaceAsync(string reverseGeoUrl,
string searchGeoUrl, double latitude, double longitude)
      {
         reverse_url = reverseGeoUrl +
"?lat={0}&lon={1}&zoom=10&addressdetails=1&format=json";
         search_url = searchGeoUrl + "?q={0},{1}&addressdetails=1&format=json";
         try
         {
            var goAround = await GoAround(latitude, longitude);
            return (true, JsonConvert.SerializeObject(goAround));
         }
         catch (Exception ex)
         {
            return (false, ex.Message);
         }
      }

      public virtual async Task<string> SendRequestAsync(string url)
      {
```

```
        if(cache.TryGetValue(url, out string cachedResponse))
        {
            return cachedResponse;
        }
        using var httpClient = new HttpClient();
        httpClient.DefaultRequestHeaders.Add("accept-language", "uk-UA");
        httpClient.DefaultRequestHeaders.Add("user-agent", "ExcelAdapter");
        using var response = await httpClient.GetAsync(url);
        if(response.IsSuccessStatusCode)
        {
            string content = await response.Content.ReadAsStringAsync();

            var cacheEntryOptions = new
MemoryCacheEntryOptions().SetSize(1).SetSlidingExpiration(TimeSpan.FromHours
(1));
            cache.Set(url, content, cacheEntryOptions);
            return content;
        }
        else
        {
            throw new Exception();
        }
    }

    private async Task<Address> TryGetAddress(string url)
    {
        await semaphore.WaitAsync();
        Address res;
        try
        {
            var resp = await SendRequestAsync(url);
            var isRespObj = !resp.StartsWith('[');
            var address = isRespObj ? JObject.Parse(resp)["address"] :
JArray.Parse(resp)[0]["address"];
            res = ParseAddress(address);
        }
        finally
        {
            semaphore.Release();
        }
        return res;
    }

    private Address ParseAddress(JToken address){
```

```
        string city = string.Empty;
        string district = address["district"] != null ? address["district"].ToString() :
(address["borough"] != null ? address["borough"].ToString() : string.Empty);
        string region = address["state"] != null ? address["state"].ToString() :
(address["region"] != null ? address["region"].ToString() : string.Empty);
        string country = address["country"] != null ? address["country"].ToString() :
string.Empty;
        if(address["city"] != null)
        {
           city = address["city"].ToString();
        }
        else if(address["town"] != null)
        {
           city = address["town"].ToString();
        }
        else if(address["village"] != null)
        {
           city = address["village"].ToString();
        }
        return new Address { City = city, Region = region, Country = country,
District = district };
     }

     private async Task<Address> GoAround(double startLat, double startLon)
     {
        double[] directions = { 0, 45, 90, 135, 180, 225, 270, 315 };
        int steps = 6;
        double distance = 0;
        Address result = new Address();
        for(int i = 1; i <= steps && !result.HasRequiredInfo(); ++i)
        {
           List<Task<Address>> tasks = new();
           foreach(double direction in directions)
           {
              double newLat = startLat;
              double newLon = startLon;
              if(distance > 0)
              {
                 var newCoordinates = CalculateNewCoordinates(startLat, startLon,
distance, direction);
                 newLat = newCoordinates.Item1;
                 newLon = newCoordinates.Item2;
              }
```

```
                tasks.Add(TryGetAddress(string.Format(reverse_url, newLat,
newLon)));
                tasks.Add(TryGetAddress(string.Format(search_url, newLat, newLon)));
                if(distance == 0)
                {
                    --i;
                    break;
                }
            }
            var results = await Task.WhenAll(tasks);
            foreach(var res in results)
            {
                if(result.HasRequiredInfo())
                {
                    break;
                }
                result.AssignNotNull(res);
            }
            distance = distance == 0 ? 500 : distance *= 3;
        }
        return result;
    }
    private (double, double) CalculateNewCoordinates(double lat, double lon,
double distance, double direction)
    {
        double radiusEarth = 6371000;
        double radianLat = ToRadians(lat);
        double radianLon = ToRadians(lon);
        double radianDirection = ToRadians(direction);
        double angularDistance = distance / radiusEarth;

        double newLat = Math.Asin(Math.Sin(radianLat) *
Math.Cos(angularDistance) + Math.Cos(radianLat) * Math.Sin(angularDistance) *
Math.Cos(radianDirection));
        double newLon = radianLon + Math.Atan2(Math.Sin(radianDirection) *
Math.Sin(angularDistance) * Math.Cos(radianLat), Math.Cos(angularDistance) -
Math.Sin(radianLat) * Math.Sin(newLat));
        return (ToDegrees(newLat),  ToDegrees(newLon));
    }

    private double ToRadians(double degrees) => degrees * (Math.PI / 180);

    private double ToDegrees(double radians) => radians * (180 / Math.PI);}
}
```