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
_______ Наталія АУШЕВА
“___”____________2025 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “ Інтеграція картографічних даних у програмне забезпечення за допомогою Google Maps API ”

Виконав: студент 4 курсу, групи ТР-12
Ілендзеєр Артем Олександрович ____________
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент каф. ЦТЕ, Артем ГУРІН ____________
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент доцент каф. АЕП, к.т.н., доцент, Павло НОВІКОВ ____________
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент каф. ЦТЕ, Володимир РУДИК ____________
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

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

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
(підпис)
"___" ___________2025 р.

З А В Д А Н Н Я
на дипломну роботу студенту

Ілендзеєру Артему Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи "Інтеграція картографічних даних у програмне забезпечення за допомогою Google Maps API"

Науковий керівник Гурін Артем Леонідович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від " 2 " червня 2025 року № 1875-с

2. Термін подання студентом роботи 09.06.2025р.

3. Вихідні дані до роботи мова програмування Kotlin, набір інструментів Android SDK, середовище розробки Android Studio.

4. Перелік питань, які потрібно розробити 1) провести аналіз серед наявних аналогів; 2) визначити головні функції мобільного застосунку, що дозволяє зберігати геодані; 3) аргументувати вибрані інструменти, алгоритми та технології для реалізації мобільного додатку; 4) побудувати архітектуру програмного забезпечення; 5) створити прикладний програмний інтерфейс користувача; 6) представити процес застосування мобільного застосунку.

5. Орієнтований перелік ілюстративного матеріалу схема архітектури проєкту, схема роботи з віддаленими сервісами, діаграма прецедентів, скріншоти роботи користувачів з застосунком.

6. Дата видачі завдання 13.09.2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	20.02.25	Виконано
2.	Аналіз методів та засобів розв'язання задачі	17-25.04.25	Виконано
3.	Розробка архітектури та загальної структури системи	21-25.04.25	Виконано
4.	Розробка окремих підсистем	25.04-02.05.25	Виконано
5.	Програмна реалізація системи	03-07.05.25	Виконано
6.	Оформлення пояснювальної записки	08-12.05.25	Виконано
7.	Захист програмного забезпечення	15.05.25	Виконано
8.	Передзахист	26.05.25	Виконано
9.	Захист		

Студент ____________________ Артем ІЛЕНДЗЕЄР
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник ____________________ Артем ГУРІН
(підпис) (ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 64 сторінках, містить 18 ілюстрацій, 1 додатку, 25 джерел у списку використаних джерел.

Мета роботи — створення мобільного застосунку для платформи Android з інтеграцією картографічних даних за допомогою Google Maps API.

Методи та засоби: інструменти Android SDK, мова програмування Kotlin, архітектурні підходи Clean Architecture та MVVM, бібліотека для побудови залежностей Dagger Hilt, сервіси Firebase Firestore та Firebase Storage, бібліотека для завантаження зображень Coil, використання корутин для асинхронної обробки даних.

Результат — мобільний застосунок, що дозволяє додавати, редагувати та переглядати географічні об'єкти з відображенням на інтерактивній карті, з можливістю відкриття координат у Google Maps.

Ключові слова: АНДРОІД, ГУГЛ МАПС АПІ, ГЕОЛОКАЦІЯ, ФАЄРБЕЙЗ, КОТЛІН, МОБІЛЬНИЙ ЗАСТОСУНОК.

ABSTRACT

The thesis consists of 64 pages, includes 18 images, 1 appendix, and 25 references in the bibliography.

The aim of the work is to develop a mobile application for the Android platform with integration of cartographic data using the Google Maps API.

Methods and tools: Android SDK toolkit, Kotlin programming language, Clean Architecture and MVVM architectural patterns, Dagger Hilt dependency injection library, Firebase Firestore and Firebase Storage services, Coil image loading library, and coroutines for asynchronous data processing.

Result: a mobile application that allows users to add, edit, and view geographic locations displayed on an interactive map, with the option to open coordinates in Google Maps.

Keywords: ANDROID, GOOGLE MAPS API, GEOLOCATION, FIREBASE, KOTLIN, MOBILE APPLICATION.

ЗМІСТ

ВСТУП ... 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ... 10
1.1 Постановка завдання ... 10
1.2 Актуальність проблеми ... 10
1.3 Огляд існуючих картографічних систем ... 11
1.4 Актуальність рішення для основних операційних систем ... 12
1.5 Аналіз засобів розробки ... 14
2 ТЕХНОЛОГІЇ ТА МЕТОДИ РЕАЛІЗАЦІЇ ... 16
2.1 Обґрунтування вибору архітектурного підходу ... 16
2.2 Переваги Kotlin ... 17
2.3 Аналіз сучасних технологій для проектування програмного забезпечення для Android ... 19
2.3.1 Вибір технології для впровадження залежностей ... 19
2.3.2 Вибір віддаленого сховища ... 21
2.3.3 Переваги Jetpack Compose ... 22
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ... 24
3.1 Проектування архітектури системи ... 24
3.2 Підключення сервісу Google Maps до проекту ... 29
3.3 Налагодження файлів конфігурації ... 31
3.4 Реалізація основної програмної частини ... 33
3.4.1 Створення модулю бізнес-логіки ... 33
3.4.2 Розробка інтерфейсу користувача ... 37
3.4.3 Вирішення проблеми зберігання даних на екрані ... 39
3.4.4 Отримання поточних координат пристрою ... 41
3.4.5 Зберігання даних в віддаленому сховищі ... 43
3.4.6 Отримання оптимального маршруту ... 48
4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ ... 53
4.1 Системні вимоги та встановлення ... 53

4.2 Демонстрація функціоналу програмного забезпечення 54
ВИСНОВКИ 59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 60
ДОДАТОК А 63

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API — Application Programming Interface — інтерфейс прикладного програмування; набір готових функцій, які дозволяють взаємодіяти з іншим програмним забезпеченням або сервісом.

GPS — Global Positioning System — глобальна система позиціонування; технологія визначення координат об’єкта за допомогою супутникової навігації.

UI — User Interface — інтерфейс користувача; сукупність елементів, з якими взаємодіє користувач у програмному забезпеченні.

UX — User Experience — користувацький досвід; враження користувача від взаємодії з програмою або застосунком.

MVVM — Model-View-ViewModel — архітектурний патерн для поділу логіки представлення, бізнес-логіки та моделі даних у програмних додатках.

Clean Architecture — патерн проєктування, що забезпечує поділ програмного коду на шари відповідальності, підвищуючи масштабованість і підтримуваність.

Firebase — набір хмарних сервісів Google для зберігання даних, автентифікації, аналітики та інших функцій для мобільних та веб-додатків.

Firestore — хмарна база даних від Firebase, що дозволяє зберігати структуровані дані з підтримкою синхронізації в реальному часі.

Storage — сервіс від Firebase для зберігання файлів, таких як зображення, відео тощо.

Coil — бібліотека для завантаження та кешування зображень в Android-додатках, написана на Kotlin.

Dagger Hilt — бібліотека для впровадження залежностей у Android-додатках, що забезпечує зручну інтеграцію з архітектурними компонентами.

SDK — Software Development Kit — набір програмних інструментів і бібліотек для розробки програмного забезпечення на певній платформі.

Kotlin — сучасна мова програмування, що використовується для розробки Android-додатків, офіційно підтримується Google.

ВСТУП

У сучасному світі картографічні сервіси стали невід'ємною частиною багатьох цифрових рішень, що використовуються в повсякденному житті. Геолокаційні технології активно застосовуються у таких сферах, як логістика, транспорт, доставка, туризм, урбаністика, мобільна торгівля та безпека. Все частіше користувачі очікують від мобільних застосунків можливість отримання точної інформації про своє розташування, навігацію на мапі, а також взаємодію з географічними об'єктами.

Інтеграція картографічних даних у програмне забезпечення дає змогу підвищити зручність і функціональність мобільних додатків. Google Maps API є одним із найпопулярніших і найпотужніших інструментів для реалізації картографічних можливостей на платформі Android. Використання цього API дає змогу розробникам додавати інтерактивні карти, маркери, шари з даними, маршрути, а також функціонал навігації й геолокації.

Метою даної роботи є розробка мобільного застосунку на Android з інтеграцією картографічних даних за допомогою Google Maps API. Для цього необхідно виконати.

1) Аналіз підходів, методів та алгоритмів інтеграції картографічних даних в програмне забезпечення.

2) Аналіз програмних засобів для реалізації програмної системи з інтегрованим картографічним сервісом.

3) Розробка програмного забезпечення з інтегрованим картографічним сервісом.

4) Тестування розробленого програмного забезпечення.

Актуальність теми полягає у зростаючій потребі бізнесу та користувачів у точній просторовій інформації, яка надається у зручній інтерактивній формі через мобільні застосунки. Це робить інтеграцію картографічних сервісів важливим і перспективним напрямком у розробці програмного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Для реалізації інтегрування картографічних даних в програмне забезпечення використані сучасні технології розробки, такі як мова програмування, фреймворки, бібліотеки. Використані сучасні підходи до розробки, які дозволяють ефективно вирішувати задачі.

1.1 Постановка завдання

Метою цієї роботи є створення мобільного застосунку для платформи Android, що дозволяє інтегрувати картографічні дані з використанням Google Maps API. Основним завданням є реалізація функціоналу для додавання, редагування, перегляду та видалення географічних точок на мапі з прив’язкою до координат, зображень, назв та описів. Дані повинні зберігатися у хмарному середовищі. Також необхідно реалізувати можливість побудови маршруту від поточного розташування користувача до вибраної точки, а також пошук точок за текстовими запитами. Для забезпечення ефективної структури коду застосунок має бути побудований відповідно до принципів чистої архітектури.

1.2 Актуальність проблеми

У сучасному світі стрімко зростає попит на сервіси, пов’язані з доставкою їжі, товарів та поштових відправлень. Сервіси на кшталт Glovo, Nova Poshta, Rozetka активно використовують картографічні віджети у своїх мобільних застосунках для відображення розташування кур’єра, вибору адреси доставки або перегляду місць видачі посилок. Користувачі очікують зручного та наочного інтерфейсу, який дозволяє взаємодіяти з картою у режимі реального часу. Це робить

інтеграцію картографічних сервісів, таких як Google Maps API, не лише корисною, а й необхідною умовою створення конкурентоспроможного програмного забезпечення. Відповідно, розробка інструментів, що дозволяють ефективно працювати з геоданими, є актуальним і затребуваним напрямом у сфері інформаційних технологій [1, 2].

1.3 Огляд існуючих картографічних систем

Сучасні картографічні системи активно використовуються в багатьох сферах, включаючи логістику, транспорт, туризм, моніторинг навколишнього середовища та побутові сервіси. Найбільш відомими з них є Google Maps, OpenStreetMap, Mapbox, Here Maps та Apple Maps.

Google Maps API є одним із найпоширеніших інструментів для інтеграції карт у мобільні та веб-застосунки. Він надає широкий функціонал: відображення карти, маркерів, побудова маршрутів, робота з геолокацією та геокодуванням. Система має високу точність, добре задокументований інтерфейс та активну підтримку спільноти [3].

OpenStreetMap (OSM) — це відкритий проєкт зі створення карт, який дозволяє безкоштовно використовувати картографічні дані. Його головна перевага — відкритість та можливість модифікації даних, однак реалізація повноцінного API для інтеграції часто потребує додаткових рішень.

Mapbox є гнучкою платформою для розробки картографічних застосунків з акцентом на кастомізацію дизайну карт. Вона підтримує інтеграцію з Android та iOS, має зручні інструменти для розробників, але вимагає підключення сторонніх сервісів для роботи з маршрутами або геокодуванням.

Apple Maps — це картографічна система, розроблена компанією Apple, яка є стандартною для пристроїв на iOS. Останні роки Apple значно покращила якість своїх карт, зокрема у великих містах. Для розробників доступний MapKit API, який дозволяє вбудовувати карти у додатки, додавати анотації, створювати маршрути та

виконувати пошук локацій. Проте головним обмеженням є екосистемна замкненість — Apple Maps доступні лише для продуктів Apple, що звужує його застосування для кросплатформних рішень.

Таким чином, для розробки мобільного застосунку з високим рівнем інтерактивності та стабільності найкраще підходить Google Maps API, що й стало обґрунтованим вибором для реалізації даної дипломної роботи.

1.4 Актуальність рішення для основних операційних систем

Інтеграція інтерактивних карт у програмне забезпечення є важливою складовою сучасних цифрових рішень, особливо в контексті стрімкого розвитку логістичних сервісів, доставки товарів, транспортного моніторингу та мобільної навігації. Багато сфер діяльності, зокрема електронна комерція, сервіси виклику таксі, поштові оператори, туристичні додатки та служби доставки їжі, потребують точного відображення просторової інформації, маршрутів та геолокації користувачів. У зв’язку з цим, питання впровадження картографічних даних є актуальним для всіх основних операційних систем [4].

Операційна система Windows найчастіше використовується на стаціонарних і портативних комп’ютерах. Для настільних застосунків підтримка карт зазвичай реалізується через веб-браузер або вбудовані компоненти на базі сторонніх API, таких як Bing Maps або Google Maps [5]. Використання інтерактивної карти в десктопних програмах може бути зручним для систем управління логістикою, складуванням, моніторингу або навігації в межах великих об’єктів. Проте через обмежену мобільність пристроїв під управлінням Windows, використання карти не є критично необхідним для повсякденного користування.

Операційна система macOS, як і Windows, орієнтована на десктопні пристрої, хоча має тісну інтеграцію з екосистемою Apple, зокрема з мобільними пристроями на iOS. На macOS є можливість інтеграції картографічного сервісу Apple Maps через MapKit API, однак цей API доступний лише в межах екосистеми Apple.

Важливо зазначити, що macOS переважно використовується для професійної діяльності, медіаконтенту, розробки тощо, тому необхідність у впровадженні карти виникає рідше, ніж у мобільних рішеннях [6].

Операційна система Linux переважно застосовується у серверних рішеннях, системах з відкритим кодом, а також на пристроях спеціалізованого призначення. Впровадження карт у додатки для Linux також можливе через веб-інтерфейси або бібліотеки, проте рідко зустрічається в повсякденному користуванні [7]. Виняток становлять професійні системи геоінформаційного аналізу (наприклад, QGIS), які активно використовуються в наукових, аналітичних та інженерних галузях [8].

Операційна система iOS — одна з провідних мобільних операційних систем, що активно використовується в усьому світі. Вона забезпечує зручну інтеграцію з Apple Maps через MapKit, а також підтримує сторонні рішення через WebView. Карти широко застосовуються в додатках для навігації, доставки, транспорту та комерції. Проте розробка застосунків для iOS має низку обмежень: висока вартість пристроїв, сувора політика Apple щодо публікації застосунків у App Store, а також необхідність розробки на macOS з використанням Xcode.

Android є найпоширенішою мобільною операційною системою у світі. Її відкрита архітектура, велика частка ринку, гнучка політика публікації застосунків у Google Play, а також доступність великої кількості пристроїв різного цінового сегменту роблять її найпривабливішою платформою для розробки мобільних додатків. Google Maps API для Android надає потужні можливості з інтеграції карт, геолокації, побудови маршрутів, додавання міток, відображення інформації та багато іншого. Застосунок, що інтегрує ці функції, може бути легко масштабований, швидко оновлюваний та адаптований під різні потреби кінцевих користувачів [9].

Отже, можна зробити висновок, що хоч інтеграція картографічних сервісів можлива на всіх сучасних операційних системах, найбільш виправданим і ефективним є впровадження подібного функціоналу саме в мобільних додатках [10]. Через мобільність, повсякденне використання смартфонів та попит на сервіси з інтегрованою геолокацією, платформа Android є найкращим вибором для реалізації програмного продукту з вбудованою інтерактивною картою. Вона

поєднує в собі гнучкість, широку аудиторію користувачів, відкриті інструменти розробки та доступ до потужних сервісів Google Maps API [11].

1.5 Аналіз засобів розробки

Для створення сучасного, функціонального та масштабованого мобільного застосунку на платформі Android було обрано набір перевірених інструментів і технологій, що забезпечують ефективну розробку, підтримку та подальший розвиток програмного продукту.

Android Software Development Kit (SDK) — це офіційний набір інструментів для створення додатків під операційну систему Android. Він включає компілятор, інструменти для налагодження, емулятори, бібліотеки та приклади коду. Android SDK підтримує усі необхідні інтерфейси для роботи з сенсорами, GPS, камерою, мережею та картографічними сервісами, зокрема Google Maps API [12].

Android Studio — офіційне інтегроване середовище розробки (IDE) від Google для створення Android-додатків. Воно базується на IntelliJ IDEA та надає потужні інструменти для дизайну інтерфейсу, роботи з кодом, відлагодження, тестування та профілювання. Android Studio підтримує Android Emulator, який дозволяє запускати та тестувати додаток на різних версіях Android та розмірах екранів [13].

Kotlin — сучасна мова програмування, яка є офіційно рекомендованою Google для Android-розробки. Вона відзначається лаконічністю, безпекою щодо null-посилань, високою продуктивністю та сумісністю з Java. Kotlin підтримує об'єктно-орієнтовану і функціональну парадигми, що дозволяє ефективно реалізовувати бізнес-логіку додатків [14, 15, 16].

Jetpack Compose — сучасний декларативний фреймворк від Google для створення інтерфейсів користувача в Android-додатках. На відміну від традиційного підходу з використанням XML-розмітки, Compose дозволяє описувати UI безпосередньо в коді, що спрощує його підтримку, тестування та

повторне використання. Compose інтегрується з ViewModel, навігацією, анімацією та іншими компонентами Jetpack.

Firestore — це хмарна документно-орієнтована база даних у реальному часі від Google, яка дозволяє зберігати структуровані дані та забезпечує синхронізацію між клієнтами. Вона підтримує складні запити, масштабування та офлайн-доступ, що робить її ідеальним рішенням для мобільних застосунків, де потрібне швидке збереження та оновлення інформації про об’єкти, такі як геомітки або текстові описи.

Firebase Storage — це сервіс зберігання файлів (зокрема зображень) у хмарі. Він забезпечує надійне, масштабоване та безпечне сховище для мобільних додатків. У рамках даного проєкту Firebase Storage використовується для збереження фотографій об’єктів, прив’язаних до відповідних точок на мапі.

Retrofit — це потужна бібліотека для взаємодії з REST API, яка використовується для отримання зовнішніх даних, зокрема з сервісів, таких як Google Directions API. Вона підтримує парсинг відповідей у форматі JSON, легко інтегрується з Coroutines та дозволяє зручно будувати HTTP-запити.

Dagger Hilt — офіційний інструмент для впровадження залежностей (Dependency Injection) в Android-проєкти. Він дозволяє керувати життєвим циклом об’єктів, спрощує їх тестування та модульність коду. У даному застосунку Dagger Hilt забезпечує зручне впровадження ViewModel, репозиторіїв та інших компонентів.

Короутини в Kotlin — це механізм для асинхронного програмування, що дозволяє виконувати довготривалі операції (наприклад, мережеві запити або звернення до бази даних) без блокування основного потоку. Завдяки короутинам досягається висока продуктивність додатку та плавність взаємодії користувача з інтерфейсом.

2 ТЕХНОЛОГІЇ ТА МЕТОДИ РЕАЛІЗАЦІЇ

Розробка мобільного застосунку з інтеграцією картографічної інформації вимагає глибокого розуміння технологій, які забезпечують роботу з геоданими, базами даних і взаємодією між компонентами системи. Особливу увагу слід приділяти вибору інструментів розробки, що дозволяють створити продуктивний, зручний і стабільний застосунок. Важливим є також урахування архітектурних рішень, які забезпечують масштабованість, підтримуваність і простоту тестування. Застосування перевірених технологічних рішень та інструментів дозволяє реалізувати ефективний і сучасний програмний продукт, орієнтований на потреби користувача.

2.1 Обґрунтування вибору архітектурного підходу

Розробка масштабованого та підтримуваного програмного забезпечення потребує чіткого архітектурного підходу, що дозволяє відокремити логіку, відповідальність компонентів та забезпечити легкість у тестуванні, повторному використанні коду та впровадженні змін. У процесі розробки мобільного застосунку було обрано Clean Architecture (чиста архітектура) — підхід, який базується на принципах незалежності, модульності та інверсії залежностей.

Clean Architecture структурує програму за рівнями. Рівень презентації (UI) відповідає за відображення даних і взаємодію з користувачем. Він залежить від ViewModel, яка керує станом інтерфейсу.

Рівень домену (бізнес-логіка) містить абстрактні інтерфейси та сценарії (use cases), які не залежать від жодної платформи, фреймворку або бібліотеки. Це ядро додатку, і воно є максимально незалежним.

Рівень даних реалізує конкретні способи взаємодії з зовнішніми джерелами даних (бази даних, API, файлові системи тощо). Цей рівень взаємодіє з Firebase

Firestore, Firebase Storage та Google Directions API у вигляді реалізацій відповідних інтерфейсів, оголошених у доменному рівні [21].

Такий підхід забезпечує:

— легкість у тестуванні — завдяки ізоляції бізнес-логіки її можна перевіряти незалежно від інших компонентів;

— гнучкість — заміна Firebase на іншу базу даних можлива без змін у доменній логіці;

— масштабованість — архітектура дозволяє легко додавати нові функції без порушення існуючої структури;

— зменшення зв'язності — компоненти не залежать один від одного безпосередньо, їх пов'язують лише через інтерфейси [22].

Clean Architecture ідеально поєднується з таким сучасним інструментом Android-розробки як Dagger Hilt — інструмент для впровадження залежностей, що автоматизує створення та життєвий цикл об'єктів, дозволяє легко передавати залежності між модулями [23].

Вибір Clean Architecture обґрунтований необхідністю створити довготривалий, адаптивний та легко розширюваний програмний продукт, який відповідатиме вимогам реального бізнесу та мінімізує технічний борг у майбутньому. Такий підхід дозволяє створювати застосунки, які можуть розвиватися разом із новими потребами користувачів та технологічними змінами, не втрачаючи при цьому стабільності та якості.

2.2 Переваги Kotlin

Вибір мови програмування для мобільної розробки на платформі Android є одним із ключових рішень, що впливають на ефективність, продуктивність розробника, якість коду та підтримуваність застосунку. Протягом багатьох років основною мовою для створення Android-додатків була Java, однак із 2017 року компанія Google офіційно визнала мову Kotlin як повноцінну мову для Android-

розробки, а з 2019 року Kotlin став рекомендованою мовою для створення нових Android-застосунків.

Хоча Java досі використовується в багатьох проєктах, Kotlin має низку суттєвих переваг, які роблять її більш зручною та сучасною мовою розробки.

Kotlin дозволяє писати менше коду для реалізації тієї ж функціональності. Завдяки вбудованим функціям, таким як data-класи, extension-функції, деструктуризація та лямбда-вирази, багато шаблонного коду, характерного для Java, автоматично усувається.

Kotlin реалізує строгий контроль за значеннями, які можуть бути null, що значно зменшує кількість помилок типу NullPointerException — однієї з найпоширеніших проблем у Java [17].

Також він компілюється у байт-код JVM, тому його можна без проблем використовувати разом із існуючим Java-кодом. Це дозволяє поетапно переносити великі проєкти або використовувати наявні Java-бібліотеки.

Kotlin підтримує функціональні концепції, як-от map, filter, reduce, які значно полегшують обробку колекцій і роблять код декларативним і читабельним [18].

А ще Kotlin має нативну підтримку корутин — легковагових потоків, які спрощують написання асинхронного та багатопотокового коду. Це особливо важливо для мобільних застосунків, де важливо уникати блокування головного потоку [19]. Завдяки сучасному синтаксису, чіткості конструкцій та мінімізації шаблонного коду, Kotlin-код легше читати, підтримувати та тестувати [20].

Kotlin має розвинену систему типів, яка дозволяє виражати більш складну логіку типів без надмірної складності коду.

У контексті розробки Android-застосунку з інтегрованими картографічними сервісами, базами даних та асинхронною логікою обробки даних, Kotlin демонструє себе як мова, що дозволяє створювати ефективні, безпечні й легко підтримувані рішення. Його вбудована інтеграція з Android SDK, підтримка Android Studio, а також сумісність із Jetpack Compose, Retrofit, Firebase, Dagger Hilt та Coroutines забезпечує сучасний рівень розробки.

Отже, з огляду на описані переваги, вибір Kotlin як основної мови програмування для даного проєкту є повністю обґрунтованим. Він не лише сприяє підвищенню продуктивності розробника, а й забезпечує якість і довговічність створеного програмного продукту.

2.3 Аналіз сучасних технологій для проектування програмного забезпечення для Android

У цьому розділі розглянуто сучасні технології та інструменти, що використовуються під час проєктування програмного забезпечення для операційної системи Android.

Особливу увагу приділено бібліотекам, засобам роботи з інтерфейсом користувача, а також можливостям взаємодії з зовнішніми сервісами та апаратними компонентами пристрою. Аналіз проведено з урахуванням вимог до надійності, масштабованості та зручності підтримки програмного продукту.

2.3.1 Вибір технології для впровадження залежностей

Впровадження залежностей (Dependency Injection, DI) є фундаментальним підходом у сучасній розробці програмного забезпечення, що дозволяє спростити управління залежностями між компонентами, підвищити тестованість коду та забезпечити гнучку модульну архітектуру. Особливо це актуально для Android-розробки, де застосунок має складну структуру з численними залежностями, пов’язаними з життєвим циклом компонентів системи.

На сьогоднішній день існує декілька популярних рішень для реалізації впровадження залежностей у середовищі Android:

Dagger 2 — один із найвідоміших та найпотужніших фреймворків для DI, створений компанією Google. Він забезпечує статичну генерацію коду на етапі

компіляції, що гарантує високу продуктивність. Проте його конфігурація є складною та потребує великої кількості шаблонного коду, що ускладнює початкове налаштування.

Koin — легковажний фреймворк для DI, написаний повністю на Kotlin. Його ключова перевага — простота у використанні та мінімальна конфігурація. Втім, оскільки Koin реалізує впровадження динамічно (через рефлексію), це може негативно впливати на продуктивність великих застосунків.

Kodein — ще одна легка бібліотека для DI, яка також відома простотою та гнучкістю. Вона добре підходить для невеликих проєктів, але менш активно підтримується в екосистемі Android порівняно з іншими рішеннями.

Dagger Hilt — офіційне розширення над Dagger 2, створене спеціально для Android. Hilt значно спрощує процес інтеграції DI у проєкт за рахунок попередньо налаштованих компонентів, які відповідають життєвому циклу Android-компонентів (Activity, Fragment, ViewModel тощо). Hilt підтримується командою Android, повністю інтегрований із Android Jetpack і дозволяє уникнути надлишкового шаблонного коду, властивого Dagger 2.

У межах цього проєкту вибір було зроблено на користь Dagger Hilt, з огляду на наступні переваги:

Офіційна підтримка Google та Android Jetpack — гарантує стабільність, оновлення та відповідність сучасним стандартам розробки.

Автоматичне управління життєвим циклом компонентів — забезпечує зручне впровадження залежностей у ViewModel, Activity, Fragment без необхідності вручну створювати модулі та компоненти.

Покращена зручність конфігурації — спрощений синтаксис та анотації (@HiltAndroidApp, @AndroidEntryPoint, @Inject, @Module, @Provides) дозволяють швидко впроваджувати DI без великої кількості коду.

Компіляторна перевірка залежностей — забезпечує виявлення помилок на етапі компіляції, що підвищує надійність коду.

Таким чином, з урахуванням вимог проєкту, архітектурного підходу Clean Architecture, використання ViewModel та потреби в модульності — Dagger Hilt є

найбільш доцільним та ефективним вибором для реалізації впровадження залежностей у даному Android-застосунку.

2.3.2 Вибір віддаленого сховища

У сучасній розробці мобільних застосунків важливим є наявність надійного віддаленого сховища, яке забезпечує зберігання, синхронізацію та обмін даними між клієнтами в реальному часі. Для реалізації поставленої задачі були обрані сервіси Firebase Firestore Database та Firebase Storage, які надають масштабовані та ефективні рішення для зберігання структурованих даних та мультимедійного контенту відповідно.

Firebase Firestore є документно-орієнтованою базою даних нового покоління, яка дозволяє зберігати та обробляти структуровані дані у вигляді колекцій і документів. Вибір цієї технології для зберігання текстових даних (назв, описів, координат) був зумовлений такими перевагами:

Синхронізація в реальному часі. Firestore автоматично оновлює дані на всіх пристроях, що забезпечує актуальність інформації без необхідності ручного оновлення.

Масштабованість. Система підтримує велику кількість одночасних клієнтів і здатна обробляти високі навантаження, що особливо важливо для багатокористувацьких застосунків.

Інтеграція з Firebase. Firestore легко поєднується з іншими сервісами екосистеми Firebase, такими як Authentication, Storage, Cloud Functions тощо.

Безпека. Налаштування правил доступу на основі користувачів та ролей дозволяє точно контролювати, хто може читати чи записувати дані.

Кросплатформеність. Підтримка Android, iOS, Web і Server SDK дозволяє легко масштабувати застосунок на різні платформи без необхідності змін у серверній частині.

Завдяки вищезазначеним перевагам, Firebase Firestore став оптимальним вибором для реалізації логіки збереження геолокаційних міток, текстових описів і метаданих.

Для зберігання зображень, які додаються користувачами до точок на карті, було використано Firebase Storage — масштабоване об’єктне сховище, спеціально призначене для зберігання файлів великого обсягу, таких як фотографії, відео та інші мультимедійні ресурси. Основні переваги цього сервісу.

Надійне зберігання та висока доступність. Firebase Storage використовує Google Cloud Storage у якості основи, що гарантує надійність і безперервний доступ до даних.

Інтеграція з Firebase Authentication. Дає можливість легко обмежувати доступ до файлів відповідно до прав користувача.

Гнучка система посилань. Після завантаження файлу система автоматично генерує URL, який можна зберегти у Firestore та використовувати для завантаження зображень у клієнтському застосунку.

Автоматичне масштабування. Платформа автоматично обробляє великий обсяг запитів, що дозволяє не турбуватися про серверну інфраструктуру.

Вбудована безпека. Можна налаштувати правила доступу на основі ідентифікації користувача або типу запиту.

Поєднання Firestore Database для зберігання структурованої інформації та Firebase Storage для роботи з файлами дозволяє ефективно розділити відповідальність між двома сервісами, підвищити надійність застосунку та забезпечити його масштабованість.

2.3.3 Переваги Jetpack Compose

Jetpack Compose — це сучасний інструментарій для створення інтерфейсів користувача в Android-застосунках, розроблений компанією Google. На відміну від традиційного підходу з використанням XML-розмітки до побудови UI, Jetpack

Compose пропонує декларативну модель, яка значно спрощує процес розробки, супроводу та тестування інтерфейсів.

Серед основних переваг Jetpack Compose варто виокремити наступні:

Декларативність. Інтерфейс створюється як функція стану, що дозволяє легко визначати, як має виглядати UI в залежності від даних. Це значно зменшує кількість коду, необхідного для відображення змін у інтерфейсі при оновленні стану.

Менше коду. Завдяки компактності синтаксису та відсутності необхідності у XML-файлах, Jetpack Compose дозволяє реалізовувати навіть складні UI-компоненти у вигляді простих функцій, що підвищує читабельність і знижує ймовірність помилок.

Jetpack Compose органічно працює з архітектурними компонентами Android, зокрема з ViewModel, що дозволяє зберігати стан UI при зміні конфігурації (наприклад, при повороті екрана, або при зміні системної теми пристрою).

Розробник має повний контроль над виглядом і поведінкою кожного елементу інтерфейсу. Компоненти легко стилізуються, комбінуються та розширюються.

Jetpack Compose підтримує всі принципи Material Design “з коробки”, що дозволяє створювати сучасні, візуально привабливі інтерфейси без надмірних зусиль [24].

Інтерактивні прев’ю та гаряче оновлення (Live Preview). Android Studio надає зручні інструменти для перегляду інтерфейсів у реальному часі, що значно пришвидшує розробку та спрощує налагодження UI.

Єдність підходу. На відміну від традиційної побудови UI на Android, де логіка розпорошена між XML-файлами, класами Activity/Fragment, адаптерами та іншими компонентами, Jetpack Compose дозволяє зосередити всю логіку в одному місці — в Kotlin-коді.

У контексті реалізації мобільного застосунку, де важливо забезпечити простоту інтерфейсу, швидке оновлення відображення даних та підтримку адаптивного дизайну, Jetpack Compose є оптимальним вибором. Його використання дозволяє створити сучасний, інтуїтивно зрозумілий та легко підтримуваний інтерфейс користувача, що відповідає актуальним вимогам до Android-додатків.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

Реалізація програмного забезпечення є ключовим етапом у процесі створення мобільного застосунку, оскільки саме на цьому етапі ідеї, закладені в архітектурі та технічному проєктуванні, втілюються у вигляді робочого продукту. Успішна реалізація залежить від правильного вибору інструментів, дотримання обраних принципів розробки, а також ефективної взаємодії між логікою застосунку, інтерфейсом користувача та зовнішніми сервісами. Програмна реалізація повинна не лише відповідати функціональним вимогам, але й забезпечувати зручність використання, стабільність роботи та масштабованість у майбутньому.

3.1 Проектування архітектури системи

Для початку проектування архітектури системи спочатку проектується діаграма прецедентів (рис. 3.1).

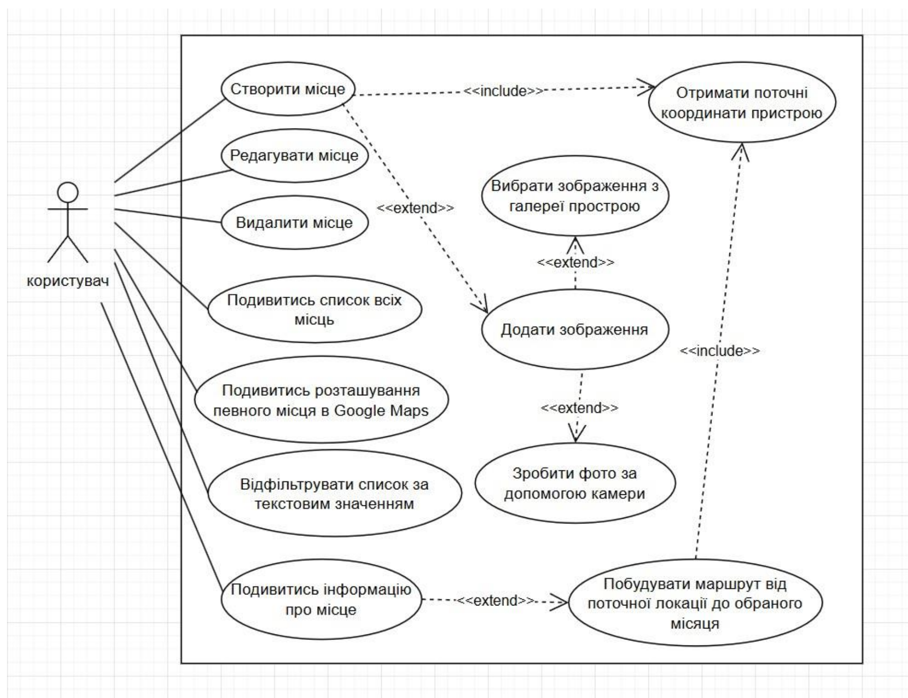

Рисунок 3.1 — Діаграма прецедентів

Наступним кроком визначається діаграма основних класів системи окремо для кожного модуля.

Спочатку проектується структура модулю бізнес-логіки. Це основний модуль, який описує роботу усього додатку. Такий опис робиться через абстракції, тому створюються інтерфейси, які будуть реалізовані в модулях додатку даних. (рис. 3.2)

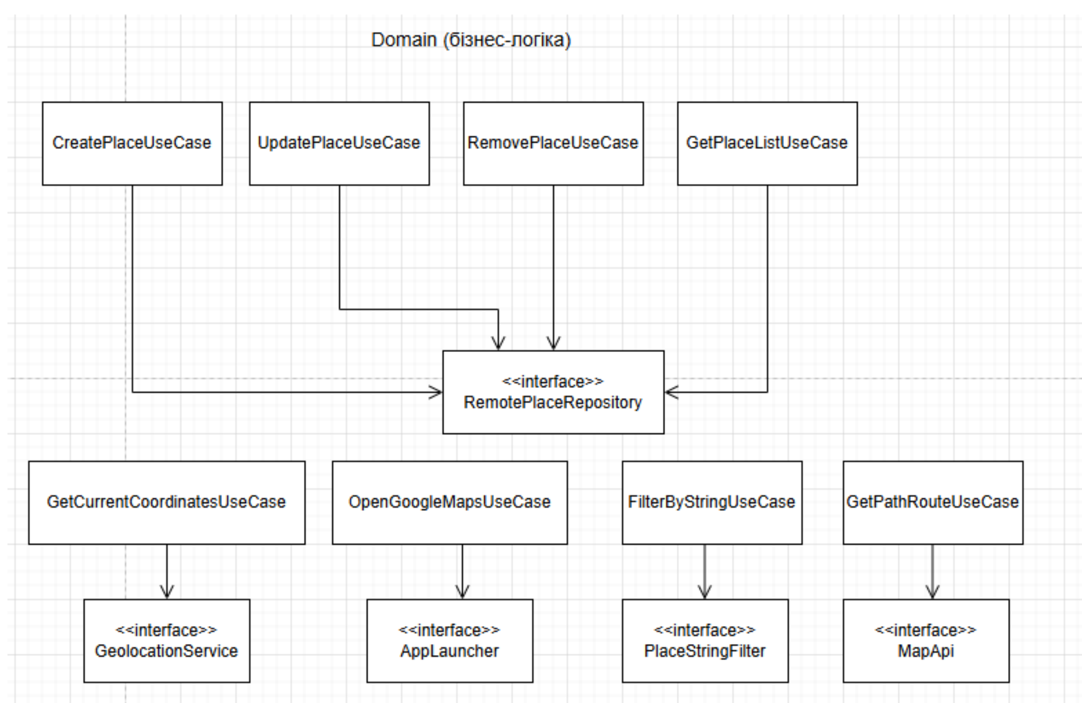

Рисунок 3.2 — Діаграма класів модулю бізнес-логіки

Створюється окремий клас для кожного варіанту використання. Для роботи з віддаленим сховищем створюється інтерфейс RemotePlaceRepository. Для отримання координат пристрою створюється інтерфейс Geolocation Service. Для запуску додатку Google Maps створюється інтерфейс AppLauncher. Для фільтрації списка точок створюється інтерфейс PlaceStringFilter. Для отримання оптимального маршруту з сервіса Directions API створюється інтерфейс MapApi.

Після проектування діаграми класів модулю бізнес-логіки, проектується структура модулю даних. В модулі реалізовуються інтерфейси, створені в модулі бізнес-логіки, а також створюються інші класи, для розділення відповідальності

над окремими процесами. Таке розділення дозволить легше контролювати логіку, тестувати її, а також заміняти окермі частини логіки, не порушуючи інші частини. (рис. 3.3)

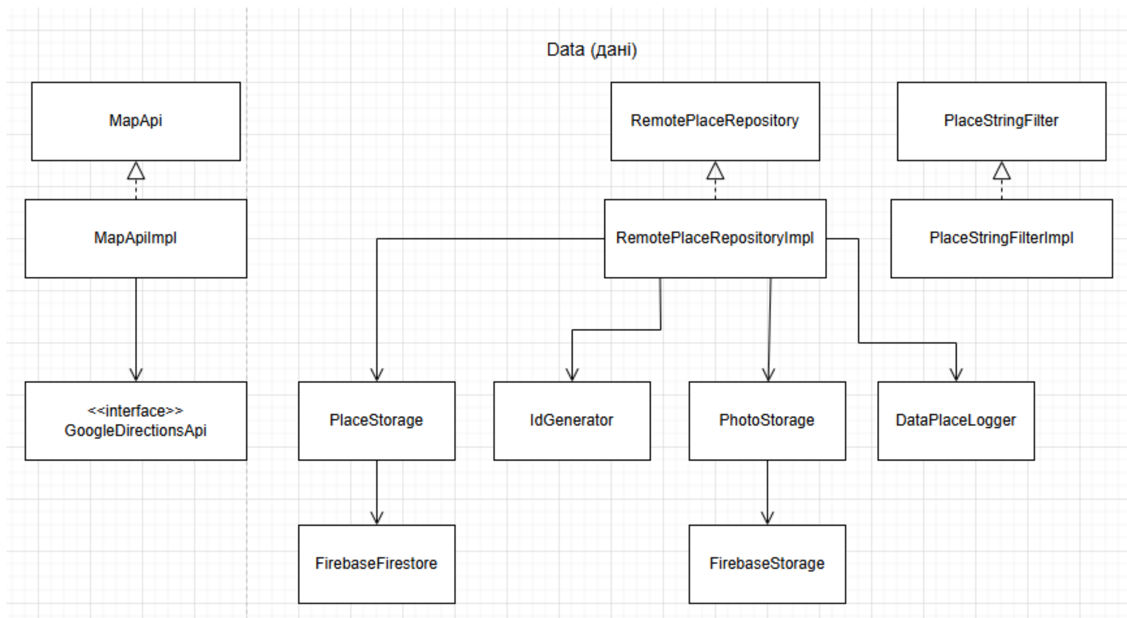

Рисунок 3.3 — Діаграма класів модулю даних

Оскільки було вирішено використовувати Retrofit для взаємодії з API, то MapApiImpl використовує інтерфейс, а реалізація буде автоматично створена бібліотекою та передана в конструктор класу.

Для роботи безпосередньо з сутностями Place створюється клас Place Storage, який в свою чергу використовує конкретний сервіс для збереження в віддалене сховище — FirebaseFirestore.

Для автоматичної генерації ідентифікаторів для сутностей створюється клас IdGenerator.

Для роботи в віддаленому сховищі безпосередньо з зображеннями створюється клас PhotoStorage, який в свою чергу використовуэ конкретний сервіс для збереження зображень в віддалене сховище — FirebaseStorage.

Для обробки помилок, а також для спостерігання за роботою з віддаленим сховищем, створюється клас DataPlaceLogger.

Після проектування модулю даних, створюється діаграма класів останнього модулю — модулю додатку. В цьому модулі розташовуються класи роботи з графічним інтерфейсом, а також з системними компонентами, такі як інтернет, bluetooth, та інші. (рис. 3.4)

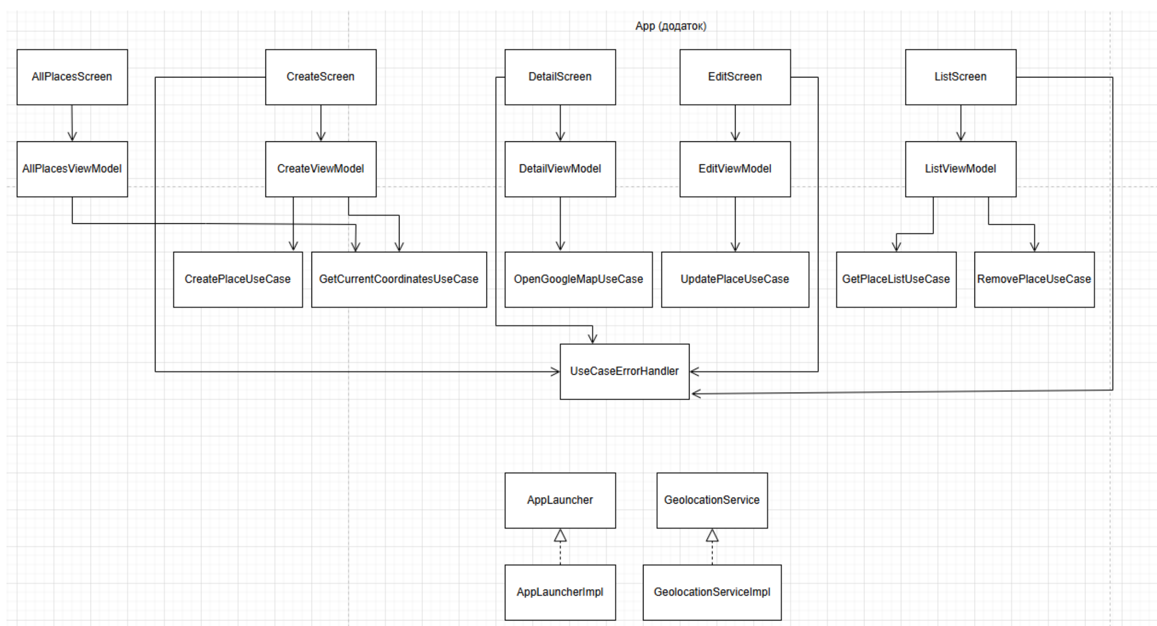

Рисунок 3.4 — Діаграма класів модулю додатку

Спочатку створюються класи для кожного екрану. Кожний екран використовує свій особистий клас ViewModel для роботи з даними, а також для їх збереження. Кожний клас ViewModel використовує класи варіантів використання з модулю бізнес-логіки, які повинні виконуватись у відповідному екрані.

Також створюється оброблювач помилок, для більш коректної роботи додатку при неочікуваних результатах.

Створена реалізація інтерфейсу AppLauncher. Реалізація містить конкретну логіку запуску додатку Google Maps. Також створена реалызація інтерфейсу GeolocationService.

Сторонні сервіси надають можливість винести частину обчислювальної або інформаційної логіки за межі основного програмного забезпечення, що дозволяє зменшити навантаження на клієнтську частину, скоротити обсяг локального коду та

значно розширити функціональні можливості системи. Такі сервіси надають доступ до різноманітних ресурсів через інтернет, зокрема для роботи з геолокацією, побудови маршрутів, зберігання зображень тощо.

З метою забезпечення ефективної взаємодії з такими сервісами була розроблена відповідна схема (рис. 3.5), яка ілюструє принципи інтеграції та обміну даними.

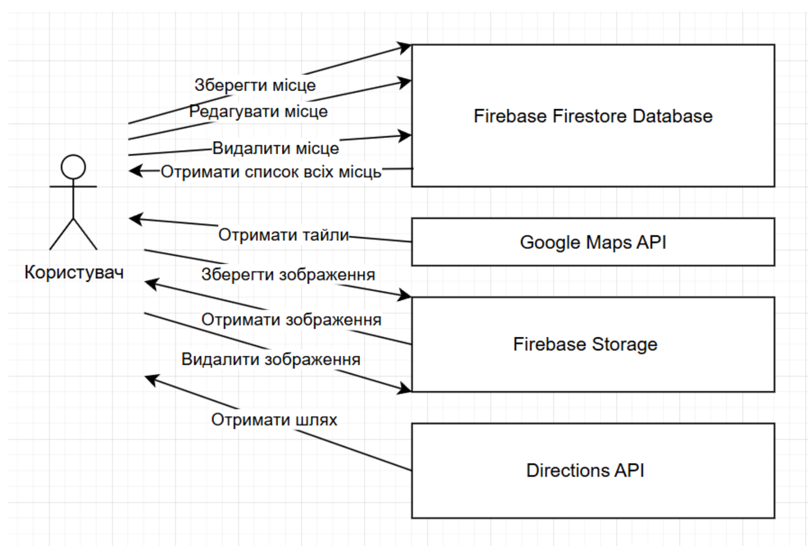

Рисунок 3.5 — Схема взаємодії зі сторонніми сервісами

Схема відображає, яким чином програмна система комунікує з зовнішніми API, зокрема Google Maps Directions API, для отримання оптимальних маршрутів, а також з Google Maps API, для отримання тайлів карти.

Взаємодія з сервісом Firestore Database включає в себе також і відправку даних, для того щоб ці дані були збережені в віддаленому сховищі. Також присутні запити на видалення та редагування цих даних. Основним запитом є отримання списку всіх місць.

Взаємодія з Firebase Firestore включає в себе запити на додавання зображення, видалення, а також отримання.

Такий підхід забезпечує масштабованість, модульність і гнучкість архітектури програмного забезпечення.

Побудована структура програми дозволяє розробити програмне забезпечення, поступово створюючи незалежні один від одного частини. Така структура дозволяє з мінімальним зусиллям розширювати та підтримувати додаток.

3.2 Підключення сервісу Google Maps до проекту

Для підключення сервісу Google Maps до Android-проєкту необхідно виконати кілька обов’язкових кроків. Перш за все, потрібно додати необхідні залежності до конфігураційного файлу Gradle. Йдеться про файл build.gradle.kts, який відповідає за налаштування модуля app. У цьому файлі, всередині блоку dependencies, необхідно додати наступні рядки:

— implementation("com.google.android.gms:play-services-maps:17.0.1");

— implementation("com.google.android.gms:play-services-location:18.0.0");

— implementation("com.google.maps.android:maps-compose:2.11.4").

Ці залежності забезпечують базову підтримку роботи з Google Maps, можливість взаємодії з геолокацією пристрою, а також інтеграцію з бібліотекою Jetpack Compose для декларативного відображення карти.

Після додавання залежностей важливо синхронізувати проєкт із Gradle, щоб усі необхідні бібліотеки були завантажені, і середовище розробки могло коректно їх розпізнати.

Наступним кроком є підключення API ключа Google Maps, який видається в Google Cloud Console для конкретного проєкту. Для цього необхідно відкрити файл AndroidManifest.xml і додати всередину тегу <application> наступний фрагмент: <meta-data android:name="com.google.android.geo.API_KEY" android:value="значення-ключа" />

З міркувань безпеки у цьому прикладі значення ключа приховано. У реальному застосунку замість значення-ключа необхідно підставити дійсний API ключ, згенерований у вашому обліковому записі Google Cloud.

Після успішного підключення залежностей та API ключа, можна переходити безпосередньо до відображення карти в одному з екранів додатку. У рамках архітектури Jetpack Compose для цього використовують функцію GoogleMap(), яка дозволяє легко вставити карту в будь-який Composable-компонент.

Щоб спростити використання карти та винести логіку її відображення в окрему функцію, створюється Composable-функція ShowMap() (рис. 3.6).

```
@Composable
fun ShowMap(
    modifier: Modifier = Modifier,
    lat: Double,
    lng: Double,
    pathPoints: List<LatLng>,
    title: String? = "",
    snippet: String? = "",
) {
    val markerPosition = LatLng(lat, lng)
    val cameraPositionState = rememberCameraPositionState {
        position = CameraPosition.fromLatLngZoom(markerPosition, defaultMapZoom)
    }

    GoogleMap(
        modifier = modifier,
        cameraPositionState = cameraPositionState
    ) {
        Polyline(pathPoints, color = Color.Blue, width = 8f)
        Marker(
            state = MarkerState(position = markerPosition),
            title = title,
            snippet = snippet
        )
    }
}
```

Рисунок 3.6 — функція ShowMap()

Ця функція приймає кілька параметрів: modifier — стандартний об’єкт з Jetpack Compose, який дозволяє налаштовувати розміри, відступи, взаємодію з користувачем та інші стилістичні аспекти компонента; lat — значення широти місця, яке потрібно відобразити на карті; lng — значення довготи; title — текстовий заголовок, який буде відображений у вікні маркера; snippet — додатковий опис маркера.

У тілі функції спочатку створюється об’єкт LatLng, який репрезентує географічну позицію на основі переданих широти та довготи. Ця координата

використовується як точка, в якій буде розташований маркер: val markerPosition = LatLng(lat, lng)

Наступним кроком створюється об’єкт cameraPositionState, який відповідає за поточне положення камери на карті. Він ініціалізується зі значенням, що відповідає маркеру, та встановленим масштабом перегляду.

Значення масштабу визначається змінною defaultMapZoom, яка за замовчуванням дорівнює 10. Це дозволяє автоматично сфокусувати камеру на потрібній області карти при її завантаженні: “val cameraPositionState = rememberCameraPositionState { position = CameraPosition.fromLatLngZoom(markerPosition, defaultMapZoom) }

Далі викликається функція GoogleMap(), яка відображає саму карту. У неї передаються параметри modifier та cameraPositionState, щоб правильно налаштувати вигляд та позицію камери: GoogleMap(modifier = modifier, cameraPositionState = cameraPositionState)

Всередині GoogleMap визначено блок для відображення маркерів. За допомогою функції Marker() створюється маркер, який буде відображено в заданій позиції.

Параметри title та snippet дозволяють додати до маркера підпис і опис, які з’являються при взаємодії з ним. Стан маркера (MarkerState) створюється безпосередньо у виклику, але при бажанні його можна винести за межі функції для більшої гнучкості та можливості зміни положення маркера в реальному часі.

Таким чином, ця функція дозволяє швидко й ефективно відобразити карту з маркером у потрібній точці, а також забезпечує просту інтеграцію карти в інтерфейс користувача.

3.3 Налагодження файлів конфігурації

Щоб мобільний застосунок функціонував коректно та взаємодіяв з усіма необхідними апаратними й програмними ресурсами, потрібно налаштувати

конфігураційні файли проєкту. Основними серед них є файл AndroidManifest.xml та файли Gradle, які визначають властивості застосунку, дозволи та залежності.

AndroidManifest.xml — це службовий XML-файл, який містить основну інформацію про застосунок: назву, іконку, компоненти (активності, сервіси, провайдери), а також список дозволів, які застосунок потребує для доступу до апаратних і програмних ресурсів Android-пристрою.

Зокрема, у Manifest необхідно вказати такі дозволи.

Для використання камери (але без обов'язкової її наявності на пристрої): <uses-permission android:name="android.permission.CAMERA" /> <uses-feature android:name="android.hardware.camera" android:required="false" />

Для доступу до геолокації: <uses- permission android:name="android.permission.ACCESS_FINE_LOCATION" /> <uses- permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

Для зчитування файлів із внутрішнього сховища (зокрема, зображень із галереї): <uses- permission android:name="android.permission.READ_EXTERNAL_STORAGE" /> <uses- permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

Для доступу до мережі Інтернет: <uses- permission android:name="android.permission.INTERNET" /> Також у тегу <application> потрібно вказати параметр: android:name=".app.App"

Це необхідно для коректного запуску фреймворку Dagger Hilt, який потребує доступу до базового контексту застосунку.

Gradle — це система автоматичного збирання проєктів, що використовується в Android Studio. Вона керує залежностями, налаштуванням збірки та компіляцією. Для кожного модуля проєкту (app, data, domain) існує окремий файл build.gradle.kts, у якому зазначаються відповідні залежності.

У файлі build.gradle.kts модуля app слід додати: implementation(project(":domain")) implementation(project(":data"))

Для модуля data: implementation(project(":domain")). Модуль domain є базовим і не містить залежностей до інших модулів.

Крім того, для коректної роботи Dagger Hilt необхідно додати до модуля app наступне: kapt { correctErrorTypes = true }

Також у секції plugins файлу Gradle слід додати плагіни, які відповідають за Kotlin, Android, Hilt та інші технології, що використовуються у проєкті.

Правильне налаштування цих конфігураційних файлів забезпечує стабільну роботу застосунку, спрощує масштабування і підтримку коду, а також дозволяє коректно інтегрувати зовнішні сервіси та бібліотеки.

3.4 Реалізація основної програмної частини

У цьому розділі описано процес реалізації основної програмної частини мобільного додатку, включаючи побудову користувацького інтерфейсу, реалізацію бізнес-логіки, організацію навігації між екранами, зберігання даних та інтеграцію з зовнішніми сервісами.

3.4.1 Створення модулю бізнес-логіки

Створення модулю бізнес-логіки починається зі створення основної моделі/сутності, з якою буде працювати основна частина коду в цьому модулі.

Цей клас реалізований як data class у мові Kotlin, що дозволяє автоматично генерувати функції equals(), hashCode(), toString() та інші, спрощуючи роботу з об’єктами цього типу (рис. 3.7).

id: PlaceID — унікальний ідентифікатор точки. Тип PlaceID є псевдонімом для String, оголошеним за допомогою ключового слова typealias. Це дозволяє підвищити семантичну зрозумілість коду: замість загального типу String явно зазначено, що ця змінна є ідентифікатором місця.

name: String — назва точки. Відображається в списку об’єктів і на карті, є основним текстовим маркером для користувача.

lat: Double — географічна широта точки. Визначає вертикальну координату на карті.

lng: Double — географічна довгота точки. Визначає горизонтальну координату на карті. description:

String? — текстовий опис об’єкта.

Поле є необов’язковим (nullable) і може бути використане для додаткової інформації про точку (наприклад, коментар або короткий опис призначення). photoUrl: String? — URL-адреса зображення, прикріпленого до точки. Це також необов’язкове поле, що дозволяє візуально ідентифікувати об’єкт або місце.

```
data class Place(
    val id: PlaceID,
    val name: String,
    val lat: Double,
    val lng: Double,
    val description: String?,
    val photoUrl: String?
)

typealias PlaceID = String
```

Рисунок 3.7 — Клас сутності місця

В конструктор передається інтерфейс RemotePlaceRepository. Створюється функція execute(data: CreatePlaceData): Result, яка викликає функцію з інтерфейсу RemotePlaceRepository.

Варто зазначити, що асинхрона функція викликається в діспетчері Dispatchers.IO за допомогою корутин, тому що ця функція виконує запит в інтернет. Потім визначається тип результату, який повертається з функції.

Клас CreatePlaceData використовується для збирання та передачі даних, необхідних на етапі створення нової географічної точки до її збереження у віддаленому сховищі.

На відміну від основної сутності Place, яка містить повноцінну інформацію про вже збережений об’єкт (зокрема, ідентифікатор та URL зображення),

CreatePlaceData репрезентує тільки вхідні дані, які користувач вводить у формі створення точки (рис. 3.8).

Цей підхід дозволяє чітко відокремити дані введення від даних, які вже є частиною структури системи.

```
data class CreatePlaceData(
    val name: String,
    val lat: Double,
    val lng: Double,
    val description: String?,
    val photoByteArray: ByteArray?
)
```

Рисунок 3.8 — Клас даних для створення сутності місця

Опис полів: name: String — назва точки, яку вводить користувач. Це обов’язкове поле, що слугує основною міткою для ідентифікації об’єкта на карті. lat: Double — широта точки. Визначається автоматично або задається вручну відповідно до місця на карті, яке вибрав користувач. lng: Double — довгота точки. Так само, як і широта, є частиною координат об’єкта. description: String? — додатковий текстовий опис точки. Є необов’язковим полем, яке дозволяє деталізувати призначення або характеристики об’єкта. photoByteArray: ByteArray? — зображення, прикріплене до точки у вигляді масиву байтів. Це поле використовується для передачі фотографії на сервер через Firebase Storage, після чого генерується відповідний URL, який вже зберігається у полі photoUrl класу Place. Масив байтів є зручним форматом для тимчасового зберігання зображень у пам’яті додатку до моменту завантаження.

Такий поділ моделей сприяє кращій структуризації коду, спрощує валідацію даних під час створення точки, та забезпечує гнучкість у роботі з різними етапами життєвого циклу об’єктів у застосунку.

Після створення моделі даних наступним етапом є реалізація варіантів використання (use cases), які описують бізнес-логіку застосунку в абстрактній

формі. Одним із таких класів є CreatePlaceUseCase, що відповідає за процес створення нової точки на мапі (рис. 3.9).

```
class CreatePlaceUseCase(
    private val remotePlaceRepository: RemotePlaceRepository
) {
    suspend fun execute(data: CreatePlaceData): Result {
        val result = withContext(Dispatchers.IO) {
            remotePlaceRepository.create(data)
        }
        return when (result) {
            is RemotePlaceRepository.CreateResult
            .Success -> Result.Success(result.created)

            is RemotePlaceRepository.CreateResult
            .Error -> Result.Error(result.message)
        }
    }

    sealed class Result {
        data class Success(val created: Place) : Result()
        data class Error(val message: String) : Result()
    }
}
```

Рисунок 3.9 — Клас створення місця

У конструктор класу впроваджується інтерфейс RemotePlaceRepository, що дозволяє працювати з віддаленим джерелом даних. Основна функція execute(data: CreatePlaceData): Result виконує асинхронний виклик функції створення точки через цей репозиторій.

Оскільки операція включає мережеву взаємодію, вона виконується в окремому потоці з використанням корутин і диспетчера Dispatchers.IO, що є оптимальним для задач введення-виведення.

Після завершення виклику результат обробляється за допомогою конструкції when, і повертається відповідний тип — або Success з інформацією про створену точку, або Error з повідомленням про помилку.

Таким чином, клас CreatePlaceUseCase інкапсулює логіку створення точки у вигляді незалежного сценарію використання, забезпечуючи чистоту архітектури та відокремлення бізнес-логіки від конкретних реалізацій.

Інші класи варіантів використання розроблені аналогічним способом.

3.4.2 Розробка інтерфейсу користувача

Для роботи інтерфейсу користувача необхідно налагодити навігації між екранами.Функція AppNavHost є центральним елементом навігації. Вона приймає об’єкт NavHostController, який відповідає за управління переходами між екранами. У тілі функції оголошується NavHost, де вказано початковий маршрут (у даному випадку ListScreenRoute) і задається логіка переходів між окремими екранами: списку об’єктів, створення нової точки, редагування, перегляду детальної інформації, а також відображення всіх точок на мапі.

Для кожного маршруту визначено, які параметри він приймає, та які компоненти мають відображатися. Наприклад, екран ListScreen підтримує навігацію до створення, редагування чи перегляду об’єкта, передаючи відповідні дані через навігаційний контролер.

Оскільки застосунок оперує складними структурами даних, такими як об’єкти UiPlace або списки цих об’єктів, використано спеціальні типи параметрів навігації — так звані користувацькі типи (NavType). Зокрема, об’єкт PlacesList є реалізацією NavType<List<UiPlace>>, що дозволяє передавати список об’єктів через маршрути.

Цей тип реалізує функцію для серіалізації та десеріалізації даних у форматі JSON. Функція serializeAsValue також кодує URL зображень для безпечної передачі. Таке рішення дозволяє безпечно передавати дані між екранами через рядкові параметри, зберігаючи чистоту архітектури та уникнувши прямої залежності між компонентами.

Після налагодження навігації проектується програмний код екранів. В Jetpack compose такий код пишеться всередені функцій, помічених анотацією @Composable.

Принцип побудови екрану у середовищі Jetpack Compose можна продемонструвати на прикладі CreateScreen, який побудовано із використанням декларативного підходу Jetpack Compose. Компоненти інтерфейсу користувача

ретельно структуровані та організовані в ієрархію, що забезпечує зручність взаємодії та логічне розташування елементів.

Компонент Column(modifier) є базовим контейнером, що розміщує дочірні елементи вертикально один під одним. Всередині нього використано ScrollableColumn (нині застарілий, але в цьому контексті слугує для вертикального скролінгу вмісту), що дозволяє прокручувати вміст екрана при надлишку інформації.

Основні UI-компоненти:

PlaceholderPhotoView — кастомний компонент, призначений для відображення зображення, яке може завантажити користувач. Він займає всю ширину екрана (fillMaxWidth) і реагує на дію користувача — вибір зображення. Отримане зображення у вигляді ByteArray зберігається у змінній byteArray.

TextField — стандартний компонент для введення тексту, що відображає поле вводу назви точки (name). Має підпис (label), автоматично зв’язаний зі змінною стану name.

BasicTextField — компонент для введення опису точки. Текстове поле оформлене вручну з обрамленням (border), відступами (padding) та кастомізованим стилем (textStyle). Це дозволяє створити поле вводу з багаторядковою підтримкою (minLines = 3), що підходить для введення описових даних.

CoordChooser — ще один кастомний компонент, який відповідає за вибір координат. Він передає обрані значення широти та довготи до зовнішніх змінних lat і lng.

Button — кнопка створення об’єкта, натискання якої викликає функцію create() у viewModel. Функція отримує всі введені дані: назву, координати, опис та фото. Після успішного завершення дії виконується навігація назад до екрана списку (ListScreenRoute).

Під час запуску екрана (LaunchedEffect) ініціюється отримання координат через функцію viewModel.fetchCoords().

Інший LaunchedEffect слухає зміну координат з потоку viewModel.coords, та оновлює локальні змінні lat і lng, які використовуються у формі.

3.4.3 Вирішення проблеми зберігання даних на екрані

Однією з особливостей системи Android є те, що при зміні конфігурації пристрою — наприклад, при повороті екрана, зміні мови або перемиканні між світлою та темною темами — компонент Activity, який відповідає за відображення інтерфейсу користувача, автоматично знищується та створюється заново. Це може призводити до втрати стану інтерфейсу та втрачених даних, які користувач уже ввів або які були завантажені.

Для вирішення цієї проблеми в архітектурі Android застосовується підхід, що базується на використанні ViewModel. Цей компонент є частиною архітектурних компонентів Android Jetpack і призначений для зберігання та обробки даних, необхідних для інтерфейсу, незалежно від життєвого циклу Activity або Fragment. ViewModel дозволяє зберігати стан екрана під час зміни конфігурації та забезпечує зручну взаємодію з бізнес-логікою без повторного завантаження або втрати інформації.

У межах цього проєкту для екрана створення нової географічної точки реалізовано клас CreateViewModel. Його завдання — зберігати тимчасовий стан введених даних, керувати запитами до зовнішніх джерел (наприклад, отримання координат), а також виконувати логіку створення нової точки за допомогою відповідного use case. ViewModel зберігає такі поля, як назва точки, опис, координати та зображення у вигляді ByteArray, які користувач може ввести або змінити.

Оскільки CreateViewModel існує незалежно від життєвого циклу UI-компонентів, це дозволяє зберігати інформацію між повторними відтвореннями інтерфейсу, що критично важливо для стабільної роботи застосунку. Це рішення забезпечує більш передбачувану поведінку, покращує досвід користувача та підвищує надійність додатку.Анотація @HiltViewModel вказує, що даний клас є ViewModel, яку створює та керує нею фреймворк Dagger Hilt — популярний інструмент для впровадження залежностей в Android. Це дозволяє автоматично

передавати необхідні залежності до конструктора класу без явного створення екземпляра вручну.

Конструктор @Inject constructor(private val useCases: CreateUseCases, private val errorHandler: UseCaseErrorHandler)

ViewModel отримує дві залежності: CreateUseCases — об’єкт, який містить усі сценарії використання (use cases), пов’язані зі створенням географічної точки та UseCaseErrorHandler — обробник помилок, який дозволяє коректно реагувати на збої в логіці (наприклад, показати повідомлення користувачеві).

Змінна coords private val _coords = MutableStateFlow<Coordinates?>(null) val coords: Flow<Coordinates?> get() = _coords

_coords — приватне поле типу MutableStateFlow, яке зберігає координати, отримані з use case.

coords — відкритий інтерфейс доступу до координат у вигляді потоку (Flow), який можуть підписуватися інші компоненти (наприклад, Composable UI) для реактивного оновлення.

Функція create() fun create(name: String, lat: Double, lng: Double, description: String?, photoByteArray: ByteArray?, onSuccess: () -> Unit)

Ця функція викликається для створення нової точки на карті. Його логіка така:

1) створюється об’єкт CreatePlaceData, який інкапсулює всі необхідні дані для створення точки;

2) функція виконується всередині корутини в viewModelScope, що гарантує коректне керування життєвим циклом;

3) викликається useCases.create.execute(data) — головний сценарій створення;

4) у випадку успіху викликається onSuccess() — колбек для переходу назад або оновлення інтерфейсу. У разі помилки використовується errorHandler для її обробки.

Функція fetchCoords() (рис. 3.10) відповідає за отримання поточних координат пристрою, які надалі використовуються як стартові значення при

створенні нової точки. Він також виконується в корутині та оновлює _coords, що дозволяє автоматично оновити пов’язані з координатами UI-елементи.

```
fun fetchCoords() {
    viewModelScope.launch {
        _coords.value = useCases.getCoordinates.execute()
    }
}
```

Рисунок 3.10 — функція fetchCoords()

CreateViewModel інкапсулює всю бізнес-логіку створення точки: використовує корутини (viewModelScope.launch) для асинхронної роботи, використовує StateFlow/Flow для реактивної передачі даних до UI, забезпечує інверсію залежностей через Dagger Hilt, забезпечує розділення відповідальностей, де UI відповідає лише за відображення, а ViewModel — за обробку подій та виклики логіки.

Це приклад добре організованого та масштабованого коду, що відповідає сучасним практикам Android-розробки. Інші ViewModel класи в системі створені аналогічним чином.

3.4.4 Отримання поточних координат пристрою

Однією з важливих функцій мобільних додатків із картографічною складовою є можливість визначення поточного місцезнаходження користувача. Отримання координат пристрою (широти та довготи) дозволяє покращити користувацький досвід, автоматизувати введення даних, забезпечити релевантність результатів пошуку на мапі, а також реалізувати інші функції, пов’язані з геолокацією.

У системі Android основним способом доступу до геоданих є використання Google Play Services Location API. Цей API забезпечує більш точне,

енергоефективне й оптимізоване отримання місцезнаходження порівняно з використанням базового LocationManager.

Одним із компонентів API є клас FusedLocationProviderClient, який надає єдиний інтерфейс для роботи з різними джерелами локації — GPS, Wi-Fi, мобільні мережі тощо. Для асинхронної обробки результатів у сучасних Android-додатках доцільно використовувати Kotlin-корутини, що дозволяє зручно інтегрувати запити на геолокацію у фонові задачі.

У межах реалізації було створено клас GeolocationServiceImpl, який відповідає за отримання поточних координат пристрою (рис. 3.11).

Клас реалізує інтерфейс GeolocationService, який абстрагує логіку отримання координат.

fusedLocationClient — екземпляр клієнта, який ініціалізується за допомогою контексту застосунку через LocationServices.getFusedLocationProviderClient(...). Він забезпечує доступ до останнього відомого місцезнаходження пристрою.

@SuppressLint("MissingPermission") — анотація, яка приглушує попередження про відсутність перевірки дозволів. У реальному застосунку перед викликом цієї функції повинна виконуватись перевірка дозволів ACCESS_FINE_LOCATION або ACCESS_COARSE_LOCATION.

```
class GeolocationServiceImpl(
    private val appContext: Context
) : GeolocationService {

    private var fusedLocationClient: FusedLocationProviderClient =
        LocationServices.getFusedLocationProviderClient(appContext)

    @SuppressLint("MissingPermission")
    override suspend fun getCurrentCoordinates(): Coordinates {
        val location = fusedLocationClient.lastLocation.await()
        return Coordinates(
            lat = location.latitude,
            lng = location.longitude
        )
    }
}
```

Рисунок 3.11 — клас GeolocationServiceImpl

getCurrentCoordinates() — асинхронна функція, яка викликає lastLocation.await() для отримання останнього зафіксованого місцезнаходження. Після цього координати широти (latitude) та довготи (longitude) обгортаються в об’єкт Coordinates, що використовується у внутрішній логіці програми.

Завдяки цій реалізації геолокаційні дані можна швидко й ефективно інтегрувати в інші частини застосунку, зокрема для передзаповнення полів при створенні нової географічної точки або для відображення користувача на мапі.

3.4.5 Зберігання даних в віддаленому сховищі

Зберігання інформації про місця відбуваєтсья за допомогою сервісу FirebaseFirestore. Для зручного використання функціоналу цього сервісу створений клас PlaceStorage.

В конструктор класу передається FirebaseFirestore.

Всередені класу знаходиться приватний параметр placesRoute, який є шляхом до директорії зі збереженими місцями: private val placesRoute = "graduation/db/places".

Однією з ключових функцій класу PlaceStorage є функція create, який відповідає за додавання нового місця до бази даних. Ця функція є suspend-функцією, що означає її виконання в асинхронному режимі з використанням корутин. Такий підхід дозволяє уникнути блокування основного потоку при роботі з мережею або базою даних, що є особливо важливим для забезпечення плавного користувацького досвіду.

Сигнатура функції виглядає наступним чином: suspend fun create(place: StoragePlace): Result.

Функція приймає один параметр place — об’єкт типу StoragePlace, який містить усі необхідні дані про місце, що зберігається. Усередині функції відбувається спроба записати цей об’єкт до колекції в базі даних Firebase. Для цього викликається функція collection(placesRoute), який звертається до відповідного

маршруту в базі даних. Далі, через document(place.id) створюється або оновлюється документ з унікальним ідентифікатором місця, після чого викликається функція set(place) — саме він здійснює збереження переданого об’єкта у базі.

Операція завершення перевіряється за допомогою функції await(), що забезпечує завершення асинхронної дії перед переходом до наступного кроку. У разі успішного виконання функція повертає об’єкт Result.Success, який сигналізує про успішне збереження місця.

У випадку, якщо під час виконання виникає помилка (наприклад, через проблеми з мережею або некоректні дані), вона перехоплюється за допомогою блоку catch. У цьому блоці створюється та повертається об’єкт Result.Error, який містить повідомлення про помилку.

Ще однією важливою функцією класу PlaceStorage є функція update, який використовується для оновлення вже наявного запису про місце в базі даних. Його реалізація дуже подібна до функції create, оскільки обидві використовують один і той самий підхід для запису даних до Firebase Firestore.

Сигнатура функції: suspend fun update(place: StoragePlace): Result

Ця функція приймає той самий тип параметра — об’єкт StoragePlace, який містить актуалізовану інформацію про місце. Функція update призначений для того, щоб внести зміни до документа з відповідним ідентифікатором у колекції місць. Він звертається до колекції за маршрутом placesRoute, знаходить документ із ідентифікатором place.id і викликає функція set(place).

Використання функції set() у цьому контексті має важливу особливість — якщо документ із таким ідентифікатором уже існує, він буде перезаписаний новими даними, а якщо не існує — буде створений новий документ. Таким чином, функція фактично виконує як оновлення, так і створення в одному виклику, що є характерною особливістю Firestore.

Як і у випадку з функцією create, тут використовується оператор await() для асинхронного очікування завершення операції, а також конструкція try-catch для обробки потенційних помилок. У разі успіху повертається об’єкт Result.Success, у разі винятку — Result.Error з повідомленням про помилку.

Наступною важливою функцією класу PlaceStorage є функція remove, який відповідає за видалення місця з бази даних за заданим ідентифікатором. Ця функція реалізує операцію Delete, що є невід’ємною частиною повноцінної CRUD-логіки.

Сигнатура функції виглядає так: suspend fun remove(id: String): Result

Функція приймає один параметр — id типу String, який є унікальним ідентифікатором місця, що підлягає видаленню. У тілі функції за допомогою звернення до db.collection(placesRoute) здійснюється доступ до відповідної колекції в базі даних. Далі, через document(id) вибирається документ, який відповідає вказаному ідентифікатору, і викликається функція delete().

Операція виконується асинхронно, тому після delete() застосовується функція await(), яка дозволяє дочекатися завершення запиту без блокування потоку. В разі успішного виконання функція повертає об’єкт Result.Success, що сигналізує про те, що місце було успішно видалене з бази даних.

Як і в інших функціях, реалізована обробка помилок за допомогою конструкції try-catch. Якщо під час виконання запиту виникає виняткова ситуація — наприклад, коли документа з таким id не існує, або сталася помилка підключення до мережі — функція повертає об’єкт Result.Error з текстом помилки, що був перехоплений із винятку.

Крім створення, оновлення та видалення місць, у класі PlaceStorage реалізована функція get, яка відповідає за отримання конкретного місця за його унікальним ідентифікатором. Ця функція реалізує операцію Read з CRUD-функціональності та дозволяє завантажити об’єкт типу StoragePlace із бази даних.

Сигнатура функції має вигляд: suspend fun get(id: String): FetchResult

Функція приймає параметр id — рядок, що ідентифікує конкретне місце в базі даних. Після звернення до колекції місць за допомогою db.collection(placesRoute) викликається функція document(id), який дозволяє перейти до документа з вказаним ідентифікатором. Далі використовується функція get() для отримання цього документа з бази, а await() забезпечує очікування завершення операції без блокування потоку.

Після успішного отримання документа перевіряється його наявність за допомогою document.exists(). Якщо документ існує, він перетворюється на об’єкт класу StoragePlace за допомогою функції toObject(StoragePlace::class.java). Цей об’єкт загортається у результат FetchResult.Success і повертається як результат виконання функції.

У випадку, якщо документа з таким id не існує, генерується виняток з відповідним повідомленням, який одразу перехоплюється в блоці catch. Також перехоплюються й інші виняткові ситуації, пов’язані з мережею або некоректним запитом. У разі помилки повертається об’єкт FetchResult.Error із повідомленням про причину збою.

Останньою з ключових функцій класу PlaceStorage є функція getAll, який забезпечує отримання повного списку всіх збережених у базі даних місць. Ця функція реалізує ще один варіант операції Read, але на відміну від функції get, яка працює з одним елементом, getAll завантажує усі документи з колекції.

Сигнатура функції: suspend fun getAll(): FetchListResult

Ця функція не приймає жодних параметрів, оскільки її мета — зчитати весь вміст колекції за шляхом placesRoute. Для цього вона викликає функцію get() у ланцюжку db.collection(placesRoute), після чого використовується await() для асинхронного отримання результатів без блокування потоку.

Після завершення запиту повертається список документів, що відповідають усім об’єктам у колекції. Отримані документи перетворюються у список об’єктів типу StoragePlace за допомогою функції mapNotNull. Це дозволяє виключити ті записи, які з якоїсь причини не вдалося успішно конвертувати у необхідний тип.

Результат роботи функції обгортається у спеціальний об’єкт FetchListResult.Success, який містить у собі повний список місць. Якщо під час виконання запиту виникає помилка — наприклад, через втрату інтернет-з’єднання або помилку доступу до бази — вона перехоплюється в блоці catch, і повертається результат FetchListResult.Error з текстом помилки.

Для збереження та обробки фотографій, пов’язаних із кожним місцем, у проєкті реалізовано окремий клас — PhotoStorage. Цей клас виконує роль обгортки

над сервісом Firebase Storage, забезпечуючи простий і зручний інтерфейс для додавання та видалення зображень у хмарному сховищі.

Конструктор класу приймає як параметр екземпляр FirebaseStorage, що надає доступ до відповідного розділу у хмарному сховищі Firebase. Усередині класу створено змінну reference, яка вказує на кореневу директорію сховища, з якої надалі формуються повні шляхи до конкретних файлів.

Ключова функція класу — add, яка має наступну сигнатуру: suspend fun add(id: String, photoByteArray: ByteArray?): String?

Ця функція дозволяє завантажити фотографію до хмарного сховища, де вона буде збережена під унікальним іменем, сформованим на основі переданого ідентифікатора місця ($id.jpg). Завдяки цьому забезпечується зв'язок між зображенням і відповідним місцем у базі даних.

Якщо параметр photoByteArray не є порожнім, функція використовує його для завантаження байтового масиву у Firebase через функцію putBytes. Після завершення операції, що виконується асинхронно за допомогою await(), отримується публічне посилання на зображення через downloadUrl.await(). Це посилання повертається як результат функції — його надалі можна використовувати для збереження у базі даних, відображення на карті чи в інтерфейсі користувача.

Окрім додавання фото, клас також дозволяє видаляти зображення за наявною URL-адресою. Для цього використовується функція: fun remove(url: String)

Ця функція ініціалізує видалення зображення зі сховища, звертаючись до нього через функцію getReferenceFromUrl(url), після чого викликає delete(). Видалення фото може бути корисним, наприклад, при видаленні місця або заміні зображення.

Таким чином, PhotoStorage інкапсулює логіку взаємодії з Firebase Storage, ізолюючи цю функціональність від решти додатка і забезпечуючи зручний, зрозумілий та безпечний механізм керування медіаданими.

3.4.6 Отримання оптимального маршруту

У багатьох сучасних картографічних застосунках важливою функціональністю є побудова оптимального маршруту між двома або більше географічними точками.

Для цього зазвичай використовуються зовнішні сервіси, які надають відповідні алгоритми пошуку маршрутів, враховуючи тип транспорту, трафік, дорожні умови та інші фактори. Одним із найпопулярніших сервісів такого типу є Google Maps Directions API.

Google Directions API — це частина набору API від Google Maps Platform, яка дозволяє запитувати маршрути між точками. API повертає у форматі JSON або XML детальну інформацію про маршрут: координати всіх точок маршруту, відстань, приблизний час

Для реалізації HTTP-запитів в Android-проєктах зазвичай використовується бібліотека Retrofit — один із найпопулярніших інструментів для роботи з REST API. Retrofit спрощує процес формування запитів, обробки відповідей і серіалізації/десеріалізації JSON-даних за допомогою адаптерів, зокрема Kotlin-корутини або RxJava [25].

Основні переваги Retrofit:

— зручний опис API у вигляді інтерфейсів;

— автоматична обробка JSON через бібліотеки типу Gson або kotlinx.serialization;

— вбудована підтримка корутин;

— гнучка система обробки помилок.

Наступним кроком розроблено інтерфейс GoogleDirectionsAPI, який описує виклик до Google Maps Directions API за допомогою бібліотеки Retrofit (рис. 3.12). Цей інтерфейс використовується для надсилання HTTP-запитів до зовнішнього сервісу з метою отримання маршруту між двома географічними точками.

```
interface GoogleDirectionsAPI {

    @GET("/maps/api/directions/json")
    suspend fun getPath(
        @Query("origin") start: String,
        @Query("destination") end: String,
        @Query("key") key: String,
    ): Response<DirectionsResponse>
}
```

Рисунок 3.12 — інтерфейс GoogleDirectionsAPI

@GET("/maps/api/directions/json") — анотація Retrofit, яка вказує тип HTTP-запиту (GET) та відносний шлях до ресурсу. У даному випадку викликається маршрут /maps/api/directions/json, що відповідає Google Directions API.

Запит повертає результат у форматі JSON.

Функція getPath(...) — асинхронна функція, яка позначена ключовим словом suspend, що означає її сумісність з Kotlin-корутинами. Це дозволяє виконувати запити без блокування головного потоку, що є критично важливим для мобільних застосунків.

Параметри функції передаються як HTTP-параметри запиту за допомогою анотації @Query: start: String — координати або адреса початкової точки маршруту. Наприклад: “50.4501,30.5234”. end: String — координати або адреса кінцевої точки маршруту. key: String — API-ключ, який видається Google для автентифікації запитів.

Повертає об’єкт типу Response<DirectionsResponse>, де DirectionsResponse — це клас-модель, що описує структуру відповіді від API. Він містить деталі маршруту: список точок маршруту, інструкції, тривалість, відстань тощо.

Наступним етапом розроблено клас RetrofitClient (рис. 3.13).

```kotlin
class RetrofitClient {
    private val googleDirectionsBase = "https://maps.googleapis.com/"

    fun googleDirectionApi(): GoogleDirectionsAPI {
        val logging = HttpLoggingInterceptor().apply {
            level = HttpLoggingInterceptor.Level.BODY
        }

        val client = OkHttpClient.Builder()
            .addInterceptor(logging)
            .build()

        return Retrofit.Builder()
            .baseUrl(googleDirectionsBase)
            .addConverterFactory(GsonConverterFactory.create())
            .client(client)
            .build()
            .create(GoogleDirectionsAPI::class.java)
    }
}
```

Рисунок 3.13 — клас RetrofitClient

Цей клас відповідає за створення та налаштування клієнта для взаємодії з Google Directions API через бібліотеку Retrofit. Цей клас інкапсулює конфігурацію HTTP-з’єднання, логування запитів і прив’язку до базового URL API.

Створюється екземпляр OkHttpClient, до якого додається логуючий інтерсептор. Цей клієнт обробляє всі HTTP-з’єднання, виконані Retrofit’ом.

Будується екземпляр Retrofit, який використовує базову адресу googleDirectionsBase, додає конвертер GsonConverterFactory, що відповідає за перетворення JSON-даних у Kotlin-об’єкти, використовує налаштований OkHttpClient з логуванням.

За допомогою функції .create(...) створюється конкретна реалізація інтерфейсу GoogleDirectionsAPI, яка вже готова для використання в репозиторіях або сервісах.

Фінальним етапом реалізації алгоритму отримання оптимального шляху є створення класу MapApiImpl (рис. 3.14).

```kotlin
class MapApiImpl(
    private val api: GoogleDirectionsAPI,
    private val key: String
) : MapApi {
    override suspend fun getPathRoute(start: Coordinates, end: Coordinates): List<Coordinates> {
        val response = api.getPath(
            start = "${start.lat},${start.lng}",
            end = "${end.lat},${end.lng}",
            key = key
        )

        return if (response.isSuccessful) {
            val encodedPolyline = response.body()?.routes?.firstOrNull()?.overviewPolyline?.points
            val listLatLng: List<LatLng> = encodedPolyline?.let {
                PolyUtil.decode(it)
            } ?: emptyList()

            listLatLng.map { Coordinates(it.latitude, it.longitude) }
        } else {
            Log.e(
                tag: "MapApiImpl",
                msg: "getPathRoute() error. start=${start.lat},${start.lng}, " +
                    "end=${end.lat},${end.lng}. ${response.errorBody()}"
            )
            emptyList()
        }
    }
}
```

Рисунок 3.14 — Клас MapApiImpl

Цей клас є конкретною реалізацією інтерфейсу MapApi. Він відповідає за отримання маршруту між двома географічними точками за допомогою Google Directions API. Цей клас виконує HTTP-запит, обробляє відповідь та перетворює отримані дані у внутрішній формат — список координат.

MapApiImpl реалізує інтерфейс MapApi, забезпечуючи конкретну реалізацію функції для отримання маршруту.

У конструктор передаються: api — екземпляр інтерфейсу GoogleDirectionsAPI, через який виконується запит; key — API-ключ для авторизації запиту до Google Maps.

Функція getPathRoute(...) отримує координати початкової та кінцевої точок маршруту і повертає список координат, які утворюють шлях.

Викликається функція getPath() з GoogleDirectionsAPI, який формує HTTP-запит до /maps/api/directions/json.

Значення координат перетворюються у рядковий формат, як це вимагає API.

Якщо запит був успішним (HTTP 200 OK), з відповіді витягується поле overviewPolyline.points, яке містить закодовану лінію маршруту у форматі polyline.

Якщо overviewPolyline є в першому маршруті, він декодується в список об'єктів LatLng за допомогою утиліти PolyUtil.decode(it) (частина бібліотеки Google Maps Android).

Кожен LatLng перетворюється у власний тип Coordinates, після чого список повертається.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Для будь-якого програмного забезпечення важливим є не лише внутрішня логіка та архітектура, а й взаємодія з кінцевим користувачем. Особливу роль відіграє зручність інтерфейсу, логіка навігації, а також відповідність функціоналу очікуванням користувача. Окрім того, важливо враховувати технічні вимоги до пристроїв, на яких буде працювати застосунок, зокрема сумісність з операційною системою, ресурсоємність і стабільність роботи. Взаємодія користувача з програмною системою повинна бути інтуїтивно зрозумілою, а усі функції — доступними без потреби у додаткових інструкціях.

4.1 Системні вимоги та встановлення

Для коректної роботи програмного забезпечення мінімальна версія операційної системи Android повинна бути не нижчою за Android 10. Це забезпечує сумісність із усіма необхідними функціями, зокрема з підтримкою актуальних версій бібліотек та API.

Для повного функціоналу застосунку пристрій повинен мати такі можливості: Камера, яка використовується для зйомки та обробки зображень. GPS, необхідний для визначення геолокації користувача та інтеграції з картографічними даними. Інтернет-з'єднання, яке забезпечує доступ до оновлень карт та інших мережевих функцій застосунку.

Застосунок встановлюється через APK-файл, що забезпечує пряме завантаження та інсталяцію на пристрій. Після завантаження APK-контейнера на пристрій, користувач повинен дозволити встановлення додатків з невідомих джерел у налаштуваннях безпеки.

Після завершення інсталяції користувач повинен перейти в налаштування системи, знайти інстальований застосунок та надати йому всі необхідні дозволи. Також для коректної роботи додатку необхідно активувати геолокацію на пристрої.

4.2 Демонстрація функціоналу програмного забезпечення

Після запуску застосунку користувач потрапляє на головний екран, який відображає список усіх збережених географічних точок (рис. 4.1).

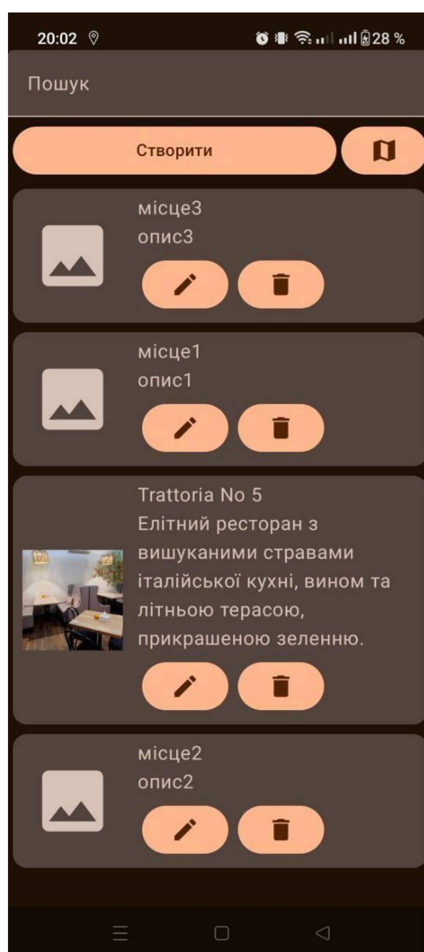

Рисунок 4.1 — Екран списку місць

Цей список є основною частиною інтерфейсу і дозволяє переглядати наявні записи. У верхній частині екрана розташовано текстове поле для пошуку. Введення

будь-якого значення в це поле автоматично фільтрує список точок відповідно до тексту, що був введений, дозволяючи зручно знаходити необхідну інформацію.

Під полем пошуку розміщено дві кнопки: “Створити” та кнопка з іконкою карти. Кнопка “Створити” відкриває екран додавання нової географічної точки (рис. 4.2).

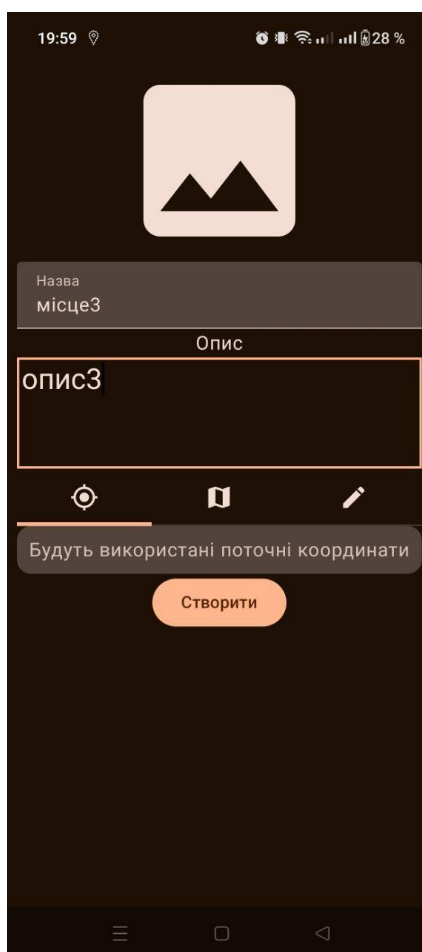

Рисунок 4.2 — Екран створення місця

Екран створення точки включає набір елементів керування, що забезпечують повноцінне введення інформації про нову географічну позицію. У верхній частині цього екрана розміщена кнопка для завантаження фотографії. Під нею знаходяться два текстових поля для введення назви та опису місця. Після натискання на кнопку завантаження фотографії з’являється діалогове вікно з двома варіантами: зробити нове фото за допомогою камери пристрою або вибрати зображення з галереї. Після

вибору зображення, воно відображається в інтерфейсі, замінюючи кнопку, й подальші натискання на зображення повторно відкривають діалог вибору.

Під полем для опису розміщено елемент з трьома вкладками, що дозволяють обрати спосіб визначення координат місця. При активній першій вкладці система автоматично визначає поточне місцезнаходження пристрою за допомогою GPS або мережі. Ці координати зберігаються разом з іншими даними при створенні точки.

При виборі другої вкладки відображається інтерактивна карта. Користувач може взаємодіяти з нею: натискання на будь-яку точку карти встановлює маркер, і вибрані координати зберігаються у системі. Можна змінювати розташування стільки разів, скільки потрібно, доки не буде вибрано бажане положення.

Третя вкладка відкриває два текстових поля для введення широти та довготи вручну. Цей спосіб зручний, коли координати вже відомі користувачу.

Після заповнення всіх необхідних полів та натискання на кнопку “Створити”, відбувається відправлення даних до віддаленого сховища. У разі успішного збереження, користувач автоматично повертається на головний екран, де новостворений об’єкт відображається у загальному списку.

Кожен елемент списку є інформаційним блоком, який відображає назву, опис та, за потреби, мініатюру зображення точки. Під елементом передбачені дві функціональні кнопки — “Редагувати” та “Видалити”. Натискання кнопки редагування відкриває той самий екран, що і для створення, однак усі поля вже попередньо заповнені даними, які можна змінити та зберегти повторно. Кнопка видалення, у свою чергу, видаляє об’єкт із бази даних та миттєво прибирає його з інтерфейсу.

Натискання на кнопку з іконкою карти переводить користувача на екран, де представлена інтерактивна карта з усіма точками (рис. 4.3).

На цій карті маркерами позначені всі географічні точки, що зберігаються у віддаленому сховищі. Користувач може масштабувати карту, переміщуватись по ній і повертати по годинниковій стрілці, або проти неї. При натисканні на маркер на мапі з’являєтсья більш детальна інформація про маркер. Завдяки цьому можна орієнтуватись, яке збережене місце знаходиться в поточній локації.

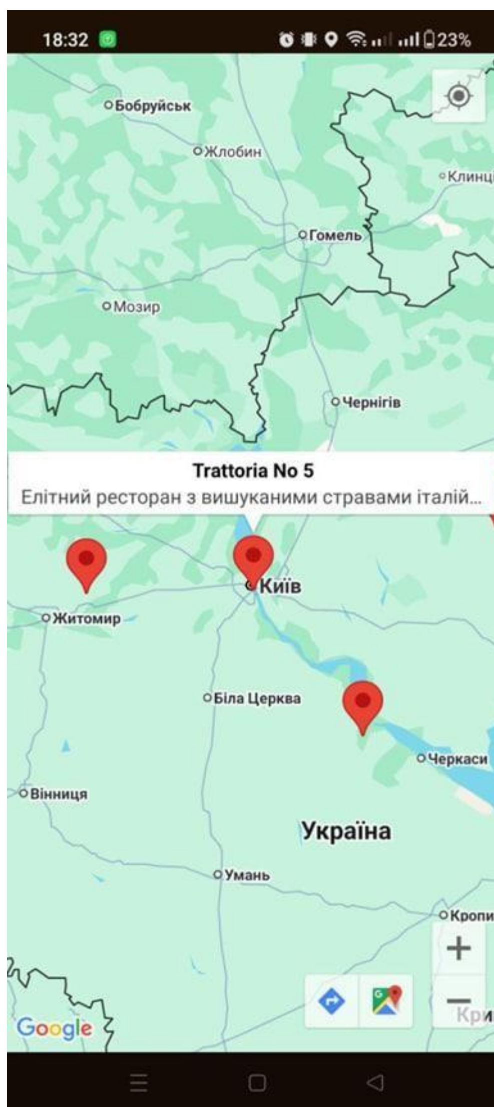

Рисунок 4.3 — Екран з картою усіх точок

З головного екрану (екрану списку) можна також потрапити на екран детальної інформації про місце (рис. 4.4). Для цього треба натиснути на елемент в списку.

У цьому вікні відображається повна інформація, включаючи назву, опис, прикріплене зображення, а також інтерактивну карту, на якій точка позначена відповідним маркером. Карту можна масштабувати, переміщати, використовувати для навігації.

Під картою розміщено кнопку для відкриття точки в Google Maps, що дозволяє перейти до зовнішнього застосунку та використовувати його інструменти.

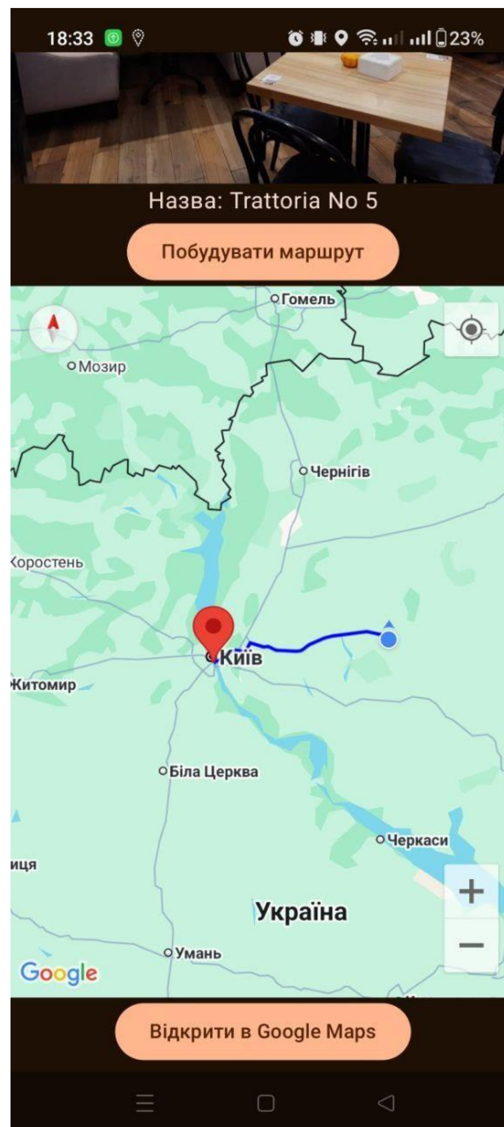

Рисунок 4.4 — Екран з детальною інформацією про вибрану географічну точку

Над картою також розміщено кнопку “Побудувати маршрут”. При її натисканні система автоматично визначає поточне місцезнаходження користувача, надсилає запит до Google Directions API, отримує оптимальний маршрут і виводить його на інтерактивній карті прямо в інтерфейсі додатку. Це дозволяє швидко зорієнтуватися та визначити найкращий шлях до вибраного об’єкта.

ВИСНОВКИ

У результаті виконання дипломної роботи було проаналізовано, обрано та використано сучасні технології, які відповідають актуальним вимогам до створення Android-додатків. Основою програмної реалізації стали такі інструменти, як мова програмування Kotlin, бібліотеки Jetpack Compose для побудови інтерфейсу, Hilt для залежностей, а також Retrofit для роботи з API.

У процесі розробки були ретельно продумані можливі сценарії використання програми, що дозволило створити зручний та інтуїтивно зрозумілий інтерфейс для користувача. Враховано декілька способів введення координат — автоматичне визначення, вибір на карті або ручне введення — що робить додаток універсальним у застосуванні та адаптованим до різних потреб користувачів. Крім того, реалізовано перегляд інформації про кожну точку, побудову маршруту до неї та зручну фільтрацію за назвою.

Для впорядкування структури програмного забезпечення було спроєктовано архітектуру додатку, визначено основні класи та створено діаграми, які ілюструють логічні зв’язки між компонентами системи. Це дозволило ще на етапі проектування забезпечити чітке розмежування відповідальностей між окремими модулями та зменшити ризики виникнення помилок під час розробки.

Написаний код, що відповідає за взаємодію з віддаленим сховищем даних, а також інтеграція з зовнішніми API. Зокрема, було реалізовано механізми надсилання та отримання інформації з бази даних Firebase, роботу з Google Maps API для відображення картографічної інформації та Google Directions API для побудови маршрутів. Також був розроблений код взаємодії з Android-системою.

У підсумку, було успішно створено мобільний додаток для операційної системи Android, що інтегрує функціонал картографічного сервісу Google Maps API та надає користувачу інструменти для збереження, перегляду й навігації до географічних точок. Отриманий результат демонструє ефективне поєднання сучасних технологій, продуманої логіки взаємодії та якісної реалізації програмної частини.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Peter Fitzgibbon. Mapping a more integrated future: the benefits of integrating GIS and ERP systems for enterprises. *GeoConnexion*. URL: https://www.geoconnexion.com/in-depth/mapping-a-more-integrated-future-the-benefits-of-integrating-gis-and-erp-systems-for-enterprises (date of access: 10.05.2025).

2. Ugochukwu_Okonkwo. Integrating GIS with AI and Machine Learning: The Future of Mapping Science for Students and Young Professionals. *Esri Young Professionals Network Blog*. URL: https://community.esri.com/t5/esri-young-professionals-network-blog/integrating-gis-with-ai-and-machine-learning-the/ba-p/1579247 (date of access: 10.05.2025).

3. Philippe. Are There Mapping Tools That Integrate with Other Software Applications? *Mapme Blog*. URL: https://mapme.com/blog/mapping-tools-that-integrate-with-other-software-applications/ (date of access: 10.05.2025).

4. Integration approaches and methods. *ArcGIS Architecture Center*. URL: https://architecture.arcgis.com/en/framework/architecture-practices/integration/integration-approaches-and-methods.html (date of access: 10.05.2025).

5. Світалінський М. Геоінформаційні системи (ГІС). *Nrv.org*. URL: https://nrv.org.ua/geoinformaczijni-systemy-gis/ (дата звернення: 10.05.2025).

6. Favour Agbanusi. Geospatial APIs: How to Integrate Mapping and Location Services in Your Application. *BigGeo*. URL: https://biggeo.com/post/geospatial-apis-how-to-integrate-mapping-and-location-services-in-your-application (date of access: 10.05.2025).

7. Afreen Khalfe. Geolocation Services in Full Stack Development: Integrating Maps and Location-based Features. *Talent500 Blog*. URL: https://talent500.com/blog/geolocation-services-in-full-stack-development/ (date of access: 10.05.2025).

8. Lippold A. GIS and BIM: The Benefits of Integration. *Esri Blog*. URL: https://www.esri.com/arcgis-blog/products/arcgis-pro/aec/gis-bim-the-benefits-of-integration (date of access: 10.05.2025).

9. What are the considerations for integration with geolocation and mapping services in desktop application development? *GTC Systems*. URL: https://gtcsys.com/faq/what-are-the-considerations-for-integration-with-geolocation-and-mapping-services-in-desktop-application-development/ (date of access: 10.05.2025).

10. Daniel Raymond. Top 10 Pros & Cons of Using GIS Mapping Software. *ProjectManagers.net*. URL: https://projectmanagers.net/top-10-pros-cons-of-using-gis-mapping-software/ (date of access: 10.05.2025).

11. Naveen Kumar . Android Usage Statistics 2025: Devices & Market Share. Demand Sage. URL: https://www.demandsage.com/android-statistics/ (date of access: 10.05.2025).

12. Android SDK: як налаштувати і почати розробку додатків. *Foxminded*. URL: https://foxminded.ua/android-sdk/ (дата звернення: 10.05.2025).

13. Meet Android Studio. *Android Developers*. URL: https://developer.android.com/studio/intro (date of access: 10.05.2025).

14. Develop Android apps with Kotlin. *Android Developers*. URL: https://developer.android.com/kotlin/ (date of access: 10.05.2025).

15. Kotlin programming language. *Kotlinlang*. URL: https://kotlinlang.org/ (date of access: 10.05.2025).

16. Kotlin Overview. *Android Developers*. URL: https://developer.android.com/kotlin/overview (date of access: 10.05.2025). (date of access: 10.05.2025).

17. Kotlin Documentation. *Kotlinlang*. URL: https://kotlinlang.org/docs/home.html (date of access: 10.05.2025).

18. Kotlin Tutorial. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/kotlin-programming-language (date of access: 10.05.2025).

19. Kotlin Features to Boost Android Development *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/kotlin-features/ (date of access: 10.05.2025).

20. Artyom, Backend. Kotlin: Key Features, Advantages, and Use Cases. *Plavno Blog*. URL: https://plavno.io/blog/kotlin-history (date of access: 10.05.2025).

21. Ezra Kanake. Clean Architecture with Kotlin. *Baeldung*. URL: https://www.baeldung.com/kotlin/clean-architecture-pattern (date of access: 10.05.2025).

22. codezup. Kotlin in Android with Clean Architecture: Example Project. *CodeZup*. URL: https://codezup.com/kotlin-android-clean-architecture-example-project/ (date of access: 10.05.2025).

23. codezup. Implementing Clean Architecture in Kotlin Android: A Guide. *CodeZup*. URL: https://codezup.com/clean-architecture-in-kotlin-android-guide/ (date of access: 10.05.2025).

24. Maps Compose Library | Maps SDK for Android. *Google Developers*. URL: https://developers.google.com/maps/documentation/android-sdk/maps-compose (date of access: 10.05.2025).

25. Retrofit with Kotlin Coroutine in Android. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/retrofit-with-kotlin-coroutine-in-android/ (date of access: 10.05.2025).

ДОДАТОК А

Інтеграція картографічних даних у програмне забезпечення за допомогою

Google Maps API

Реалізація логіки отримання оптимального маршруту до точки

УКР. НТУУ “КПІ імені Ігоря Сікорського”

Аркушів 2

Київ — 2025

Програмні засоби:

— мова програмування — Kotlin;

— бібліотека для http-запитів Retrofit.

```
interface GoogleDirectionsAPI {
    @GET("/maps/api/directions/json")
    suspend fun getPath(
        @Query("origin") start: String,
        @Query("destination") end: String,
        @Query("key") key: String,
    ): Response<DirectionsResponse>
}data class DirectionsResponse(
    val routes: List<Route>
)

data class Route(
    @SerializedName("overview_polyline")
    val overviewPolyline: OverviewPolyline
)

data class OverviewPolyline(
    val points: String
)
class RetrofitClient {
    private val googleDirectionsBase = "https://maps.googleapis.com/"

     fun googleDirectionApi(): GoogleDirectionsAPI {
         val logging = HttpLoggingInterceptor().apply {
             level = HttpLoggingInterceptor.Level.BODY
        }

        val client = OkHttpClient.Builder()
            .addInterceptor(logging)
            .build()

        return Retrofit.Builder()
            .baseUrl(googleDirectionsBase)
            .addConverterFactory(GsonConverterFactory.create())
            .client(client)
            .build()
            .create(GoogleDirectionsAPI::class.java)
    }
}
class MapApiImpl(
    private val api: GoogleDirectionsAPI,
    private val key: String
) : MapApi {
    override suspend fun getPathRoute(start: Coordinates, end: Coordinates): List<Coordinates> {
        val response = api.getPath(
            start = "${start.lat},${start.lng}",
            end = "${end.lat},${end.lng}",
            key = key
        )

        return if (response.isSuccessful) {
            val encodedPolyline = response.body()?.routes?.firstOrNull()?.overviewPolyline?.points
            val listLatLng: List<LatLng> = encodedPolyline?.let {
                PolyUtil.decode(it)
            } ?: emptyList()

            listLatLng.map { Coordinates(it.latitude, it.longitude) }
        } else {
            Log.e(
                "MapApiImpl",
                "getPathRoute() error. start=${start.lat},${start.lng}, end=${end.lat},${end.lng}. ${response.errorBody()}"
            )
            emptyList()
        }
    }
}
```