ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка мобільного додатка-гіду для Android на основі Kotlin з використанням Firebase та Google Maps API»

на здобуття освітнього ступеня бакалавра
зі спеціальності 126 Інформаційні системи та технології
(код, найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____	Богдан СКРИПНІК
(підпис)	*Ім'я, ПРІЗВИЩЕ здобувача*

Виконав:	здобувач вищої освіти гр. ІСД- 41
	Богдан СКРИПНІК
	Ім'я, ПРІЗВИЩЕ
Керівник:	*PhD,* Віра МИКОЛАЙЧУК
	Ім'я, ПРІЗВИЩЕ
науковий ступінь, вчене звання	
Рецензент:	_____
	Ім'я, ПРІЗВИЩЕ
науковий ступінь, вчене звання	

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення автоматизованих систем

Ступінь вищої освіти бакалавр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІПЗАС
_____Каміла СТОРЧАК
«____» _____2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Скрипнік Богдан Іванович

(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Розробка мобільного додатка-гіду для Android на основі Kotlin з використанням Firebase та Google Maps API

керівник кваліфікаційної роботи Віра МИКОЛАЙЧУК, PhD,

затверджені наказом Державного університету інформаційно-комунікаційних технологій
від «27» лютого 2024 р. № 36

2. Строк подання кваліфікаційної роботи «31» травня 2024р.

3. Вихідні дані до кваліфікаційної роботи:
   1. Нормативні матеріали
   2. Наукова-технічна література по Android та екосистемі мобільних додатків
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Архітектура додатка.
   2. Функціональні можливості додатка.
   3. Програмування додатку на основі Kotlin
   4. Дизайн і користувацький інтерфейс.
   5. Перелік ілюстративного матеріалу: *презентація*
5. Ілюстративний матеріал: *презентація*

6. Дата видачі завдання: «27» лютого 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	27.02.2024-05.03.2024	
2	Обгрунтування актуальності роботи	06.03.2024-11.03.2024	
3	Аналіз мобільних архітектур	12.03.2024-27.03.2024	
4	Вибір технологій та середовища проектування	28.03.2024-10.04.2024	
5	Розробка інформаційної системи	11.04.2024-15.05.2024	
6	Вступ, висновки, реферат	12.05.2024-15.05.2024	
7	Розробка демонстраційних матеріалів	16.05.2024-22.05.2024	
8	Попередній захист роботи	30.05.2024	

Здобувач вищої освіти _____ Богдан СКРИПНІК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Віра МИКОЛАЙЧУК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 77 стор., 31 рис., 1 табл., 7 джерел.

Мета роботи - створення функціонального, зручного та безпечного мобільного додатку на основі Kotlin з використанням Firebase та Google Maps Api. Котрий зможе задовольнити потреби користувачів під час планування та здійснення поїздок і дати їм можливість максимально використовувати мобільні технології.

Об'єкт дослідження -.мобільний додаток на основі Kotlin з використанням Firebase.

Предмет дослідження - предметом дослідження є процес розробки та впровадження мобільного додатку гіду на базі Kotlin для Android з використанням Firebase та Google Maps API. Цей об'єкт включає всі аспекти, пов'язані з проектуванням, програмуванням, тестуванням, розгортанням і підтримкою програми, а також взаємодією з користувачами та забезпеченням ефективної роботи програми. Таким чином, сфера дослідження охоплює всі аспекти процесу створення та експлуатації мобільного додатку-путівника на базі платформи Android з використанням Kotlin, Firebase та Google Maps API.

Короткий зміст роботи -. Дипломна робота присвячена розробці мобільного додатку на основі Kotlin та використанням Firebase та Google Maps API.

КЛЮЧОВІ СЛОВА: KOTLIN, GOOGLE MAPS API, FIREBASE

ABSTRACT

The text part of the qualifying work for obtaining a bachelor's degree: 77 pages, 43 figures, 1 table, 7 sources.

The purpose of the work is to create a functional, convenient and secure mobile application based on Kotlin using Firebase and Google Maps Api. Which will be able to meet the needs of users when planning and making trips and give them the opportunity to make the most of mobile technologies.

Object of research is a mobile application based on Kotlin using Firebase.

Subject of research the subject of the study is the process of developing and implementing a Kotlin-based mobile guide application for Android using Firebase and Google Maps API. This entity includes all aspects related to the design, programming, testing, deployment, and maintenance of the application, as well as interacting with users and ensuring that the application works efficiently. Thus, the scope of research covers all aspects of the process of creating and operating a mobile guide application based on the Android platform using Kotlin, Firebase and Google Maps API.

Summary of the work: The thesis is devoted to the development of a mobile application based on Kotlin and the use of Firebase and Google Maps API.

KEYWORDS: KOTLIN, GOOGLE MAPS API, FIREBASE

ЗМІСТ

ВСТУП

Мобільні додатки стали невід'ємною частиною сучасного життя, пропонуючи користувачам широкий спектр можливостей для планування, спілкування та розваг. Особливо популярні мобільні додатки-путівники, які допомагають планувати екскурсії, відвідувати цікаві місця та отримувати інформацію про околиці. У цьому контексті розробка мобільних додатків-гідів для платформи Android, що відповідають потребам сучасних користувачів, стала актуальною задачею.

Метою роботи є розробка та реалізація мобільного додатку путівника для Android на основі мови програмування Kotlin з використанням API-сервісів Firebase та Google Maps. Метою цієї програми є надання користувачам практичних інструментів для планування маршрутів, пошуку визначних місць і навігації в реальному часі.

Об'єкт дослідження - мобільний додаток на основі Kotlin з використанням Firebase.

*Предмет дослідженн*я - предметом дослідження є процес розробки та впровадження мобільного додатку гіду на базі Kotlin для Android з використанням Firebase та Google Maps API. Цей об'єкт включає всі аспекти, пов'язані з проектуванням, програмуванням, тестуванням, розгортанням і підтримкою програми, а також взаємодією з користувачами та забезпеченням ефективної роботи програми. Таким чином, сфера дослідження охоплює всі аспекти процесу створення та експлуатації мобільного додатку-путівника на базі платформи Android з використанням Kotlin, Firebase та Google Maps API.

Для досягнення мети ця робота включає детальний аналіз літературних джерел з розробки мобільних додатків, вивчення особливостей мови програмування Kotlin, а також вивчення функцій і функцій Firebase і Google Maps API. Крім того, ця робота включає проектування архітектури додатків, функціональну реалізацію та тестування.

Практичне застосування знань і навичок, набутих у процесі розробки додатків, є дуже важливим, оскільки це сприяє подальшому вдосконаленню мобільних технологій і покращенню взаємодії з користувачем.

Завдання дослідження:

1. Вивчити особливості мови програмування Kotlin та визначити її переваги у розробці мобільних додатків для платформи Android.
2. Ознайомитися з можливостями та функціоналом сервісу Firebase, зокрема засобами збереження даних та аутентифікації користувачів, та визначити їх придатність для використання у розробці додатка-гіда.
3. Ознайомитися з можливостями та функціоналом сервісу Firebase, зокрема засобами збереження даних та аутентифікації користувачів, та визначити їх придатність для використання у розробці додатка-гіда.
4. Дослідити можливості та функціонал API Google Maps для додавання картографічних сервісів у мобільний додаток.
5. Спроектувати архітектуру додатка-гіда, обрати оптимальний підхід до розробки та визначити структуру додатка.
6. Реалізувати необхідний функціонал додатка-гіда з використанням Kotlin, Firebase та Google Maps API.
7. Провести тестування розробленого додатка для перевірки його функціональності та стабільності.

1 АНАЛІЗ АРХІТЕКТУРИ МОБІЛЬНИХ ДОДАТКІВ

1.1 Огляд існуючих архітектурних підходів.

У світі швидкого розвитку мобільних технологій, архітектурні стратегії стають критичним фактором для успішної розробки мобільних додатків. В даному дослідженні ми ретельно аналізуємо різноманітні підходи, які використовуються в мобільній розробці, зокрема, розглядаємо їх особливості, переваги та недоліки.

1. Архітектура "Клієнт-сервер"

Одним із найпоширеніших підходів є архітектурна модель "Клієнт-Сервер". Ця стратегія розділяє функції на клієнтській і серверній стороні програми. Клієнтська частина відповідає за відображення інтерфейсу та взаємодію з користувачем, тоді як серверна частина відповідає за обробку бізнес-логіки та зберігання даних.

Особливості: функції розділяються на дві частини: клієнт та сервер. Клієнт відповідає за показ і взаємодію з користувачем, а сервер за обробку логіки та збереження даних.

Переваги: ця архітектурна структура забезпечує централізоване зберігання та керування даними, спрощуючи технічне обслуговування та вдосконалення системи. Крім того, він забезпечує ефективне керування безпекою та контрольований доступ до даних.

Недоліки: у ситуаціях з обмеженим доступом до Інтернету використання серверів може призвести до проблем із продуктивністю та доступністю. Крім того, централізований підхід до керування може зробити систему вразливою до збоїв сервера.

2. Модульна архітектура

Інший поширений підхід - це модульна архітектура, яка базується на розділенні функціональності на невеликі, незалежні модулі. Кожен модуль

відповідає за певний аспект функціональності або набір пов'язаних функцій. Цей підхід сприяє полегшенню тестування, розширенню та підтримці додатку.

Особливості: концепцією цього підходу є обертання навколо розбиття функціональних можливостей на окремі самодостатні модулі, кожен з яких присвячений певному аспекту або групі функцій.

Переваги: легше тестування, розширення та підтримка програми.

Недоліки: у складних системах керування залежностями між модулями та подальше збільшення складності коду може викликати труднощі.

3. MVC (Model-View-Controller) та MVVM (Model-View-ViewModel)

MVC та MVVM - це популярні архітектурні шаблони, що використовуються у мобільній розробці. Вони дозволяють розділити функціональність додатку на моделі (дані), види (представлення) та контролери (логіка). MVVM додає до цієї моделі ще один компонент - модель представлення, яка дозволяє розділити бізнес-логіку від інтерфейсу користувача.

Особливості: Обидва підходи дозволяють розділити функціональність додатку на компоненти, що спрощує розробку та підтримку. MVC розділяє функціональність на моделі (дані), види (представлення) та контролери (логіка), тоді як MVVM додає модель представлення.

Переваги: Обидва підходи полегшують тестування та розширення додатку, а також дозволяють забезпечити чистоту та структурованість коду.

Недоліки: У складних додатках може знадобитися додаткове керування станом та залежностями між компонентами, що може призвести до складнощів у розробці та розумінні.

4. Архітектурні патерни

Архітектурні патерни, такі як Singleton, Factory, Observer, є важливими засобами у розробці мобільних додатків. Вони надають загальні рекомендації щодо організації коду та сприяють його покращенню та розширенню.

Особливості: Архітектурні патерни, такі як Singleton, Factory, Observer, є загальноприйнятими рішеннями для типових проблем у розробці програмного

забезпечення. Вони надають шаблони для організації коду та сприяють його покращенню та розширенню.

Переваги: Використання архітектурних патернів допомагає створити більш стабільні та простіші у розумінні додатки. Вони забезпечують ефективне використання ресурсів та покращують читабельність коду.

Недоліки: У деяких випадках використання паттернів може призвести до зайвої складності або надмірного використання пам'яті. Також не завжди можна знайти ідеальний паттерн для конкретної задачі, що може призвести до компромісів у якості коду.

1.1.1 Порівняння архітектурних підходів

У світі швидкої технічного розвитку, мобільні програми грають важливу роль у спілкуванні, розвагах, торгівлі та багатьох інших частинах нашого щоденного життя. Кожен день бачимо, що все більше і більше людей користуються різними мобільними платформами. Це означає, що розробникам потрібно не лише створити корисні програми, а й розробити ефективні способи будування цих програм.

Важливі стратегії для успішної роботи над мобільними програмами. Вибір найкращого підходу залежить від потреб проекту, його розміру, вимог до продуктивності та швидкості розробки. Ми дослідимо й порівняємо кілька ключових підходів, таких як "Клієнт-Сервер", модульна архітектура, а також популярні архітектурні шаблони, наприклад MVC та MVVM. Це допоможе нам краще розібратися у перевагах та недоліках кожного з них і обрати найбільш підходящий для певних випадків розробки мобільних програм.

Порівнюючи підходи, можна визначити, який з них найкраще підходить для конкретного проекту з урахуванням його потреб, масштабу та специфіки.

Архітектура "Клієнт-Сервер":

Переваги - Простіше підтримувати роботу системи та вносити зміни, коли всі дані та логіка знаходяться в одному місці. Захищає речі та дозволяє легко отримувати доступ до даних.

Недоліки - Наявність сервера може спричинити проблеми зі швидкістю та надійністю вашого веб-сайту. Коли все контролюється з одного місця, вся система може легко зламатися, якщо сервер вийде з ладу

Модульна архітектура:

Переваги - «Це як будівництво з кубиків Lego». Цікава річ у наявності незалежних модулів полягає в тому, що тестувати та розробляти програму легко. Ви можете створити кожну частину програми окремо, що полегшить і пришвидшить її створення.
Недоліки - Недоліком складних систем є те, що їм важко керувати залежностями між модулями, що може ускладнити код

MVVM:

Переваги - Дозволяє зберігати бізнес-логіку та інтерфейс користувача окремо, що полегшує тестування та виправлення коду.

Простіше кажучи, використання прив’язок даних і команд дозволяє пояснити, як взаємодіють модель і представлення, без необхідності писати багато коду, що полегшує створення та виправлення коду.

Реактивне програмування: ви можете використовувати відстеження змін у ViewModel для автоматичного оновлення інтерфейсу користувача, коли змінюються дані..

Недоліки - для новачків MVVM може бути справжньою проблемою, щоб обгорнути голову, оскільки є дуже багато частин, і те, як вони всі працюють разом, може бути трохи заплутаним. Також, іноді нам потрібно використовувати додаткові бібліотеки чи фреймворки для виконання деяких речей MVVM, і це може ускладнити їх проектування.

MVC:

Переваги - MVC поділяє функції програми на три частини: моделі, представлення та контролери. Простота тестування: компоненти можна тестувати окремо, що спрощує процес виявлення та вирішення проблем». Крім того, наявність чітко визначених ролей для кожного компонента дає змогу змінювати та розширювати програму, не впливаючи на інші компоненти.

Недоліки - Складні додатки можуть мати надлишкову взаємодію між компонентами, через що код важко зрозуміти та не відставати від нього. Крім того, відстеження стану програми може бути складним у складних або великомасштабних програмах

Розділення відповідальності:

MVC: Модель відповідає за дані та бізнес-логіку.Вид представляє інтерфейс користувача. Контролер обробляє взаємодію користувача та моделі.

MVVM: Модель відповідає за дані. Вид відображає інтерфейс та реагує на дії користувача. Модель представлення відокремлює бізнес-логіку від інтерфейсу та забезпечує зв'язок між видом та моделлю.

Модульна архітектура: Розділення функціональності на невеликі, незалежні модулі, кожен з яких відповідає за своє обмежене завдання.

"Клієнт-сервер": У даній архітектурі сервер відповідає за обробку запитів від користувачів, виконання бізнес-логіки та збереження даних. Клієнтська сторона взяла на себе завдання виведення інтерфейсу та спілкування з користувачем.

Архітектурні патерни: Запропонований розподіл обов'язків між різними компонентами програми забезпечує чіткий розподіл завдань у сферах бізнес-логіки, презентації та керування даними.

Чистота та підтримка коду:

MVC та MVVM: Обидва підходи дозволяють відділити бізнес-логіку від інтерфейсу користувача, що полегшує тестування та підтримку.

Модульна архітектура: Незалежні модулі спрощують тестування та підтримку, оскільки кожен може розвиватись окремо.

"Клієнт-сервер": Такий підхід дозволяє зберегти багато коду для бізнес-логіки та даних на серверному боці, що робить підтримку та управління кодом простими. Проте, якщо функціональність додатка росте, можуть виникати проблеми із підтримкою обсягового коду на стороні сервера.

Архітектурні патерни: Надання стандартизованих шаблонів розробки програмного забезпечення допомагає підтримувати та читати код.

Тестування:

MVC та MVVM: Дозволяють легко тестувати бізнес-логіку та інтерфейс користувача окремо.

Модульна архітектура: Легке тестування окремих модулів спрощує виявлення помилок та покращує якість програми.

“Клієнт-сервер”: Тестування може бути складним у цьому підході поряд із іншими архітектурними методами, оскільки потребує перевірки як клієнтської, так і серверної частин.

*Архітектурні патерни**:*** Завдяки структурованості та чіткому розподілу обов’язків вони часто полегшують тестування, що полегшує розробку, керовану тестуванням.

Гнучкість та розширюваність:

*MVC та MVVM**:*** Забезпечують гнучкість у внесенні змін та розширенні додатку.

Модульна архітектура: Дозволяє легко додавати нові функції та модулі без необхідності змінювати весь додаток.

“Клієнт-сервер”: Ця архітектура надає гнучкості та легкості розширення, оскільки окремі компоненти можуть бути модифіковані або замінені без впливу на інші частини системи. Наприклад, можна легко поширити функціональність серверної частини без необхідності змінювати клієнтську сторону й навпаки.

Архітектурні патерни: Вони забезпечують гнучкість і розширюваність за допомогою стандартизованих шаблонів, які дозволяють додавати нові функції або змінювати існуючі без необхідності повторювати всю програму.

Ситуації для використання "Клієнт-Сервер", Модульної архітектури, MVC та MVVM :

З практичної точки зору, ці архітектури – мають зовсім різні підходи до розробки програмного забезпечення. Вони мають свої переваги та властивості додаткових використань, розглянемо ситуації, коли краще використовувати кожен з цих підходів:

Архітектура "Клієнт-Сервер":

Ситуації з централізованою обробкою даних: якщо вашому додатку потрібна централізована обробка даних або великі обсяги обчислень, архітектура Клієнт-Сервер виявиться ефективнішою. Сервер може бути масштабованим та оптимізованим, що забезпечить високу обробку запитів від клієнтів. Додатки, яким потрібно постійний доступ до серверу: у випадку, якщо ваш додаток потребує постійного доступу до серверу для отримання оновлень даних або синхронізації інформації між різними клієнтами, архітектура Клієнт-Сервер також стане кращим вибором.

Модульна архітектура:

Випадки, коли є необхідність адаптивності і масштабованості: модульна конструкція може бути корисною у випадках, коли необхідно збільшити гнучкість або змінити характеристики. Кожен модуль може бути створений та протестований окремо, що спрощує підтримку миттєвого розгортання.

Програми на місці з обмеженим підключенням до сервера: якщо програма має обмежений доступ до сервера або діє в основному в автономному режимі, то модульна конструкція може бути доречнішою. Кожна частина може працювати незалежно у локальному середовищі програми.

MVC (Model-View-Controller):

Складні веб-додатки: MVC може бути вигідним вибором у великих веб-додатках, де бізнес-логіка, відображення та керування станом програми мають бути чітко розділені. Це дозволяє підтримувати код у чистоті та легко розширювати програму.

Програми з великою кількістю взаємодій: у програмах із великою кількістю взаємодій між різними елементами інтерфейсу користувача MVC може допомогти зберегти ці взаємодії розділеними та організованими.

MVVM (Model-View-ViewModel):

Мобільні програми: для мобільних програм, які вимагають швидкості розробки та простоти тестування, MVVM може бути більш практичним вибором.

Декларативний підхід дозволяє легко прив’язувати дані та представлення, не вимагаючи багато додаткового коду.

Проекти, які використовують багато логіки інтерфейсу користувача: для програм, які використовують багато даних і логіки інтерфейсу користувача, MVVM може спростити розробку та обслуговування, відокремивши бізнес-логіку від презентації.

1.1.2 Тенденції в архітектурі мобільних додатків

За останні роки в області архітектури мобільних додатків було помічено кілька важливих тенденцій:

Розподілена архітектура: Архітектурні підходи, засновані на розподіленій моделі, стають популярними. Це дозволяє розміщувати бізнес-логіку та функціональні можливості на кількох рівнях, наприклад на стороні клієнта, на стороні сервера та хмарних службах, забезпечуючи більшу масштабованість і гнучкість.

Архітектура мікросервісів: використання архітектури мікросервісів стає все більш популярним у мобільній розробці. Такий підхід дозволяє вам розділити вашу програму на невеликі автономні служби, які працюють незалежно одна від одної, сприяючи підвищеній гнучкості та швидкості розробки.

Хмарні рішення: використання хмарних служб для зберігання даних, функцій обробки та керування інфраструктурою стає все більш популярним. Це дозволяє зосередитися на розробці функціональних можливостей програми, усуваючи проблеми з інфраструктурою на стороні хмарного постачальника.

Чуйне програмне забезпечення: використання реактивних принципів у розробці мобільних додатків стає все більш популярним. Це дозволяє створювати додатки, які реагують на події в реальному часі та автоматично адаптуються до змін середовища.

Інтеграція штучного інтелекту (AI) і машинного навчання (ML): розгортання технології штучного інтелекту та машинного навчання в мобільних

додатках дозволяє створювати більш інтелектуальні та персоналізовані рішення для використання.

Безпека: Все більше уваги приділяється безпеці мобільних додатків, включаючи захист особистих даних користувачів за допомогою засобів шифрування та автентифікації.

Усі ці тенденції відображають постійну еволюцію та зростання складності мобільних додатків, а також важливість використання сучасних технологій і архітектурних методів для розробки та успішної експлуатації цих додатків.

Тенденції що стосуються "Клієнт-Сервер", модульної архітектури, MVC та MVVM:

У сучасному світі архітектурні підходи до мобільної розробки постійно змінюються, відображаючи потреби ринку та технологічні досягнення. Давайте краще розглянемо ці зміни і тенденції з точки зору людей, які стоять за їх розвитком:

"Клієнт-Сервер":

Ми бачимо, що тенденція до мікросервісної архітектури допомагає розробникам розділити функціональність додатків на більш маневрені та самодостатні сервіси, що сприяє покращенню якості та швидкості розробки.

Модульна архітектура:

Нова епоха компонентної розробки дозволяє розробникам створювати "кубики Lego" для додатків, що значно спрощує їх розробку та підтримку.

MVC (Model-View-Controller):

Нові можливості інтерактивності та реактивності дають користувачам більше можливостей для взаємодії з додатками в реальному часі, зберігаючи при цьому легкість розробки та підтримки.

MVVM (Model-View-ViewModel):

Ми продовжуємо бачити зростання популярності MVVM серед розробників, які оцінюють його зручність та швидкість розробки.

Ці тенденції відображають постійну динаміку та розвиток у світі мобільної розробки, де кожен змінюється, забезпечуючи кращі інструменти та підходи для створення додатків, які відповідають потребам сучасного користувача.\

1.2 Вплив архітектурних рішень на продуктивність та швидкість додатків

Під час розробки мобільних додатків, вибір архітектури грає велику роль у визначенні продуктивності та швидкості роботи додатків. У цьому розділі ми розглянемо вплив різних архітектурних рішень на ефективність розробки та функціональність додатків.

Розподілена архітектура

Подібно до спільного будівництва міста, де кожна частина має своє призначення, розподілена архітектура розділяє функціонал на різні частини, які можуть працювати незалежно одна від одної. Це забезпечує більшу гнучкість та масштабованість, але може збільшити складність взаємодії між компонентами.

Компанія Netflix використовує розподілену архітектуру у своєму мобільному додатку. Розміщення бізнес-логіки на сервері дозволяє їм забезпечити масштабованість та швидкість навіть при великому обсязі даних.

Переваги: розподілена архітектура дозволяє розміщувати бізнес-логіку та функціональність на різних рівнях, сприяючи більшій гнучкості та масштабованості. Наприклад, це дозволяє таким компаніям, як Netflix, забезпечувати стабільну роботу додатків навіть за великих обсягів даних і високих навантажень.

Недоліки: Передача даних між різними складовими архітектури може займати час і споживати ресурси мережі, особливо при великому обсязі даних. Крім того, вище зазначена складність взаємодії може призвести до затримок у відповіді додатку на дії користувачів.

Модульна архітектура

Модульна архітектура, подібна до LEGO, розділяє програму на невеликі незалежні модулі.

Pinterest - приклад компанії, яка успішно використовує модульну архітектуру у своєму додатку. Це дозволяє їм швидко розвивати та тестувати нові функції, не впливаючи на існуючий функціонал.

Переваги: Модульна архітектура полегшує швидку розробку та тестування нових функцій, оскільки додаток розділено на невеликі незалежні модулі. Наприклад, Pinterest використовує модульну архітектуру, щоб швидко розгортати нові ідеї та функції, не впливаючи на ефективність старих функцій.

Недоліки: Збільшення кількості модулів може призвести до збільшення часу завантаження додатку або сторінок, особливо якщо кожен модуль має великий обсяг коду або потребує великої кількості ресурсів.

MVC та MVVM

Архітектурні шаблони MVC та MVVM впорядковують функціонал додатку на моделі, види та контролери або моделі, представлення та моделі представлення відповідно. Це допомагає зробити код більш організованим та легше зрозумілим, але може вимагати більше зусиль для реалізації.

Компанія Spotify використовує шаблони архітектури MVC та MVVM у своєму додатку. Це дозволяє їм розміщувати логіку додатку у відповідних компонентах, забезпечуючи чистоту коду та зручність для розробників.

Переваги: Архітектурні шаблони MVC і MVVM дозволяють окремо організувати логіку програми та представлення даних, що полегшує розуміння та підтримку коду. Наприклад, Spotify використовує ці шаблони, щоб переконатися, що код є чистим і простим у розробці.

Недоліки: За деяких умов, особливо при неправильному використанні, перенесення логіки бізнес-процесів на клієнтську сторону (вид) може призвести до збільшення обсягу даних, які потрібно передавати через мережу, що може вплинути на швидкість реакції додатку та його продуктивність.

Архітектура "Клієнт-Сервер"

Архітектура "Клієнт-Сервер" розділяє додаток на клієнтську та серверну частини. Це дозволяє ефективно керувати доступом до даних та зберігати бізнес-логіку на сервері, що полегшує підтримку та розвиток додатку. Однак недоліком може бути залежність від доступності сервера та можливість виникнення проблем з швидкістю при обміні даними.

Компанія Uber використовує архітектуру "Клієнт-Сервер" у своєму додатку. Це дозволяє їм ефективно керувати доступом до даних та забезпечити стабільну роботу додатку навіть при високому навантаженні.

Переваги: Розділення програми на клієнтську та серверну частини дозволяє ефективно керувати доступом до даних і забезпечує стабільну роботу програми. Наприклад, Uber використовує цю архітектуру, щоб забезпечити стабільну роботу свого додатка навіть під великим навантаженням.

Недоліки: Якщо серверне обладнання або програмне забезпечення не здатне ефективно обробляти запити великої кількості користувачів, це може призвести до затримок у відповіді додатку та зниження продуктивності.

Вибір архітектурного рішення має великий вплив на продуктивність та швидкість мобільних додатків. Кожен підхід має свої переваги та недоліки, і важливо враховувати їх при розробці додатку, щоб досягти оптимального балансу між продуктивністю та функціональністю.

1.2.1 Вплив архітектурних рішень на оптимізацію додатків

Архітектурні рішення мають значний вплив на оптимізацію мобільних додатків. Тут ми розглянемо деякі аспекти цього впливу:

Продуктивність і швидкість:

Архітектура клієнт-сервер: ця архітектура може збільшити швидкість програм, оскільки бізнес-логіка та обробка даних відбуваються на сервері, а не на пристрої користувача. Це може зменшити навантаження на стороні клієнта, тим самим покращуючи реакцію програми та реакцію на дії користувача.

MVC і MVVM: ці шаблони дозволяють ефективно розділяти бізнес-логіку та дисплей, що може полегшити реагування на дії користувача та покращити загальну продуктивність програми.

Модульна архітектура: модульна архітектура допомагає оптимізувати швидкість завантаження програми, оскільки дозволяє завантажувати лише необхідні модулі відповідно до потреб користувача.

Розподілена архітектура: за допомогою розподіленої архітектури ви можете розмістити бізнес-логіку на сервері, що дозволяє оптимізувати час відповіді програми та спрощує масштабування.

Ефективно використовуйте ресурси:

Всі ці архітектурні рішення. Всі підходи забезпечують ефективне використання ресурсів, розподіл завдань та оптимізацію продуктивності додатків.

Легке тестування та підтримка:

MVC і MVVM: ці архітектурні шаблони забезпечують чіткий розподіл між компонентами, полегшуючи тестування окремих частин програми та підтримку коду.

Модульна архітектура: модульна архітектура також полегшує тестування та обслуговування, оскільки окремі модулі можна тестувати незалежно один від одного.

Розподілена архітектура: за допомогою розподіленої архітектури ви можете легко перевірити бізнес-логіку на сервері та відокремити її від інтерфейсу користувача.

Можливість розширення:

Всі архітектурні рішення. Завдяки потенціалу масштабованості всі ці архітектурні підходи можна налаштувати для оптимальної роботи в середовищах із великою кількістю користувачів і даних.

Враховуючи ці аспекти, важливо проаналізувати вимоги та характеристики конкретного проекту, щоб вибрати оптимальну архітектуру, яка забезпечує оптимізацію програми та відповідає потребам користувачів.

Оптимізація мобільних додатків відбувається за допомогою різних архітектурних рішень та підходів. Розглянемо способи їх оптимізації:

Розподілена архітектура:

Оптимізація використання ресурсів: розміщення бізнес-логіки на сервері дозволяє ефективно використовувати його ресурси, тим самим зменшуючи навантаження на пристрої користувачів.

Кешування даних: використання кешу на рівнях клієнта та сервера дозволяє зберігати часто використовувані дані та запити, тим самим зменшуючи час завантаження даних і покращуючи швидкість реагування програми.

Модульна архітектура:

Оптимізація завантажень: Розбиття програми на невеликі модулі дозволяє завантажувати лише ті частини, які потрібні користувачеві, тим самим зменшуючи час завантаження та споживання ресурсів.

Оптимізація тестування: легке тестування окремих модулів дозволяє швидше виявляти та виправляти помилки, тим самим покращуючи загальну якість програми.

MVC і MVVM:

Розподіл відповідальності: Чіткий розподіл між бізнес-логікою та презентацією дозволяє оптимізувати роботу кожного компонента окремо, тим самим підвищуючи швидкість і продуктивність програми.

Кешування даних: використання кешування на рівні моделі та перегляду може значно скоротити час завантаження та обробки даних.

Архітектура клієнт-сервер:

Оптимізуйте мережеву взаємодію. Ефективно керуйте передачею даних між клієнтами та серверами, наприклад, використовуючи стиснення даних і кешування, що зменшує час оновлення та збільшує швидкість програми.

Розподілена обробка: розміщення бізнес-логіки на сервері оптимізує час відповіді програми та зменшує навантаження на пристрої користувачів.

1.3 Вплив архітектурних рішень на витрати ресурсів пристрою

Архітектурні рішення мають значний вплив на споживання ресурсів пристрою, таких як ЦП, пам’ять і потужність. Розглянемо, як кожне архітектурне рішення може вплинути на ці ресурси:

Розподілена архітектура:

Вартість процесорного часу: може призвести до збільшення витрат процесорного часу на обробку віддалених запитів та спілкування з серверами.

Переваги:

Гнучкість і масштабованість: можливість розміщувати функціональність на різних рівнях, збільшуючи масштаб і гнучкість системи.

Висока доступність: завдяки розподіленому характеру архітектури можливість забезпечити безперервну роботу програми.

Недоліки:

Збільшений час процесора: додатковий час, необхідний для обробки запитів і віддаленої взаємодії з сервером, може збільшити споживання ресурсів пристрою

Модульна архітектура:

Накладні витрати пам’яті: розділення програми на окремі модулі може збільшити накладні витрати пам’яті для завантаження та зберігання даних з кожного окремого модуля.

Затрати ЦП: перемикання між модулями та взаємодія між ними може призвести до додаткових витрат ЦП.

Переваги:

Простота тестування та обслуговування. Розбиття функціональних можливостей на невеликі незалежні модулі спрощує тестування та обслуговування програм.

Модульність: можливість розробки та розгортання окремих модулів незалежно один від одного.

Недоліки:

Накладні витрати пам'яті: поділ програми на окремі модулі може збільшити накладні витрати пам'яті на завантаження та зберігання даних кожного окремого модуля.

Навантаження на ЦП: перемикання між модулями та взаємодія з ними може спричинити додаткові навантаження на ЦП..

MVC і MVVM:

Вартість пам'яті: розділення функціональних можливостей програми на моделі, подання та контролери (або моделі, подання та моделі подання у випадку MVVM) може збільшити витрати на пам'ять для зберігання даних і подання.

Використання часу ЦП: розділення функцій на різні компоненти може збільшити час ЦП, який витрачається на взаємодію один з одним і обробку даних.

Переваги:

Організація та чистота коду: розділення функціональних можливостей програми на моделі, подання та контролери (або моделі, подання та моделі подання у випадку MVVM) робить код більш організованим і легшим для розуміння.

Краща підтримка: цей підхід покращує підтримку коду та полегшує життя розробникам.

Недоліки:

Витрати на пам'ять: розділення функцій програми на моделі, подання та контролери може збільшити витрати на пам'ять для зберігання даних і подання.

Використання часу ЦП: Розбиття функцій на різні компоненти може збільшити час, необхідний для обробки даних і взаємодії одна з одною.

Архітектура клієнт-сервер:

Використання мережевих ресурсів: взаємодія з сервером може збільшити мережеві ресурси та енергоспоживання, особливо якщо безперервно взаємодіяти з сервером для оновлення даних і виконання запитів.

Переваги:

Знижене навантаження на пристрій: більшість обчислень і обробки даних виконуються на сервері, що допомагає зменшити навантаження на пристрій користувача.

Швидка реакція на зміни: Оновлення та вдосконалення можна робити централізовано на сервері, що дозволяє швидко реагувати на зміни та їх впровадження в програму.

Недоліки:

Залежність від мережі: необхідність постійного підключення до мережі для отримання даних може спричинити проблеми, особливо в умовах обмеженого або нестабільного підключення до мережі.

Збільшений мережевий трафік: часті взаємодії з сервером можуть спричинити збільшення мережевого трафіку, що може вплинути на швидкість програми та споживання даних.

Враховуючи ці фактори, розробники можуть вибрати архітектурне рішення, яке оптимізує використання ресурсів пристрою, забезпечуючи тим самим оптимальну продуктивність і ефективність програми.

1.3.1 Різниця впливу архітектурних рішень для різних видів пристроїв

Різні архітектурні рішення можуть по-різному впливати на споживання ресурсів пристрою залежно від його характеристик і типу. Ось як це може вплинути на різні типи пристроїв:

На мобільних пристроях, таких як смартфони та планшети:

Розподілена архітектура може збільшити час процесора, який витрачається на обробку віддалених запитів і зв’язок із серверами через мобільний Інтернет.

Модульна архітектура може збільшити витрати пам’яті для завантаження та зберігання даних окремих модулів, що може вплинути на продуктивність пристрою.

Архітектура клієнт-сервер може збільшити мережеві ресурси та енергоспоживання завдяки безперервній взаємодії з сервером для оновлення даних і виконання запитів.

На ПК:

Споживання ресурсів може бути менш помітним, оскільки ПК зазвичай мають більше ЦП і пам’яті, ніж мобільні пристрої. Однак ПК також може страждати від великих обсягів даних і надмірного споживання пам’яті, що може призвести до зниження продуктивності, особливо на слабших комп’ютерах.

У кожному конкретному випадку важливо ретельно враховувати характеристики пристрою і особливості архітектурного рішення, щоб мінімізувати споживання ресурсів і забезпечити оптимальну продуктивність.

1.4 Безпека архітектури мобільних додатків

Безпека архітектури мобільних додатків вкрай важлива, оскільки вони мають доступ до особистих даних користувачів та чутливої інформації. Ось деякі аспекти безпеки архітектури мобільних додатків:

Автентифікація та авторизація. Додатки повинні використовувати надійні методи автентифікації, наприклад двофакторну автентифікацію, і належним чином керувати рівнями доступу користувачів, щоб запобігти несанкціонованому доступу до даних.

Шифрування даних: дані, що передаються між програмою та сервером, а також дані, що зберігаються на пристрої користувача, мають бути зашифровані для захисту від перехоплення та несанкціонованого доступу.

Захист від вразливостей: розробникам слід уникати використання вразливих бібліотек і фреймворків і розглядати потенційні атаки, такі як перехоплення даних, впровадження коду та маніпуляції введенням.

Оновлення безпеки та моніторинг: програми необхідно регулярно оновлювати, щоб усунути виявлені вразливості та забезпечити безпеку. Крім того,

слід використовувати моніторинг безпеки для виявлення підозрілих дій і вразливостей.

Внутрішня безпека: якщо програма використовує бек-енд, вона повинна приділяти належну увагу безпеці цього сервера, включаючи застосування заходів для захисту даних і мережевого зв’язку. Відповідність правилам і нормам. Програми мають відповідати вимогам захисту даних, таким як GDPR у Європейському Союзі або HIPAA у Сполучених Штатах.

Забезпечення безпеки мобільних додатків – це безперервний процес, який вимагає пильної уваги на всіх етапах розробки, від проектування до випуску та підтримки.

2 ВИКОРИСТАННЯ МОВИ ПРОГРАМУАННЯ KOTLIN У РОЗРОБЦІ МОБІЛЬНИХ ДОДАТКІВ

2.1 Мова програмування Kotlin для IoT.

Kotlin стає все більш поширеною мовою програмування для розробки рішень в Інтернеті речей (IoT). Вона приваблива для цього напряму кількома особливостями.

Переваги використання Kotlin для IoT:

Сумісність з Java: Kotlin повністю сумісний з Java, що дозволяє використовувати існуючі бібліотеки та фреймворки Java. Це особливо корисно в IoT, де багато пристроїв уже використовують Java. Наприклад, ви можете інтегрувати Kotlin в існуючі проекти Java IoT за допомогою бібліотек для роботи з датчиками або мережевими протоколами.

Лаконічність і безпека: Kotlin пропонує більш стислий синтаксис, ніж Java, що зменшує кількість коду та ризик помилок. Крім того, вбудовані механізми запобігання нульовим (небезпечним) покажчикам роблять код більш надійним. Це важливо в проектах IoT, де помилки можуть призвести до збою пристрою.

Підтримка між платформами: Kotlin Multiplatform дозволяє писати спільний код для різних платформ, включаючи Android, iOS, веб і сервери. Це означає, що може бути єдина кодова база для пристроїв IoT і мобільних додатків, які ними керують. Наприклад, ви можете розробити програму для смартфона, яка керуватиме розумним будинком, використовуючи той самий код для мобільної програми та мікроконтролера.

Продуктивність: Kotlin скомпільована з використанням байт-коду JVM, що забезпечує високу продуктивність на пристроях з обмеженими ресурсами. Крім того, Kotlin/Native дозволяє компілювати код безпосередньо в машинний код для виконання на платформах, які не є JVM. Наприклад, ви можете писати високопродуктивні програми для мікроконтролерів на мові C і керувати логікою на Kotlin, забезпечуючи оптимальне використання ресурсів.

Приклад використання Kotlin в IoT:

Розумний дім: використовуйте Kotlin для розробки додатків, які керують і автоматизують розумні домашні пристрої, такі як системи освітлення, системи безпеки та кондиціонери. Наприклад, на Kotlin можна написати програму, яка контролює освітлення та температуру через смартфон.

Промисловий IoT: Kotlin можна використовувати для розробки систем моніторингу та управління в промислових додатках, де надійність і продуктивність є важливими. Наприклад, система моніторингу виробничого обладнання збирає дані з датчиків і передає їх на сервер для аналізу.

Здоров'я та фітнес: розробка носіїв і додатків для відстеження здоров'я та фізичної активності. Наприклад, фітнес-трекери збирають дані про активність користувача та передають їх у мобільний додаток для подальшого аналізу.

2.2 Фреймворки, бібліотеки Kotlin для створення мобільного додатка

Kotlin Multiplatform Mobile (KMM)

Опис: KMM дозволяє використовувати одну базу коду для написання логіки, яка працює на Android та iOS. Це значно спрощує технічне обслуговування та оновлення додатків, оскільки однакова логіка може використовуватися на обох платформах.

Переваги:

Кодова база: кодова база для бізнес-логіки на обох платформах.

Проста інтеграція: інтеграція з існуючими проектами Android та iOS.

Modern Toolbox: підтримує такі IDE, як IntelliJ IDEA та Android Studio.

Приклад використання: створюйте програми, яким потрібно синхронізувати дані між платформами, наприклад мобільні клієнти для соціальних мереж або обміну повідомленнями.

Jetpack Compose

Опис: Jetpack Compose — це сучасний інструмент розробки інтерфейсу користувача Android, який дозволяє створювати інтерфейс користувача за допомогою декларативного підходу.

Переваги:

Простий і стислий: зменшує кількість коду, необхідного для створення інтерфейсу користувача.

Швидкість реагування: забезпечує автоматичне оновлення інтерфейсу користувача при зміні даних.

Інтеграція з Android Studio: повна підтримка в середовищах розробки.

Приклад використання: дизайн інтерфейсу для мобільних додатків, які потребують швидкої та легкої зміни інтерфейсу користувача, наприклад додатків для новин або соціальних платформ.

Ktor

Опис: Ktor — це платформа для створення асинхронних серверних і клієнтських програм, розроблена JetBrains. Його можна використовувати для створення серверних програм, які обслуговують мобільні програми.

Переваги:

Асинхронний: підтримує асинхронні операції для високої продуктивності. Гнучкість: легко налаштовується для будь-яких потреб.

Кросплатформенність: можливість створювати сервери, які працюють на різних платформах.

Приклад використання: розробка серверів для мобільних програм, які вимагають швидкої обробки запитів, наприклад програм електронної комерції або потокових служб.

SQLDelight

Опис: SQLDelight — це бібліотека для роботи з базами даних SQLite, яка генерує типізований код на основі запитів SQL.

Переваги:

Безпека типів: створіть типізований API для взаємодії з базою даних. Кросплатформенність: підтримка Android, iOS та інших платформ. Інтеграція IDE: автоматична генерація коду та підказки в IDE.

Приклади використання: використовуйте в програмах, які вимагають складного локального керування даними, наприклад у програмах для відстеження фінансів або щоденниках.

Koin

Опис: Koin — це легка бібліотека для впровадження залежностей у Kotlin, яка проста у використанні та налаштуванні.

Переваги:

Просте встановлення: просте налаштування без необхідності писати багато коду. Кодова база Kotlin: повна підтримка Kotlin, що забезпечує стислий код. Універсальність: можна використовувати в різних типах проектів, включаючи мобільні програми.

Приклади використання: використовуйте в мобільних програмах для керування залежностями, як-от програми з модульною архітектурою або програми з великою кількістю служб.

Coroutine

Опис: Kotlin Coroutines — це бібліотека асинхронного програмування, яка спрощує обробку асинхронних завдань, наприклад мережевих запитів або обробки великих обсягів даних.

Переваги:

Простота використання: легко інтегрується з існуючим кодом.

Асинхронний: ефективно обробляє асинхронні операції.

Інтеграція з іншими бібліотеками: проста інтеграція з Jetpack, Ktor та іншими бібліотеками.

Приклади використання: використовуйте в програмах, які потребують ефективної обробки асинхронних завдань, наприклад у програмах соціальних мереж або програмах новин.

2.3 Переваги Kotlin для мобільної розробки

Kotlin стає все більш популярним серед розробників мобільних додатків завдяки численним перевагам. Нижче наведено детальний огляд ключових переваг використання Kotlin у мобільній розробці.

Лаконічність та читабельність коду

Мінімізація шаблонного коду: Kotlin дозволяє зменшити кількість шаблонного коду (boilerplate), необхідного для створення додатків. Наприклад, завдяки використанню конструкцій, таких як data-класи, автоматично генеруються методи equals(), hashCode(), toString(), що значно спрощує код.

Читабельність: Завдяки лаконічному синтаксису код на Kotlin легше читати та розуміти, що сприяє кращій підтримці і масштабуванню проектів.

Безпека типів (null safety)

Запобігання NullPointerException: Kotlin має вбудовану систему безпеки типів, яка дозволяє уникати NullPointerException, що є однією з найпоширеніших помилок у Java. У Kotlin типи за замовчуванням не допускають значення null, і якщо змінна може містити null, це явно вказується.

Оператор "?" (Safe Call Operator): Оператор ?. дозволяє безпечно викликати методи на об'єктах, які можуть бути null. Це дозволяє зменшити кількість перевірок на null у коді.

Сумісність з Java та існуючими проектами

Інтероперабельність з Java: Kotlin повністю сумісний з Java, що дозволяє використовувати існуючі бібліотеки та фреймворки Java у проектах Kotlin. Ви можете поступово перенести існуючі проекти Java на Kotlin або додати Kotlin до нових проектів без необхідності переписувати весь код.

Змішане використання: ви можете поєднати код Kotlin і Java в одному проекті, що зробить перехід на Kotlin більш поступовим і менш ризикованим.

Інтероперабельність з існуючими бібліотеками

Широкий вибір бібліотек: Оскільки Kotlin сумісний з Java, розробники можуть використовувати будь-яку існуючу бібліотеку Java, що значно розширює можливості розробки. Це особливо важливо для мобільної розробки, де потрібні спеціалізовані бібліотеки для роботи з мережевими запитами, базами даних, графічним інтерфейсом користувача тощо. часто використовується.

Можливість створення власних бібліотек: розробники можуть створювати власні бібліотеки Kotlin, які можна використовувати в інших проектах, як Kotlin, так і Java.

Підтримка функцій вищого порядку

Функціональне програмування: Kotlin підтримує функції вищого порядку, лямбда-вирази та анонімні функції, що дозволяє розробникам писати більш виразний та модульний код.

Підтримка корутин для асинхронного програмування

Корутіни: Kotlin Coroutine забезпечує простий і ефективний спосіб виконання асинхронних операцій. Вони дозволяють писати асинхронний код синхронним способом, що полегшує читання та підтримку коду.

Підтримка мультиплатформенності

Kotlin Multiplatform: Kotlin Multiplatform: Kotlin Multiplatform дозволяє розробляти спільну бізнес-логіку на кількох платформах, включаючи Android, iOS, веб і сервери. Це значно скорочує витрати на розробку та підтримку кросплатформних проектів.

Сучасна екосистема та динамічна спільнота

Інструменти розробки: Kotlin підтримується такими провідними середовищами розробки, як IntelliJ IDEA та Android Studio, що забезпечує високу продуктивність.

Активна спільнота: велика спільнота розробників, велика кількість навчальних матеріалів, курсів і прикладів використання Kotlin полегшують процес навчання та вирішення проблем.

Ці переваги роблять Kotlin потужним і привабливим інструментом для розробки мобільних додатків, що сприяє його популярності серед розробників по всьому світу.

2.4 Тестування мобільних додатків на Kotlin

Тестування мобільних додатків є важливим етапом розробки, який забезпечує високу якість, надійність і стабільність додатка. Kotlin надає потужні можливості тестування, включаючи інструменти для написання різних типів тестів. У цьому розділі ми розглянемо методи тестування, інструменти та приклади використання тестування в мобільних програмах Kotlin.

Підходи до тестування:

1. Unit-тести

Опис: Unit-тести (модульні тести) перевіряють окремі компоненти або функції вашого коду ізольовано від інших частин системи.

Ціль: Забезпечити коректність окремих модулів програми.

2. Instrumentation-тести

Опис: Instrumentation-тести виконуються на реальному або емуляторному пристрої і дозволяють перевіряти взаємодію додатка з користувачем.

Ціль: Забезпечити коректну роботу додатка в реальних умовах.

3. UI-тести

Опис: UI-тести автоматизують перевірку користувацького інтерфейсу, що допомагає переконатися у правильній роботі UI-компонентів.

Ціль: Забезпечити коректну взаємодію користувача з додатком.

Інструменти для тестування

JUnit

Опис: JUnit - це популярний фреймворк для написання та виконання Unit-тестів. Він широко використовується для тестування Java та Kotlin кодів.

Особливості:

Простий синтаксис для написання тестів.

Можливість створення тестових наборів та виконання тестів в різних середовищах.

Espresso

Опис: Espresso - це фреймворк для написання UI-тестів для Android. Він дозволяє автоматизувати взаємодію з елементами інтерфейсу.

Особливості:

Простий та інтуїтивно зрозумілий API.

Можливість інтеграції з Android Studio для зручного написання та виконання тестів.

Mockito

Опис: Mockito - це фреймворк для створення мок-об'єктів, який використовується для ізоляції об'єктів при тестуванні.

Особливості:

Легке створення та налаштування мок-об'єктів.

Підтримка перевірки взаємодій між об'єктами.

Robolectric

Опис: Robolectric - це фреймворк для написання тестів, які виконуються в JVM без потреби у реальному пристрої або емуляторі.

Особливості:

Швидке виконання тестів.

Можливість тестування коду, який залежить від Android SDK.

MockK

Опис: MockK - це фреймворк для мокування, створений спеціально для Kotlin. Він підтримує мокування класів, об'єктів, функцій та багато іншого.

Особливості:

Природна інтеграція з Kotlin.

Легка у використанні API для створення мок-об'єктів та перевірки взаємодій.

2.5 Інтеграція Kotlin та Firebase

Інтеграція Kotlin і Firebase не тільки поєднує два потужні інструменти, але й створює нескінченне джерело можливостей для розробки мобільних додатків. З одного боку, Kotlin, як сучасна мова програмування, забезпечує зручний та ефективний синтаксис, який дозволяє розробникам створювати програми швидше та безпечніше. З іншого боку, Firebase від Google пропонує різноманітні інструменти та служби, які дозволяють реалізувати різноманітні функції, від зберігання даних до аналітики та автентифікації користувачів, і все це в хмарному середовищі.

Наступна комбінація Kotlin і Firebase відкриває багато можливостей. Взаємодія Kotlin з Java спрощує інтеграцію, дозволяючи легко використовувати Firebase SDK та інші бібліотеки на основі Java. Зокрема, Kotlin підтримує співпрограми, роблячи асинхронний код зрозумілим і чистим, що особливо важливо при використанні асинхронних операцій у Firebase.

При розробці мобільних додатків особливо важливим є вміння забезпечити безпеку та надійність. Firebase забезпечує автентифікацію користувачів і механізми контролю доступу до даних, допомагаючи створювати безпечні програми. Крім того, Firebase Realtime Database дозволяє синхронізувати дані між різними пристроями в режимі реального часу, забезпечуючи оновлення користувачів одразу після внесення змін.

У цьому розділі ми детально розглянемо, як інтегрувати Kotlin і різні компоненти Firebase у мобільні програми, а також переваги, які це може принести розробникам і кінцевим користувачам. На практичних прикладах і етапах інтеграції ми детально розглянемо, як покращити вашу програму за допомогою цього потужного дуету Kotlin і Firebase.

Зберігайте дані в базі даних Firebase у реальному часі

Firebase Realtime Database - це хмарна база даних, яка забезпечує синхронізацію даних у реальному часі між різними пристроями. Щоб інтегрувати

базу даних Kotlin Realtime Database і Firebase, вам потрібно буде додати залежність Firebase до вашого проекту та налаштувати підключення до бази даних.

```
// Додати залежність в build.gradle (додати останню версію)
implementation 'com.google.firebase:firebase-database-ktx'

// Ініціалізація Firebase в додатку
FirebaseApp.initializeApp(context)

// Отримання посилання на базу даних
val database = Firebase.database
val myRef = database.getReference("message")

// Запис даних в базу даних
myRef.setValue("Hello, World!")

// Отримання даних з бази даних
myRef.addValueEventListener(object : ValueEventListener {
    override fun onDataChange(dataSnapshot: DataSnapshot) {
        val value = dataSnapshot.getValue(String::class.java)
        Log.d(TAG, "Value is: $value")
    }

    override fun onCancelled(error: DatabaseError) {
        Log.w(TAG, "Failed to read value.", error.toException())
    }
})
```

Рис. 2.1 Код для зберігання даних в базі даних Firebase у реальному часі

Аутентифікація користувачів за допомогою Firebase

Аутентифікація Firebase надає механізм автентифікації користувачів за допомогою електронної пошти, номера телефону, соціальних мереж та інших методів. Інтеграція Kotlin і Firebase Authentication забезпечує безпеку та доступ до функцій програми.

```
// Додати залежність в build.gradle (додати останню версію)
implementation 'com.google.firebase:firebase-auth-ktx'

// Аутентифікація користувача
val auth = FirebaseAuth.getInstance()
auth.signInWithEmailAndPassword(email, password)
    .addOnCompleteListener(this) { task ->
        if (task.isSuccessful) {
            // Успішний вхід
            val user = auth.currentUser
        } else {
            // Помилка входу
        }
    }
```

Рис. 2.2 Код для утентифікації користувачів за допомогою Firebase

Використовуйте інші функції Firebase

Крім того, Firebase пропонує інші корисні функції, такі як аналітика, хмарний обмін повідомленнями, зберігання файлів, підтримка фотографій у хмарі тощо. Інтеграція Kotlin і Firebase дозволяє розробникам використовувати ці функції для покращення функціональності та продуктивності своїх програм.

Інтеграція Kotlin та Firebase виявляється надзвичайно корисною для розробки мобільних додатків. Використання Kotlin надає зручний та безпечний синтаксис програмування, що сприяє швидкій розробці та підтримці додатків. З іншого боку, Firebase надає широкий спектр сервісів, таких як база даних в реальному часі, аутентифікація користувачів, аналітика та інші, які спрощують створення функціональних та надійних додатків у хмарному середовищі.

2.6 Інтеграція Kotlin з Google Maps Api

Інтеграція мови програмування Kotlin з Google Maps API є актуальною темою у сфері розробки мобільних додатків. Мобільні програми, які використовують послуги на основі визначення місцезнаходження, стають все більш популярними

серед користувачів і компаній. Використання мови Kotlin, яка цікавить розробників додатків для Android, у поєднанні з Google Maps API відкриє багато можливостей для створення додатків з різними функціями геолокації.

У цій частині дипломної роботи ми детально розглянемо процес інтеграції мови Kotlin з Google Maps API. Ми розберемо технічні аспекти цієї інтеграції та розглянемо практичні аспекти використання цих технологій у мобільних додатках. Наша робота включатиме покроковий посібник з інтеграції, а також погляд на можливості та переваги, які вони приносять у розробку програм із підтримкою геолокації.

Інтеграція мови програмування Kotlin з Google Maps API виявилася важливим кроком у розробці мобільних програм із можливостями геолокації. Під час нашого дослідження ми досліджували різні можливості цієї інтеграції та дійшли висновку про її переваги для користувачів і розробників мобільних додатків.

Однією з ключових можливостей, які відкриває інтеграція Kotlin з Google Maps API, є можливість відображати та маніпулювати картографічними даними в мобільних програмах. За допомогою Google Maps API розробники можуть легко інтегрувати картографічні інтерфейси у свої програми та надавати користувачам доступ до різноманітних функцій карти, таких як відображення карти, прокладка маршрутів і використання маркерів.

Крім того, інтеграція Kotlin з Google Maps API дозволяє розробникам створювати програми з розширеними функціями геолокації, такими як моніторинг місцезнаходження користувачів, відстеження їхніх переміщень і навігація до місць призначення. Ці функції роблять додаток більш цікавим і корисним для користувачів, сприяючи збільшенню його популярності та успіху на ринк.

Загалом, інтеграція Kotlin з Google Maps API відкриває багато можливостей для розробки різних мобільних додатків із функцією геолокації. Таке поєднання технологій дозволяє розробникам створювати інноваційні та конкурентоспроможні програми, які відповідають потребам сучасних користувачів і потребам ринку мобільних додатків.

3 РЕАЛІЗАЦІЯ МОБІЛЬНОГО ДОДАТКУ-ГІДА

3.1 Архітектурний огляд та взаємозв'язки компонентів

Розглянемо основні функції мобільного додатка-гіда для Android зі схемами. Додаток розроблено з використанням мови програмування Kotlin і технологій API Firebase і Google Maps, що забезпечує автентифікацію користувачів, надійне зберігання й обробку даних, а також інтеграцію з картою сервісів.

Основна мета цього додатка – надати користувачам можливість швидко та легко шукати організації, переглядати інформацію про них, робити бронювання, залишати відгуки та бачити розташування організацій на карті. Для адміністраторів передбачені функції управління організаційними даними та оновлення вмісту програми.

У цьому розділі представлені діаграми варіантів використання, які детально ілюструють взаємодію користувачів і адміністраторів із програмою, а також послідовність ключових дій у системі. Ці діаграми дозволяють краще зрозуміти логіку програми та взаємозв’язки між різними компонентами системи.

3.1.1 Діаграма Use-Case

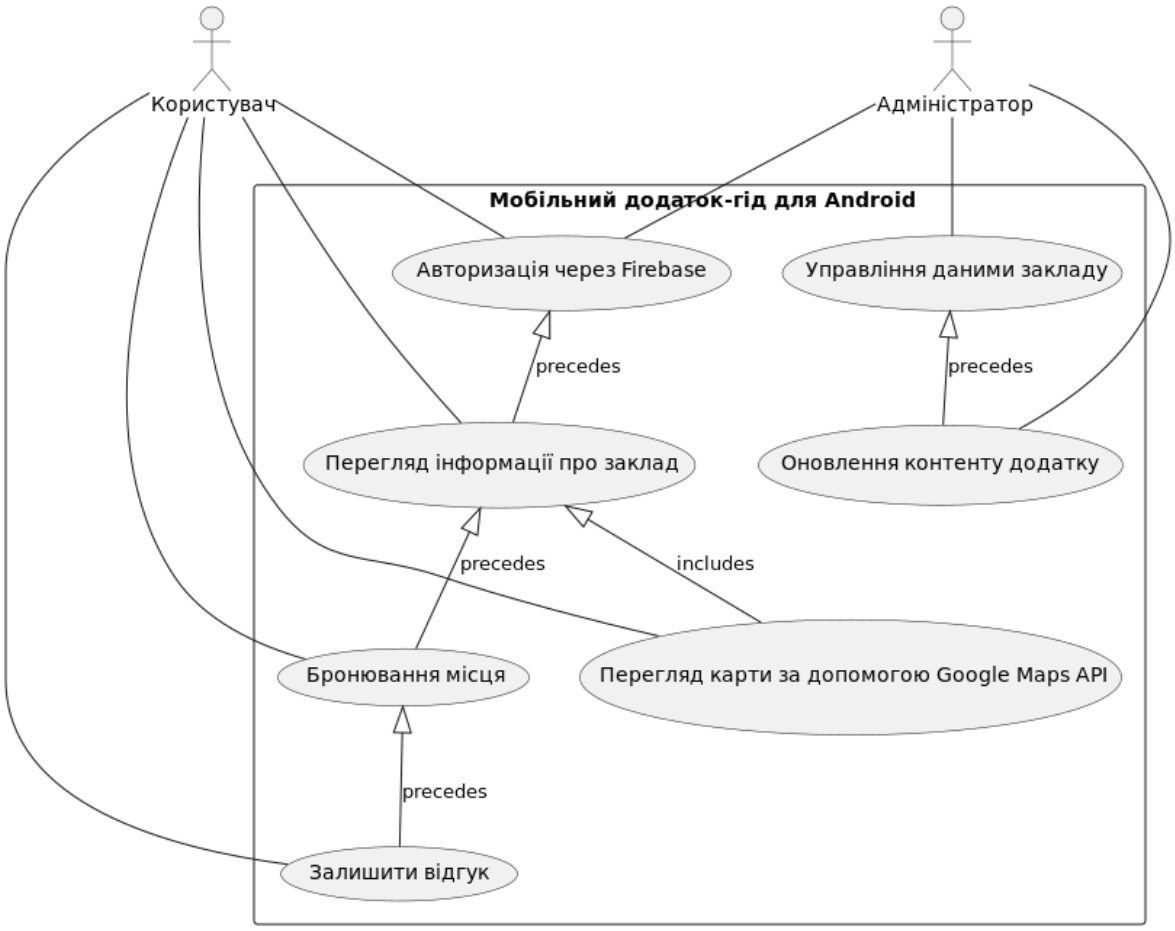


Рис. 3.1 Діаграма Use-Case для мобільного додатку-гіду

Актори:

Користувач - люди, які використовують додаток для перегляду інформації про організації, бронювання місць, залишають відгуки та переглядають карти.

Адміністратор — це особа, яка керує даними закладу та оновлює вміст програми.

Випадки використання:

Авторизація Firebase - процес входу користувачів у систему за допомогою автентифікації Firebase.

Актори: користувач, адміністратор.

Попередні кроки: немає.

Перегляд інформації про організацію - користувачі можуть переглядати детальну інформацію про організацію.

Актор: Користувач.

Попередні кроки: авторизуйтеся через Firebase.

Бронювання - користувачі можуть зробити бронювання в закладі.

Актор: Користувач.

Попередні кроки: переглянути інформацію про встановлення. Включає: відображення карти за допомогою API Карт Google.

Залишити відгук - користувачі можуть залишити відгук про організацію після відвідування.

Актор: Користувач.

Попередні кроки: Бронювання.

Перегляд карт за допомогою Google Maps API: користувачі можуть переглядати карти з функціями за допомогою Google Maps API.

Актор: Користувач.

Попередні кроки: переглянути інформацію про встановлення.

Керуйте даними організації – адміністратори можуть додавати, редагувати або видаляти інформацію про організацію.

Актор: Адміністратор.

Попередні кроки: авторизуйтеся через Firebase.

Оновлення вмісту програми – адміністратори можуть оновлювати вміст програми, додаючи нову інформацію або змінюючи існуючу.

Актор: Адміністратор.

Попередні кроки: управління інституційними даними.

3.1.2 Контекстна діаграма

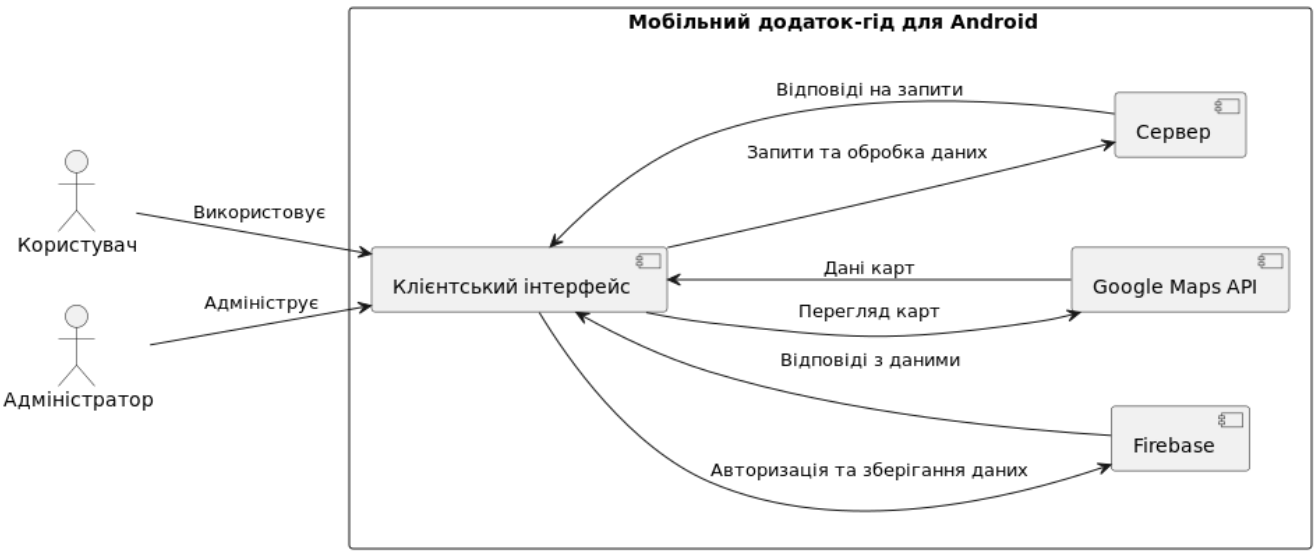


Рис. 3.2 Контекстна діаграма для мобільного додатку-гіду

Контекстна діаграма, яка показує взаємозв'язки між ключовими системними компонентами та зовнішніми учасниками для програми мобільного путівника, розробленої на основі Kotlin із використанням Firebase та Google Maps API. Нижче наведено опис кожного компонента та їх взаємодії:

Актори:

Користувачі - люди, які використовують мобільний додаток для перегляду інформації про об'єкти, бронювання місць, залишають відгуки та переглядають карти.

Адміністратор - це особа, яка керує даними закладу та оновлює вміст програми. Компоненти:

Мобільний додаток-гід для Android :

Клієнтський інтерфейс (UI) - інтерфейс користувача програми, що забезпечує взаємодію користувачів і адміністраторів із системою.

Firebase — це платформа, яка надає послуги автентифікації користувачів і зберігання даних. Google Maps API - сервіс, який інтегрує картографічні дані в програми.

Сервер - сервер обробляє запити клієнтського інтерфейсу та повертає відповіді.

Взаємодії:

Користувач взаємодіє з клієнтським інтерфейсом програми:

Використовує додаток для автентифікації через Firebase.

Переглядає інформацію про компанію.

Переглядає карти за допомогою Google Maps API.

Бронює місця у закладах.

Залишає відгуки про організації.

Адміністратор взаємодіє з клієнтським інтерфейсом програми:

Використовує додаток для автентифікації через Firebase.

Керує даними організації (додавайте, змінюйте, видаляйте інформацію).

Оновлює вміст програми.

Клієнтський інтерфейс програми взаємодіє з Firebase:

Надсилає запити на авторизацію користувачам і адміністраторам.

Отримує відповіді із даними авторизації та зберегти дані.

Клієнтський інтерфейс програми взаємодіє з Google Maps API:

Надсилає запит на отримання картографічних даних.

Отримує дані карти для відображення в програмі.

Інтерфейс клієнтської програми, що взаємодіє з Сервером:

Надсилає запити на обробку даних і виконуйте різні операції (наприклад, бронювання, оновлення вмісту).

Отримує відповіді на запити з результатами обробки даних.

Ця контекстна діаграма допомагає зрозуміти, як різні компоненти системи взаємодіють один з одним і із зовнішніми учасниками, забезпечуючи належне функціонування програми мобільного посібника.

3.1.3 Діаграма станів

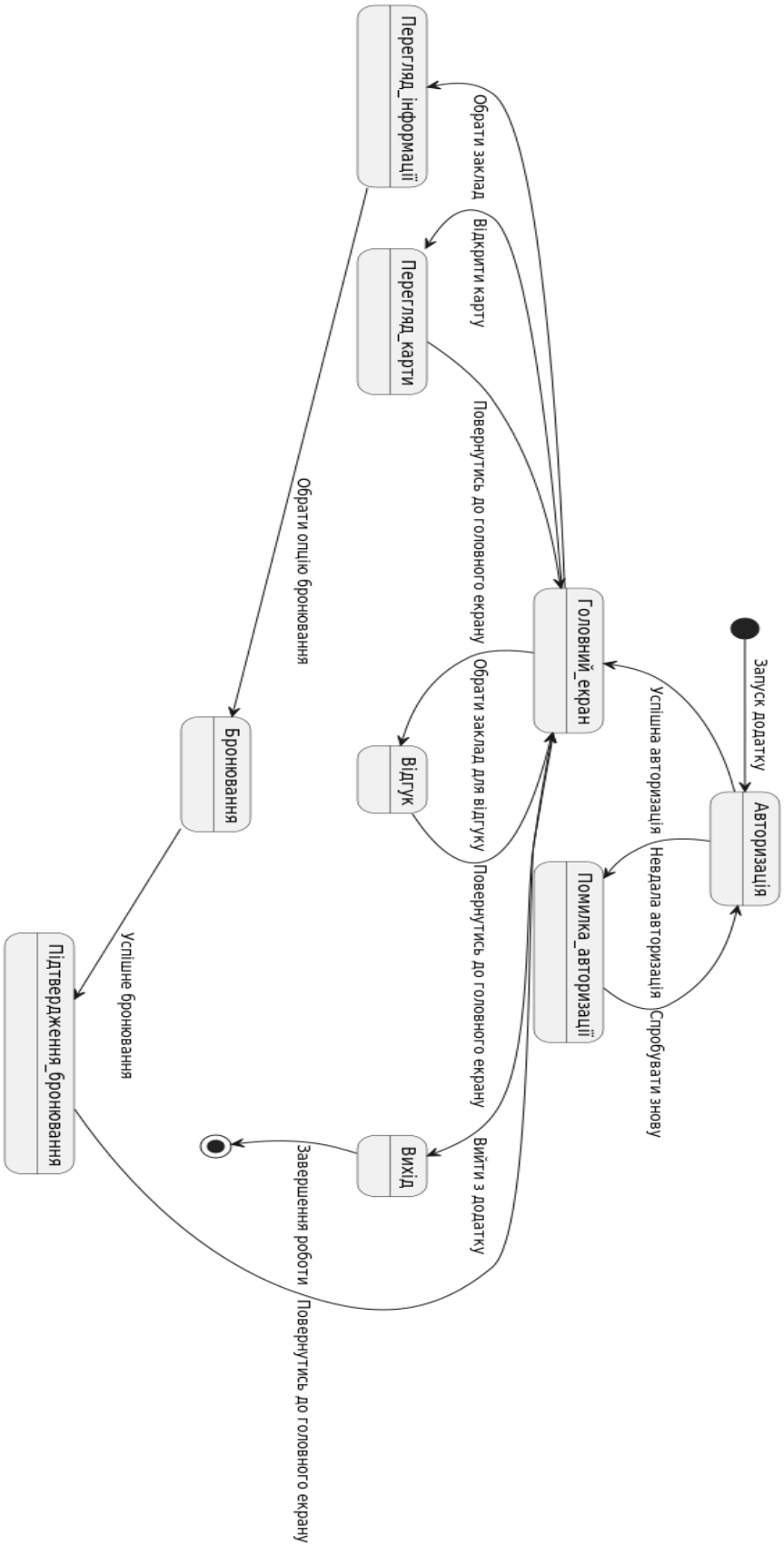


Рис. 3.3 Діаграма станів для мобільного додатку-гіду

Діаграма станів ілюструє різні стани, через які проходить програма мобільного путівника для Android за допомогою Kotlin, Firebase та API Карт Google. Описано послідовність станів від запуску програми до виходу.

Опис станів:

1. *Запуск:* Початковий стан програми, що активується при запуску.

2. *Авторизація:* Користувачі надають свої облікові дані через автентифікацію Firebase та при необхідності вказати своє місто. Якщо авторизація пройшла успішно, програма перейде на головний екран. Якщо авторизація не вдається, користувач отримає повідомлення про помилку авторизації.

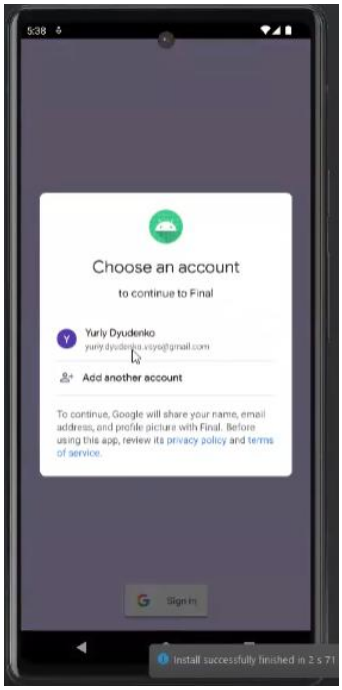


Рис. 3.4 Авторизація з допомогою Google аккаунту на основі Firebase

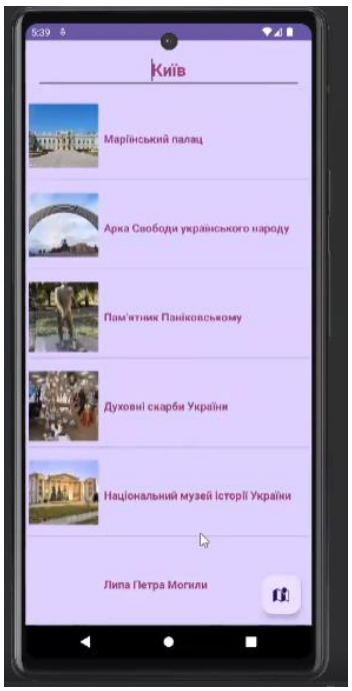


Рис. 3.5 Візуальна можливість вказати своє місто у додатку

3. *Головний екран:* Центральний екран програми, з якого користувачі можуть вибирати різні дії. На головному екрані користувачі можуть переглянути інформацію про заклад, переглянути карти, зробити бронювання або залишити відгук.

Рис. 3.6 Головний екран додатку

4. *Помилка авторизації:* Відображається, коли авторизація не вдається. Користувач може спробувати увійти ще раз.

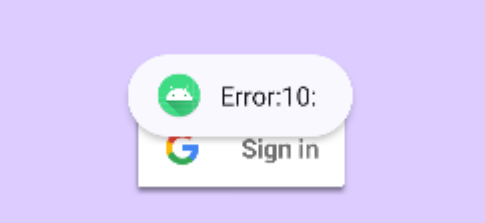


Рис. 3.7 Помилка при авторизації

5. *Відображення інформації:* Користувачі можуть переглянути детальну інформацію про обрану організацію. З цього статусу користувач може продовжити бронювання.

Рис. 3.8 Інформація про доступні об’єкти

6. *Перегляд карти:* Користувачі можуть переглядати карту з об’єктами за допомогою Google Maps API. Після перегляду карти користувачі можуть повернутися на головний екран.

7. *Бронювання:* Користувачі можуть зробити бронювання в обраному закладі. Успішне бронювання підтверджується, і користувач повертається на головний екран.

8. *Підтвердження бронювання:* Відобразиться повідомлення про успішне бронювання. Після підтвердження користувач повертається на головний екран.

9. *Залишити коментар:* Користувачі можуть залишати відгуки про процес встановлення. Залишивши коментар, користувач повертається на головний екран.

10. *Вихід:* Користувач може вийти з програми. Після виходу програма завершить роботу та перейде в кінцевий стан.

11. *Перехідні стани:* Переходи між станами відбуваються на основі дій користувача. Наприклад, після успішної авторизації програма переходить на головний екран, звідки користувач може переглянути інформацію, карти, зробити бронювання чи залишити відгук.

Ця діаграма дозволяє зрозуміти логіку програми та зв’язок між різними станами під час її використання.

3.1.4 Діаграма компонентів

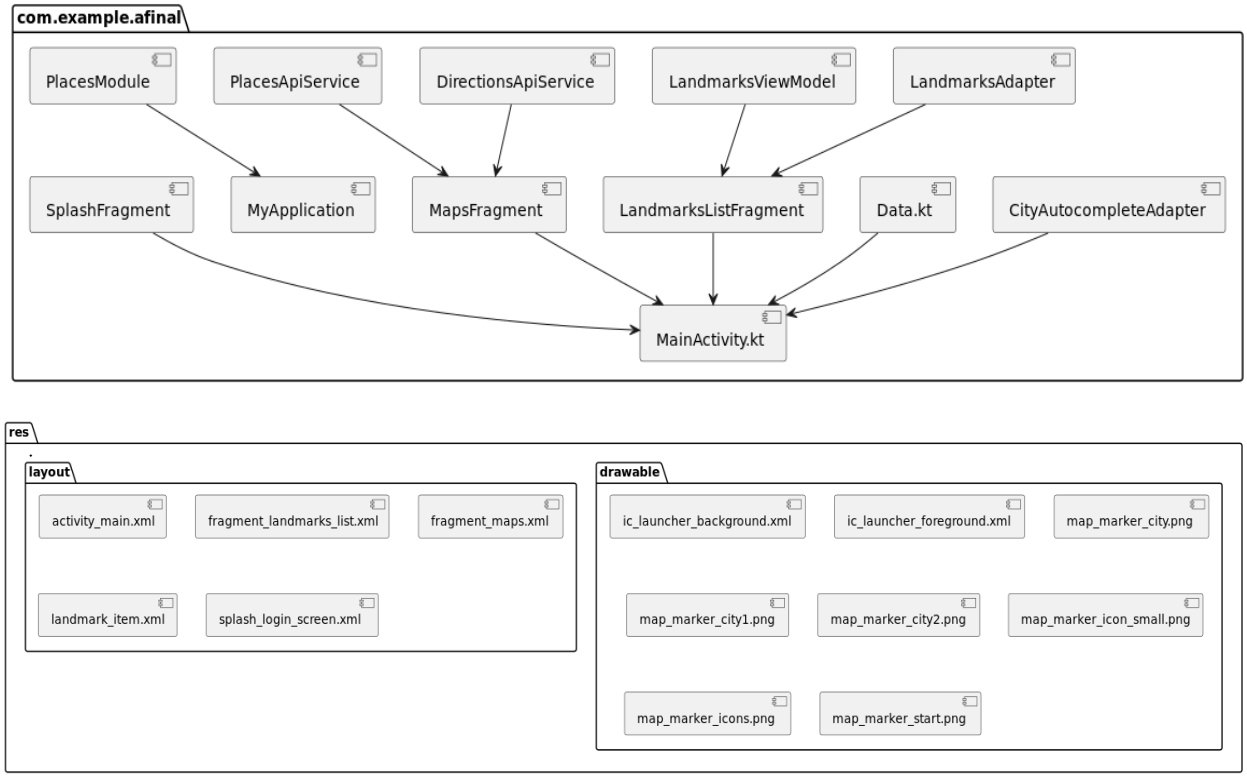


Рис. 3.9 Діаграма компонентів для мобільного додатку-гіду

Діаграма компонентів ілюструє структуру проекту мобільного додатку-гіду для Android, що використовує Kotlin, Firebase та Google Maps API. Вона відображає основні компоненти системи та їх взаємозв'язки.

Компоненти:

com.example.afinal: Основний пакет, який містить класи та фрагменти, що складають бізнес-логіку та функціонал додатку.

CityAutocompleteAdapter: Клас адаптера для автозаповнення міста.

Data.kt: Клас для роботи з даними.

DirectionsApiService: Сервіс для роботи з API напрямків.

LandmarksAdapter: Адаптер для роботи зі списком визначних місць.

LandmarksListFragment: Фрагмент, що відображає список визначних місць.

LandmarksViewModel: Модель представлення для роботи зі списком визначних місць.

MainActivity.kt: Основна активність додатку.

MapsFragment: Фрагмент для відображення карти.

MyApplication: Клас для роботи з додатком.

PlacesApiService: Сервіс для роботи з API місць.

PlacesModule: Модуль для роботи з місцями.

SplashFragment: Фрагмент для сплеш-екрану.

res: Пакет, що містить ресурси додатку.

drawable: Підпакет з графічними ресурсами.

ic_launcher_background.xml: Фон для іконки додатку.

ic_launcher_foreground.xml: Передній план для іконки додатку.

map_marker_city.png: Іконка маркера міста.

map_marker_city1.png: Іконка маркера міста (варіант 1).

map_marker_city2.png: Іконка маркера міста (варіант 2).

map_marker_icon_small.png: Маленька іконка маркера.

map_marker_icons.png: Іконки маркерів.

map_marker_start.png: Іконка маркера початкової точки.

layout: Підпакет з XML файлами макетів.

activity_main.xml: Макет для основної активності.

fragment_landmarks_list.xml: Макет для фрагмента зі списком визначних місць.

fragment_maps.xml: Макет для фрагмента з картою.

landmark_item.xml: Макет для елемента списку визначних місць.

splash_login_screen.xml: Макет для сплеш-екрану.

Взаємозв'язки між компонентами:

CityAutocompleteAdapter використовується у MainActivity.kt.

Data.kt використовується у MainActivity.kt.

DirectionsApiService використовується у MapsFragment.

LandmarksAdapter використовується у LandmarksListFragment.

LandmarksListFragment використовується у MainActivity.kt.

LandmarksViewModel використовується у LandmarksListFragment.

MapsFragment використовується у MainActivity.kt.

PlacesApiService використовується у MapsFragment.

PlacesModule використовується у MyApplication.

SplashFragment використовується у MainActivity.kt.

3.1.5 Діаграма послідовностей

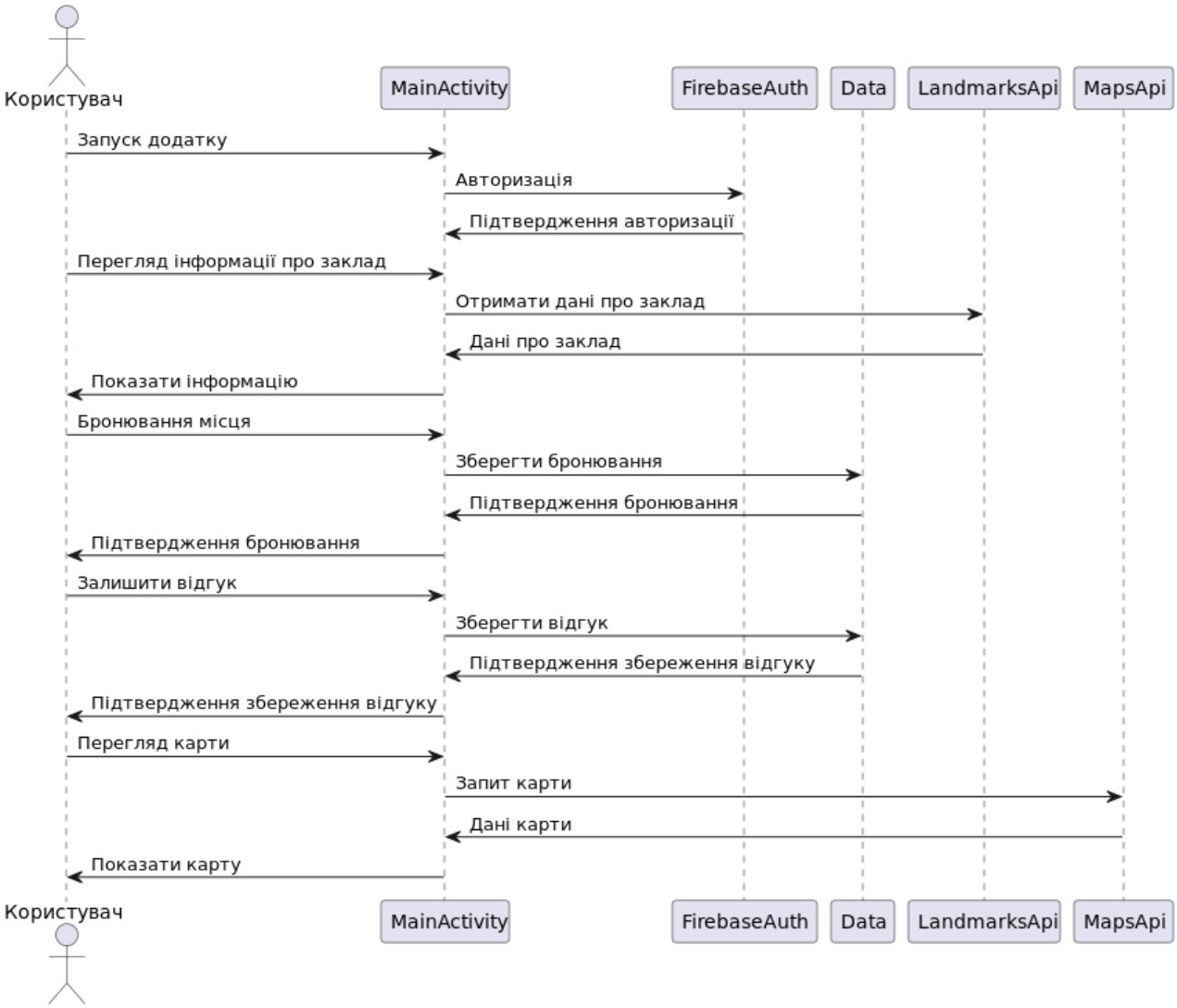


Рис. 3.10 Діаграма послідовностей для мобільного додатку-гіду

Діаграма послідовностей ілюструє взаємодію між користувачем, різними компонентами додатку та зовнішніми сервісами під час виконання основних дій у мобільному додатку-гіді для Android з використанням Kotlin, Firebase та Google Maps API.

Ключові дії та взаємодії:

1. *Запуск додатку та авторизація*: Користувач запускає програму та викликає виклик MainActivity. MainActivity виконує авторизацію через FirebaseAuth. FirebaseAuth перевіряє авторизацію та повертає результати до MainActivity.

2. *Переглянути інформацію про заклад:* Користувач запитує перегляд інформації про організацію через MainActivity. MainActivity надсилає запит LandmarksApi, щоб отримати дані налаштувань. LandmarksApi повертає дані про встановлення в MainActivity. MainActivity відображає інформацію для користувача.

3. *Бронювання:* Користувач ініціює бронювання через MainActivity. MainActivity зберігає інформацію про бронювання в Data. Дані підтверджують успішне збереження резервування та повертають результати до MainActivity. MainActivity відображає підтвердження бронювання для користувача.

4. *Залишити відгук:* Користувач ініціює відповідь через MainActivity. MainActivity зберігає відповідь у Data. Дані підтверджують успішне збереження відповіді та повертають результат до MainActivity. MainActivity відображає підтвердження збереження коментаря для користувача.

5. *Перегляд карти:* Користувач запитує перегляд карти через MainActivity. MainActivity надсилає запит до MapsApi, щоб отримати дані карти. MapsApi повертає дані карти в MainActivity. MainActivity відображає карту для користувача.

Компоненти:

– Користувач: головний агент, який взаємодіє з програмою.

– MainActivity: основна діяльність програми, яка контролює більшість дій користувача.

– FirebaseAuth: служба авторизації користувачів через Firebase.

– Дані: компонент для зберігання таких даних, як бронювання та відгуки.

– LandmarksApi: Сервіс для збору інформації про організації.

– MapsApi: служба, яка отримує картографічні дані через Google Maps API.

Послідовність дій:

1. Користувач запускає програму.

2. Автентифікація користувачів здійснюється через Firebase.

3. Користувачі переглядають інформацію про організацію, дані отримують через API.

4. Користувач робить бронювання, інформація зберігається в базі даних.

5. Користувачі залишають відгуки, відгуки зберігаються в базі.

6. Користувач переглядає карту, картографічні дані отримують через Google Maps API.

Ця схема та опис допомагають зрозуміти взаємодію між користувачами та різними компонентами програми та зовнішніми службами під час виконання основних дій у програмі.

3.1.6 Діаграма класів з використанням MVC архітектури

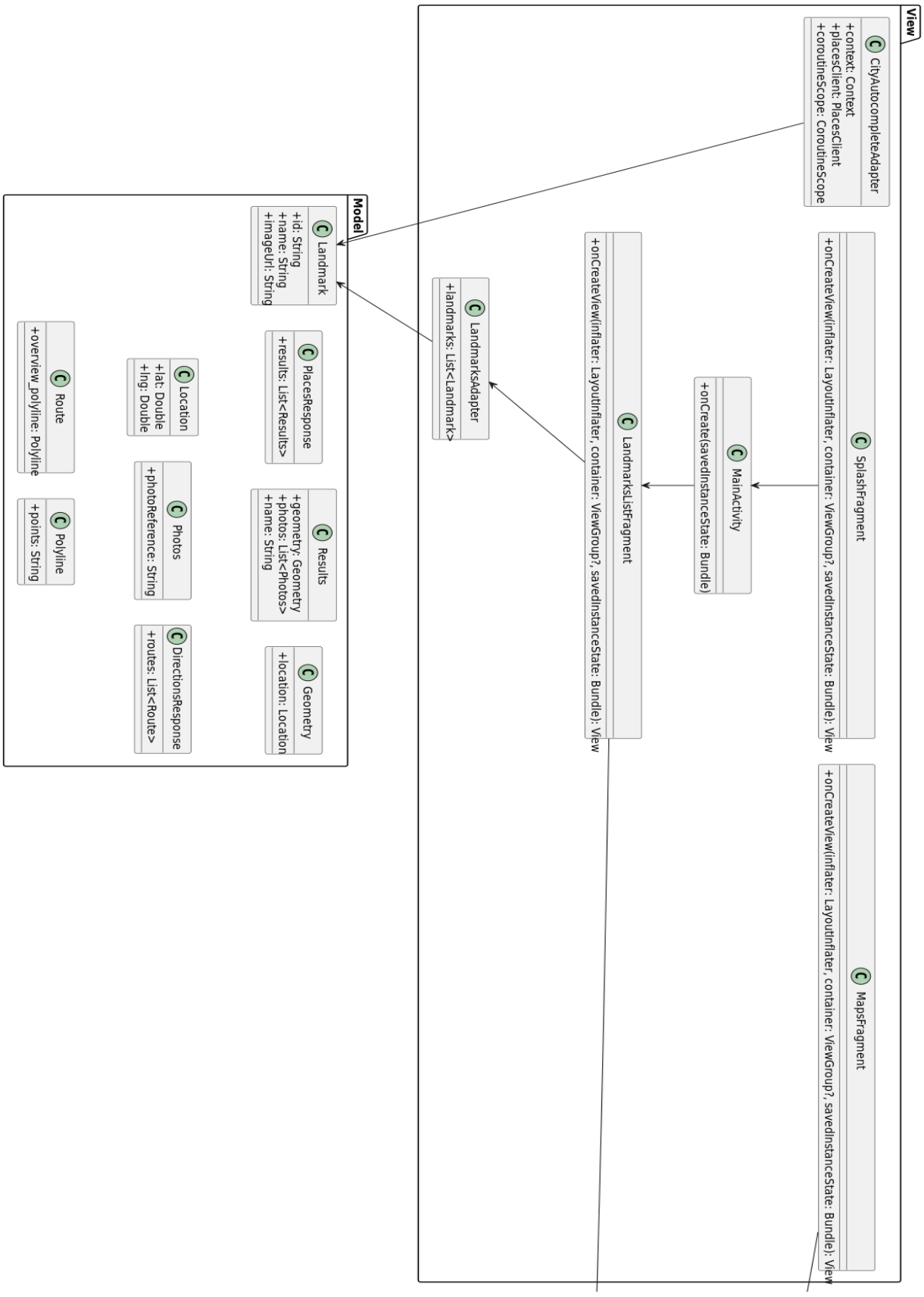


Рис. 3.11 Діаграма класів з використанням MVC архітектури для мобільного додатку-гіду

Рис. 3.12 Діаграма класів з використанням MVC архітектури для мобільного додатку-гіду (продовження)

Діаграма класів ілюструє структуру мобільного додатку-гіду для Android, розробленого з використанням архітектури MVC (Model-View-Controller). Вона показує основні класи, які відповідають за модель, представлення та контролери, а також їх взаємозв'язки.

Модель (Model):

1. *Landmark a*трибути:

– id: String

– name: String

– imageUrl: String

2. *PlacesResponse* атрибути: results: List<Results>
3. *Results* атрибути:

– geometry: Geometry

– photos: List<Photos>

– name: String

4. *Geometry* атрибути: location: Location
5. *Location* атрибути:

– lat: Double

– lng: Double

6. *Photos* Атрибути: photoReference: String
7. *DirectionsResponse* атрибути: routes: List<Route>
8. *Route* атрибути: overview_polyline: Polyline
9. *Polyline* атрибути: points: String

Представлення (View):

1. *CityAutocompleteAdapter* атрибути:

– context: Context

– placesClient: PlacesClient

– coroutineScope: CoroutineScope

2. *LandmarksAdapter* атрибути:

– landmarks: List<Landmark>

3. *MainActivity* методи: onCreate(savedInstanceState: Bundle)

4. *LandmarksListFragment* методи: onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle): View

5. *MapsFragment* методи: onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle): View

6. *SplashFragment* методи: onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle): View

Контролер (Controller):

1. *LandmarksViewModel* методи: loadLandmarks(cityLatLng: LatLng?)

2. *DirectionsApiService* методи: getComplexRoute(origin: String, destination: String, waypoints: String, mode: String, apiKey: String): Response<DirectionsResponse>

3. *PlacesApiService* методи: getNearbyPlaces(location: String, radius: Int, type: String, apiKey: String): Response<PlacesResponse>

4. *PlacesModule* методи:

– providePlacesClient(application: Application): PlacesClient

– provideHttpLoggingInterceptor(): HttpLoggingInterceptor

– provideOkHttpClient(loggingInterceptor: HttpLoggingInterceptor): OkHttpClient

– provideRetrofit(okHttpClient: OkHttpClient): Retrofit

– providePlacesApiService(retrofit: Retrofit): PlacesApiService

– provideDirectionsApiService(retrofit: Retrofit): DirectionsApiService

– provideApplicationContext(application: Application): Context

Взаємозв'язки між класами:

– CityAutocompleteAdapter взаємодіє з Landmark для відображення автозаповнення.

– LandmarksAdapter взаємодіє з Landmark для відображення списку визначних місць.

– LandmarksListFragment використовує LandmarksAdapter для відображення списку визначних місць.

– LandmarksListFragment взаємодіє з LandmarksViewModel для отримання даних.

– LandmarksViewModel використовує PlacesApiService та DirectionsApiService для отримання даних про визначні місця та маршрути.

– MapsFragment використовує DirectionsApiService для отримання даних про маршрути.

– MainActivity взаємодіє з LandmarksListFragment для відображення списку визначних місць.

– SplashFragment взаємодіє з MainActivity для відображення сплеш-екрану.

Ця діаграма та опис допомагають зрозуміти структуру додатку, його компоненти та їх взаємодію, що важливо для підтримки, розвитку та масштабування проекту.

3.1.7 Діаграма розгортання

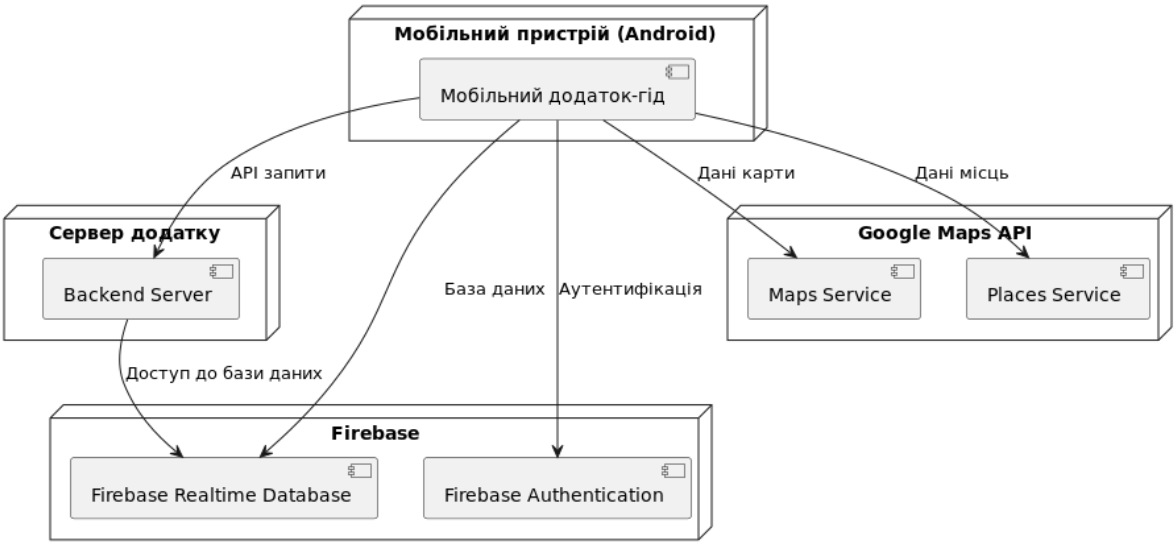


Рис. 3.13 Діаграма розгортання для мобільного додатку-гіду

Діаграми розгортання ілюструють фізичне розташування різних системних компонентів і їхні взаємозв’язки в покроковому керівництві мобільної програми

Android. Він показує, як різні частини системи взаємодіють одна з одною та їх точне розташування.

Компоненти діаграми:

– *Мобільні пристрої (Android):* Інструкції щодо мобільного додатку: клієнтський додаток встановлюється на мобільний пристрій користувача. Програма взаємодіє з іншими компонентами для виконання різноманітних завдань.

– *Сервер додатку:* Внутрішній сервер: сервер додатків обробляє запити API від мобільного додатка та взаємодіє з іншими службами, такими як Firebase.

– *Firebase:*

- Автентифікація Firebase: служба автентифікації користувача, яка використовується мобільними програмами для перевірки облікових даних користувача.

- База даних у реальному часі Firebase: база даних у реальному часі використовується для зберігання та отримання даних програми.

– *API Карт Google:*

- Картографічний сервіс: сервіс, який отримує картографічні дані, які використовуються мобільними програмами для відображення карт.

- Служби визначення місцезнаходження: служби даних про місцезнаходження використовуються мобільними програмами для відображення інформації про цікаві місця.

Взаємозв'язки між компонентами:

1. Мобільний додаток-гід взаємодіє з:

– Аутентифікація Firebase для автентифікації користувачів.

– Firebase Realtime Database для доступу до даних бази даних у реальному часі.

– Картографічний сервіс для отримання та відображення картографічних даних.

– Служба локації для отримання та відображення даних про цікаві місця.

– Внутрішній сервер для обробки запитів API.

– *Внутрішній сервер взаємодіє з* Firebase Realtime Database для доступу до бази даних у реальному часі.

Опис взаємодій:

– Коли користувач відкриває програму на своєму мобільному пристрої, програма надсилає запит до Firebase Authentication для перевірки облікових даних користувача. Якщо аутентифікація пройшла успішно, користувач може продовжити роботу з додатком.

– Мобільний додаток взаємодіє з базою даних Firebase Realtime Database, щоб зберігати та отримувати дані в режимі реального часу. Наприклад, дані про визначні місця або маршрути.

– Для відображення карт і місцезнаходження програма використовує Google Maps API (компоненти Map Services і Places Services), отримуючи відповідні дані через запити API.

– Внутрішній сервер обробляє запити API від мобільного додатка та взаємодіє з базою даних Firebase Realtime для доступу до необхідних даних.

Ця схема та опис допоможуть вам зрозуміти, як різні компоненти системи взаємодіють один з одним, де вони розташовані та як працює програма мобільного посібника для Android.

3.2 Структура мобільного додатку

3.2.1 Архітектурний підхід

У сучасному світі мобільні програми відіграють важливу роль у повсякденному житті людей, допомагаючи людям швидко отримувати доступ до інформації та послуг. Розробка таких додатків вимагає ретельного планування та проектування, щоб забезпечити їх функціональність, продуктивність і простоту використання. У цьому розділі буде розглянуто структуру мобільної програми для пошуку та відображення цікавих місць на карті за допомогою Google Places API та Google Directions API.

Опис структури

1. Основні компоненти:

1.1 Адаптери:

– *CityAutocompleteAdapter:* Цей адаптер обробляє автозаповнення назв міст за допомогою Google Places API. Він надає інструкції користувачеві на основі введення тексту. Для цього використовується асинхронне завантаження даних за допомогою співпрограми.

– LandmarksAdapter: Адаптер прив’язує дані POI до елементів інтерфейсу RecyclerView. Він відповідає за відображення назв пам'яток і їх зображень, завантажених за допомогою бібліотеки Пікассо. 1.2

1.2 Моделі даних:

– *Landmark:* Шаблон, який представляє визначне місце з такими атрибутами, як ідентифікатор зображення, назва та URL-адреса.

– *PlacesResponse, Results, Geometry, Location, Photos:* Моделі обробки відповідей Google Places API. Вони містять інформацію про форму місць, їх фотографії та інші атрибути.

– *DirectionsResponse, Route, Polyline:* Моделі, які обробляють відповіді Google Directions API, містять дані про маршрути та їх полігони.

1.3 API сервіси:

– *DirectionsApiService*: Інтерфейс, який визначає метод отримання інформації про маршрут із Google Directions API. Він використовує запити з такими параметрами, як точка відправлення, точка призначення, маршрутна точка, режим руху та ключ API.

– *PlacesApiService:* Інтерфейс для отримання інформації про визначні місця поблизу від Google Places API. Він містить такі налаштування, як розташування, радіус пошуку, тип розташування, ключ API та мова. 2. Компоненти

2. Компоненти UI:

2.1 Фрагменти:

– *LandmarksListFragment:* Фрагмент із списком улюблених місць. Він використовує CityAutocompleteAdapter для входу та пошуку міст. Після того, як користувач вибере місто, цей фрагмент оновить список орієнтирів, отримавши їх за допомогою LandmarksViewModel.

– *MapsFragment:* Сегмент, що показує карту з розташуванням міста та визначними місцями. Він також може відображати маршрут між цими місцями за допомогою DirectionsApiService для отримання даних маршруту.

– *SplashFragment:* Код показує екран-заставку з кнопкою входу. Він обробляє облікові дані користувача через вхід Google і отримує доступ до LandmarkListFragment після успішної автентифікації.

2.2 Активність:

– *MainActivity:* Основна діяльність містить фрагменти та керує навігацією між ними. Він також виконує автентифікацію за допомогою входу в систему Google і завантажує відповідні сценарії на основі статусу входу користувача. 3.

3. Ін'єкція залежностей:

– PlacesModule: Модуль, який забезпечує впровадження залежностей за допомогою Dagger. Він надає екземпляри PlacesClient, OkHttpClient, Retrofit, PlacesApiService та DirectionsApiService.

4. Клас додатку:

– MyApplication: Прикладний рівень ініціалізує API Google Places під час запуску програми.

3.2.1 Архітектурний підхід

Архітектурний підхід, який використовується в цій програмі, базується на парадигмі MVVM (Model-View-ViewModel). Цей підхід розділяє бізнес-логіку та подання, полегшуючи тестування та підтримку коду.

Основи архітектури MVVM:

– Модель (Модель): Відповідає за управління даними та бізнес-логікою програми. У нашому випадку це API-сервіси, які отримують дані від Google Places

API і Google Directions API. Наприклад, PlacesApiService та DirectionsApiService взаємодіють з відповідними API і повертають дані про визначні місця та маршрути.

– View: Відповідає за відображення даних і взаємодію з користувачем. Це фрагменти, які показують список визначних місць і карту. У нашому випадку це LandmarksListFragment, MapsFragment та SplashFragment

– ViewModel: Відповідає за підготовку даних для презентації та керування станом інтерфейсу користувача. Це модель LandmarksViewModel, яка отримує дані від API служби та робить їх доступними для перегляду через LiveData. У LandmarksViewModel дані про визначні місця та їх координати зберігаються у LiveData об'єктах, що дозволяє автоматично оновлювати UI при зміні даних.

Переваги використання MVVM:

– Чіткий розподіл обов’язків: модель відповідає за дані та бізнес-логіку, View відповідає за відображення даних і взаємодію з користувачами, а ViewModel відповідає за посередництво між ними.

– Легко перевірити: оскільки бізнес-логіка та обробка даних містяться в моделі та моделі перегляду, їх легко перевірити незалежно від інтерфейсу користувача.

– Зручне керування станом: використання LiveData у ViewModel дозволяє легко керувати станом інтерфейсу користувача та оновлювати його, коли дані змінюються.

3.2.2 Класи та їх функції

Розглянемо основні класи, які використовуються в мобільному додатку для пошуку та відображення визначних місць, їх функціональність та взаємодію.

1. CityAutocompleteAdapter:

CityAutocompleteAdapter відповідає за автозаповнення назв міст, використовуючи Google Places API.

Цей клас розширює *ArrayAdapter<AutocompletePrediction>* та реалізує інтерфейс *Filterable*.

– Поле *predictions*: Зберігає список передбачень автозаповнення.

– Метод *getCount*(): Повертає кількість передбачень.

– Метод *getItem*(position: Int): Повертає передбачення для конкретної позиції.

– Метод *getView*(position: Int, convertView: View?, parent: ViewGroup): Відображає передбачення в списку.

– Метод *getFilter*(): Повертає фільтр для обробки введення користувача та отримання передбачень від Google Places API.

2. Landmark:

Landmark - це модель даних, яка представляє визначне місце.

– Поле *id*: Унікальний ідентифікатор визначного місця.

– Поле *name*: Назва визначного місця.

– Поле *imageUrl*: URL зображення визначного місця.

3. PlacesResponse:

PlacesResponse представляє відповідь від Google Places API.

– Поле *results*: Список результатів (об'єкти класу Results).

4. Results:

Results - це модель, яка представляє результат від Google Places API.

– Поле *geometry*: Геометрична інформація про місце (об'єкт класу Geometry).

– Поле *photos*: Список фотографій місця (об'єкти класу Photos).

– Поле *name*: Назва місця.

5. Geometry:

Geometry представляє геометричну інформацію про місце.

– Поле *location*: Координати місця (об'єкт класу Location).

6. Location:

Location представляє координати місця.

– Поле lat: Широта.

– Поле lng: Довгота.

7. Photos:

Photos представляє фотографії місця.

– Поле *photoReference*: Посилання на фото.

8. DirectionsResponse:

DirectionsResponse представляє відповідь від Google Directions API.

– Поле routes: Список маршрутів (об'єкти класу Route).

9. Route:

Route представляє маршрут.

– Поле *overview_polyline*: Полігональна лінія маршруту (об'єкт класу *Polyline*).

10. Polyline:

Polyline представляє полігональну лінію маршруту.

– Поле *points*: Точки, що складають полігональну лінію.

11. DirectionsApiService:

DirectionsApiService - це інтерфейс, який визначає метод для отримання інформації про маршрут від Google Directions API.

– Метод *getComplexRoute*: Виконує запит до Google Directions API для отримання маршруту між точками.

12. LandmarksAdapter:

LandmarksAdapter відповідає за відображення списку визначних місць у *RecyclerView*.

– Клас *LandmarkViewHolder*: Внутрішній клас, який тримає посилання на елементи інтерфейсу для кожного визначного місця.

– Метод *onCreateViewHolder*: Створює новий *LandmarkViewHolder*.

– Метод *onBindViewHolder*: Прив'язує дані до *LandmarkViewHolder*.

– Метод *getItemCount*: Повертає кількість визначних місць.

– Метод *updateData*: Оновлює дані у списку та повідомляє RecyclerView про зміни.

13. LandmarksListFragment:

LandmarksListFragment відповідає за відображення списку визначних місць та взаємодію з користувачем для пошуку міст.

– Поле *binding*: Зв'язує елементи інтерфейсу з фрагментом.

– Поле *viewModel*: Отримує дані з *LandmarksViewModel*.

– Поле *placesClient*: Використовується для роботи з Google Places API.

– Метод *onCreateView*: Створює та повертає вигляд фрагменту.

– Метод *onViewCreated*: Виконує налаштування після створення вигляду, включаючи ініціалізацію *PlacesClient*, налаштування *RecyclerView* та обробку введення міст.

14. LandmarksViewModel:

LandmarksViewModel управляє даними для *LandmarksListFragment*.

– Поле *_landmarks*: Зберігає список визначних місць.

– Поле *landmarks*: Публічний доступ до списку визначних місць.

– Поле *_cityLatLng*: Зберігає координати міста.

– Поле *cityLatLng*: Публічний доступ до координат міста.

– Поле *_landmarksLatLng*: Зберігає координати визначних місць.

– Поле *landmarksLatLng*: Публічний доступ до координат визначних місць.

– Метод *loadLandmarks*: Завантажує визначні місця для заданих координат міста.

15. MainActivity:

MainActivity є головною активністю, яка управляє фрагментами та навігацією між ними.

– Метод *onCreate*: Виконує налаштування активності та завантажує початковий фрагмент.

– Метод *launch*: Запускає нову активність для аутентифікації.

– Метод *showListFragment*: Показує фрагмент з списком визначних місць.

– Метод *onActivityResult*: Обробляє результат аутентифікації через Google Sign-In.

16. MapsFragment:

MapsFragment відповідає за відображення карти з визначними місцями та маршрутами.

– Поле *directionsApiService*: Використовується для роботи з Google Directions API.

– Поле *map*: Зберігає посилання на об'єкт Google Map.

– Метод *onCreateView*: Створює та повертає вигляд фрагменту.

– Метод *onMapReady*: Виконує налаштування карти після її готовності.

17. MyApplication:

MyApplication ініціалізує додаток та забезпечує глобальні налаштування.

– Метод *onCreate*: Ініціалізує Google Places API.

18. PlacesApiService:

PlacesApiService - це інтерфейс для отримання інформації про найближчі визначні місця від Google Places API.

– Метод *getNearbyPlaces*: Виконує запит до Google Places API для отримання даних про визначні місця.

19. PlacesModule:

PlacesModule забезпечує ін'єкцію залежностей за допомогою *Dagger*.

– Методи *provide...:* Надають інстанції *PlacesClient*, *OkHttpClient*, *Retrofit*, *PlacesApiService* та *DirectionsApiService*.

20. SplashFragment:

SplashFragment показує заставку з кнопкою входу в систему.

– Поле *binding*: Зв'язує елементи інтерфейсу з фрагментом.

– Поле *animatorSet*: Використовується для анімації заставки.

– Метод *onCreateView*: Створює та повертає вигляд фрагменту.

– Метод *onViewCreated*: Виконує налаштування після створення вигляду, включаючи обробку входу через Google Sign-In.

Взаємодія між класами

Взаємодія між рівнями додатків базується на принципах архітектури MVVM. Фрагменти (*LandmarksListFragment, MapsFragment, SplashFragment*) відповідають за відображення даних і взаємодію з користувачами. *ViewModel* (*LandmarksViewModel*) керує даними та робить їх доступними для перегляду через LiveData. Адаптери (*CityAutocompleteAdapter, LandmarksAdapter*) відповідають за відображення списку даних у сегментах. Служби API (*PlacesApiService, DirectionsApiService*) забезпечують прийом даних з мережі.

Огляд основних класів додатку показує, як вони взаємодіють між собою для забезпечення функціональності пошуку та відображення визначних місць. Використання патерну MVVM забезпечує чітке розділення відповідальності між компонентами додатку, що полегшує підтримку, тестування та розширення коду. Ін'єкція залежностей за допомогою Dagger забезпечує зруч

3.2.3 Оптимізація додатку

Оптимізація мобільних додатків є важливим аспектом для забезпечення швидкості, ефективного використання ресурсів і покращення взаємодії з користувачем. У цьому розділі ми розглянемо методи, які використовувалися для оптимізації програм пошуку та відображення POI.

1. Використання ефективних форматів зображень

Одним із важливих аспектів оптимізації програми є ефективне використання форматів зображень. У нашому додатку всі зображення зберігаються у форматі WebP. Цей формат забезпечує високоякісні зображення зі значно меншими розмірами файлів, ніж традиційні формати, такі як JPEG або PNG.

Формат WebP: Забезпечує краще стиснення без втрати якості. Скорочує час завантаження зображення. Зменшує трафік і використання ресурсів пристрою.

2. Кешування даних

Стратегії кешування даних використовуються для зменшення навантаження на мережу та підвищення продуктивності програми.

– *Кешування зображень*: Бібліотека Picasso використовується для завантаження та зберігання зображень. Це дозволяє уникнути перезавантаження зображень з мережі, що значно покращує швидкість програми.

– *Вимоги до кешу:* Використання Retrofit для кешування HTTP-запитів. Це зменшує навантаження на сервер і дозволяє швидше отримувати дані при повторних запитах.

3. Асинхронна обробка даних

Асинхронна обробка даних забезпечує ефективне використання ресурсів і запобігає блокуванню основного потоку програми.

– *Корутіни:* Використання корутін для асинхронної обробки даних дозволяє виконувати мережеві запити у фоновому режимі, не блокуючи інтерфейс користувача.

– *LiveData:* Використання LiveData для оновлення інтерфейсу користувача в режимі реального часу в міру зміни даних. Це забезпечує ефективне керування станом і запобігає витокам пам’яті.

4. Оптимізація роботи з RecyclerView

Для ефективного відображення списку даних у RecyclerView використовувалися такі методи:

– *Використання DiffUtil:* Використання DiffUtil для оптимального оновлення елементів списку. Це дозволяє оновлювати лише елементи, які змінилися, зменшуючи навантаження на RecyclerView.

– *Віртуалізація:* Віртуалізація елементів списку дозволяє ефективно керувати пам’яттю та підвищувати продуктивність під час відображення великих списків даних.

5. Зменшення розміру додатку

Для зменшення розміру програми та оптимізації завантаження були використані наступні методи:

– *Розбиття на модулі:* Використовуйте Android App Bundle, щоб розділити програму на модулі. Це зменшує розмір файлу APK і завантажує лише необхідні модулі на основі пристрою користувача.

– *Мініфікація та обфускація:* Використовуйте ProGuard, щоб зменшити та приховати код. Це зменшує розмір програми та підвищує її безпеку.

6. Оптимізація роботи з картами

Для ефективної роботи з картами в додатку використовуються такі методи:

– *Кешування карти:* Використання кешування карт, щоб зменшити навантаження на мережу та прискорити візуалізацію карт.

– *Оптимізація візуалізації:* Використання оптимізованихі налаштувань дисплея, щоб пришвидшити роботу картографічних служб.

Оптимізація мобільних додатків має важливе значення для забезпечення ефективності, швидкості та покращення взаємодії з користувачем. У нашому додатку були застосовані різні стратегії, такі як використання формату WebP для зображень, кешування даних, асинхронна обробка, оптимізація роботи з RecyclerView та зменшення розміру додатка. Ці підходи забезпечують ефективне використання ресурсів, скорочення часу завантаження та підвищення загальної продуктивності програми.

3.2.4 Безпека додатку

Безпека мобільного додатка є важливим аспектом, який допомагає захистити дані користувачів і забезпечити довіру до додатка. У цьому розділі ми розглянемо основні методи та практики, які застосовувалися для забезпечення безпеки програм, які відображають і шукають цікаві місця.

1. Захист API ключів

Ключ API використовується для доступу до API Google Places і API Google Directions. Забезпечення безпеки цих ключів має важливе значення для запобігання їх несанкціонованому використанню.

– *Зберігання ключів API:* Ключ API зберігається у файлі AndroidManifest.xml у метаданих програми. Це дозволяє обмежити доступ лише до ключів програми. За бажанням для зберігання конфіденційних даних можна

використовувати безпечне сховище, наприклад EncryptedSharedPreferences або KeyStore.

– *Обмеження доступу:* У Google Cloud Console ви можете встановити обмеження на ключі API, щоб ваша програма використовувала лише їх. Це можна зробити, вказавши обмеження за IP-адресою, реферером або пакетом програми.

2. Аутентифікація користувача

Вхід через Google використовується для автентифікації користувачів у програмі, забезпечуючи надійний і безпечний спосіб входу.

– *Увійти в Google:*

- Вхід користувача відбувається через Google Sign-In, що забезпечує безпечний доступ до облікових записів користувачів. Це реалізовано в класах MainActivity і SplashFragment.

- Підключення використовує OAuth 2.0, який забезпечує високий рівень безпеки, використовуючи маркери доступу.

3. Безпечне зберігання даних

Зберігання даних у програмі також має бути захищено, щоб запобігти несанкціонованому доступу або модифікації.

– *Безпечне зберігання:*

- Використання EncryptedSharedPreferences для зберігання конфіденційних даних користувача.

- Використання кімнати для зберігання даних у зашифрованій локальній базі даних.

4. Обробка помилок

Обробка помилок і особливих ситуацій є важливим аспектом безпеки, оскільки погана обробка може призвести до витоку конфіденційної інформації або збою програми.

– *Логування помилок*: Клас LandmarksViewModel та інші класи використовують журнал помилок за допомогою Log.e. Це дозволяє відстежувати й аналізувати помилки, не розкриваючи конфіденційну інформацію.

– *Обробка винятків:* Використання конструкцію try-catch для обробки винятків, які можуть виникати під час мережевих запитів або інших операцій.

5. Захист мережевих запитів

Запити API мережі вимагають захисту, щоб запобігти перехопленню або модифікації даних.

– *HTTPS:* Використання HTTPS для всіх мережевих запитів, забезпечуючи зашифровану передачу даних.

– *OkHttp Interceptors:* Використання OkHttp Interceptors для перевірки та зміни HTTP-запитів і відповідей. Це дозволяє додатково перевірити достовірність заяв і захистити дані.

6. Обмеження доступу до ресурсів

Обмеження доступу до ресурсів програми запобігає несанкціонованому доступу до конфіденційних даних і функцій.

– *Permissions:* Використання мінімальних дозволів, необхідних для програми. Наприклад, доступ до місцезнаходження потрібен лише тоді, коли програма має працювати.

– *ProGuard:* Використання ProGuard для мінімізації та обфускації коду, ускладнюючи зворотне проектування програми.

Забезпечення безпеки мобільного додатку є важливим аспектом його розробки. Наша програма реалізувала різні методи захисту ключів API, автентифікації користувачів, безпечного зберігання даних, обробки помилок, захисту мережевих запитів і обмеження доступу до ресурсів. Ці заходи забезпечують високий рівень безпеки програми, захищають дані користувачів і забезпечують їх довіру до програми.

3.2.5 Підключення до Firebase та Google Maps Api

Інтеграція Firebase і Google Maps API у мобільний додаток пропонує багато можливостей для розширення його функціональності. У цьому розділі ми

побачимо, як реалізувати підключення до цих служб у програмі за допомогою наданих класів.

1. Підключення до Firebase

Firebase пропонує простий і ефективний спосіб автентифікації користувачів, керування даними в реальному часі та інші корисні функції для мобільних додатків.

Аутентифікація користувача: Додаток реалізує автентифікацію користувача за допомогою входу Google, інтегрованого з автентифікацією Firebase.

MainActivity:

– Основний клас, який відповідає за аутентифікацію користувачів.

– Використовується Google Sign-In для отримання ідентифікаційного токену користувача.

– Токен передається у Firebase для створення користувача або входу.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == 1) {
        val task = GoogleSignIn.getSignedInAccountFromIntent(data)
        try {
            val result = task.getResult(ApiException::class.java)
            val credential =
                GoogleAuthProvider.getCredential(result.idToken,  accessToken: null)
            val auth = FirebaseAuth.getInstance()
            auth.signInWithCredential(credential)
                .addOnCompleteListener { it: Task<AuthResult!>
                    if (it.isSuccessful) {
                        showListFragment()
                    } else {
                        Toast.makeText(
                            context: this,  text: "Authentication failed", Toast.LENGTH_SHORT
                        ).show()
                    }
                }
        } catch (e: ApiException) {
            Toast.makeText(
                context: this,  text: "Error:${e.message}",
                Toast.LENGTH_SHORT
            ).show()
        }
    }
}
```

Рис. 3.14 Код аутентифікації користувача

Застосування Firebase у SplashFragment:

SplashFragment:

– Відповідає за первинну перевірку стану аутентифікації користувача.

– При наявності дійсного облікового запису користувача автоматично переходить до списку визначних місць.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    startAnimation(binding.image)
    val googleSignInInOptions =
        GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
            .requestIdToken( serverClientId: "807068749773-94nm967ar1fbón09rkoub3ektcókvihk.apps.googleusercontent.com")
            .requestEmail()
            .build()
    val googleSignInClient =
        GoogleSignIn.getClient(requireContext(), googleSignInInOptions)
    val account = GoogleSignIn.getLastSignedInAccount(requireContext())
    val activity = requireActivity() as OnAuthLaunch
    if (account == null) {
        showSignInButton()
    } else {
        activity.showListFragment()
    }
    binding.signInButton.setOnClickListener { it: View!
        activity.launch(googleSignInClient.signInIntent)
    }
}
```

Рис. 3.15 Код аутентифікації користувача за допомогою входу Google

2. Підключення до Google Maps API

Google Maps API надає можливість інтеграції картографічних сервісів, відображення маркерів, маршрутів та іншої географічної інформації.

Ініціалізація Google Maps API:

MyApplication:

– Клас додатку, який ініціалізує Google Places API при запуску.

```
override fun onCreate() {
    super.onCreate()
    instance = this
    if (!Places.isInitialized()) {
        Places.initialize(applicationContext, apiKey: "AIzaSyDW4dB2kuQZ_YPI0YCEE6cxrnXKSphpK2I")
    }
}
```

Рис. 3.16 Код ініціації Google Places API при запуску

Використання Google Places API

LandmarksListFragment:

– Використовує Google Places API для автозаповнення назв міст та отримання місць.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    if (!Places.isInitialized()) {
        Places.initialize(requireContext(), "AIzaSyDW4dB2kuQZ_YPI0YCEE6cxrnXKSphpK2I")
    }
    placesClient = Places.createClient(requireContext())
```

Рис. 3.17 Код аутентифікації для автозаповнення назв міст та отримання місць.

Відображення карти та маркерів

MapsFragment:

– Відповідає за відображення карти, маркерів місць та маршрутів між ними.

```
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap

    map.uiSettings.isZoomControlsEnabled = true

    context?.let { ctx ->
        map.setMapStyle(MapStyleOptions.loadRawResourceStyle(ctx, R.raw.style_json))
    } ?: Log.e( tag: "MapsFragment",  msg: "Context is null, cannot set map style")

    arguments?.let { bundle ->
        val cityLatLng = bundle.getParcelable<LatLng>( key: "cityLatLng")
        val landmarksLatLng = bundle.getParcelableArrayList<LatLng>( key: "landmarksLatLng")

        cityLatLng?.let { it: LatLng
            val cityMarkerOptions = MarkerOptions()
                .position(it)
                .title("City")
                .icon(BitmapDescriptorFactory.fromResource(R.drawable.map_marker_city1))
            map.addMarker(cityMarkerOptions)
            map.animateCamera(CameraUpdateFactory.newLatLngZoom(it,  zoom: 13.5F))
        }

        landmarksLatLng?.forEach { latLng ->
            val landmarkMarkerOptions = MarkerOptions()
                .position(latLng)
                .icon(BitmapDescriptorFactory.fromResource(R.drawable.map_marker_icons))
            map.addMarker(landmarkMarkerOptions.position(latLng))
        } ^let
    }
```

Рис. 3.18 Код котрий відповідає за відображення карти, маркерів місць та маршрутів між ними

Отримання маршрутів

MapsFragment:

– Використовує Google Directions API для отримання маршрутів між місцями та їх відображення на карті.

```
lifecycleScope.launch { this: CoroutineScope
    val apiKey = "AIzaSyDW4dB2kuQZ_YPI0YCEE6cxrnXKSphpK2I"
    if (cityLatLng != null && landmarksLatLng != null && landmarksLatLng.isNotEmpty()) {
        try {
            val origin = "${cityLatLng.latitude},${cityLatLng.longitude}"
            val destination = origin
            val waypoints = landmarksLatLng.joinToString(separator = "|") { latLng ->
                "${latLng.latitude},${latLng.longitude}"
            }
            val response = directionsApiService.getComplexRoute(
                origin,
                destination,
                waypoints,
                 mode: "walking",
                apiKey
            )
            if (response.isSuccessful) {
                val polylinePoints =
                    response.body()?.routes?.firstOrNull()?.overview_polyline?.points
                        ?: return@launch
                val decodedPath = PolyUtil.decode(polylinePoints)
                withContext(Dispatchers.Main) { this: CoroutineScope
                    map.addPolyline(
                        PolylineOptions().addAll(decodedPath)
                            .color(routeColor)
                            .width(12f)
                    )
                }
            } else {
                Log.e(
                     tag: "MapsFragment",
                     msg: "Failed to fetch route: ${response.errorBody()?.string()}"
                )
            }
        } catch (e: Exception) {
            Log.e( tag: "MapsFragment",  msg: "Exception when fetching route", e)
        }
```

Рис. 3.19 Код котрий відповідає за отримання маршрутів між місцями та їх відображення на карті.

Підключення до Firebase та Google Maps API надає додатку широкі можливості для аутентифікації користувачів та інтеграції картографічних сервісів. Використання Firebase забезпечує надійну аутентифікацію, тоді як Google Maps API дозволяє відображати географічну інформацію та маршрути. Ці інтеграції

значно покращують функціональність додатку та забезпечують зручний користувацький досвід.

3.2.6 Графічна складова додатку

Графічні компоненти відіграють важливу роль у забезпеченні привабливості та зручності використання мобільного додатку. У цьому розділі ми розглянемо основні компоненти графічного інтерфейсу користувача, які використовуються в програмі для пошуку та відображення цікавих місць на основі структури ресурсу, наданої на зображенні.

1. Ресурси зображень (drawable)

Папка drawable містить різноманітні графічні ресурси, які використовуються для відображення елементів інтерфейсу, таких як іконки, маркери на карті та фонові зображення.

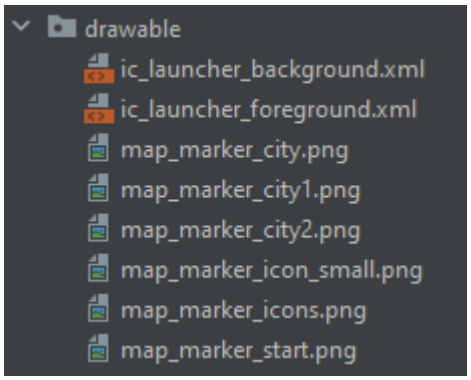


Рис. 3.20 Вміст папки drawable

Іконки лаунчера:

ic_launcher_background.xml

ic_launcher_foreground.xml

Ці файли визначають іконку додатку, що відображається на домашньому екрані пристрою.

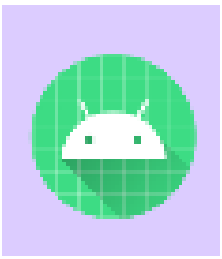


Рис. 3.21 Іконка лаунчеру

Маркери на карті:

map_marker_city.webp

map_marker_city1.webp

map_marker_city2.webp

map_marker_icon_small.webp

map_marker_icons.webp

map_marker_start.webp

Ці зображення використовуються для відображення різних типів маркерів на карті, що позначають визначні місця та інші важливі точки.

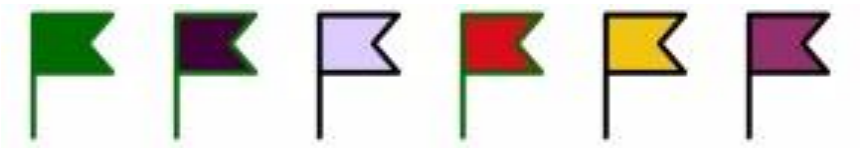


Рис. 3.22 зображення маркерів

2. Макети (layout)

Папка layout містить XML-файли, які визначають структуру та компонування інтерфейсу користувача.

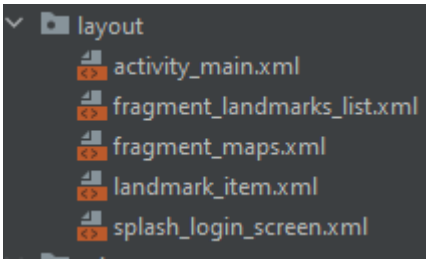


Рис. 3.23 Вміст папки layout

Основний макет активності:

activity_main.xml

Визначає основний макет для головної активності додатку.

Макети фрагментів:

fragment_landmarks_list.xml

fragment_maps.xml

Ці файли визначають макети для фрагментів, які відображають список визначних місць та карту відповідно.

Рис. 3.24 Макети для фрагментів

Макет заставки:

splash_login_screen.xml

Визначає макет для екрану заставки з кнопкою входу в систему.

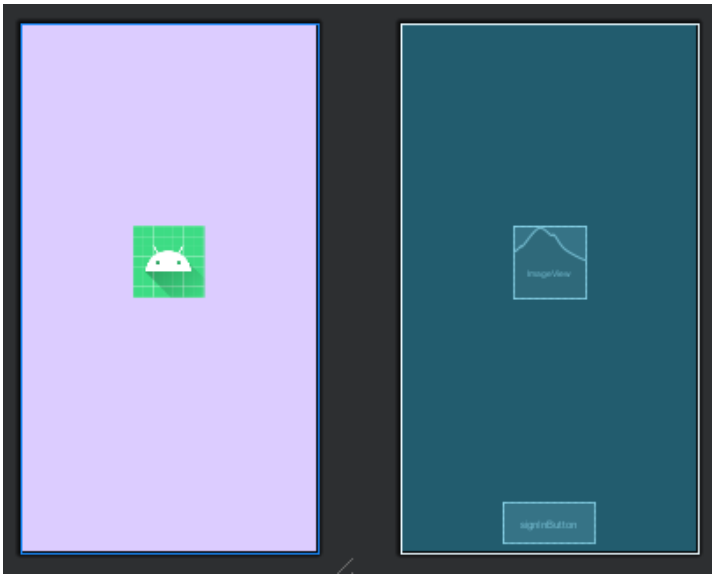


Рис. 3.25 Макет для екрану заставки

Макет елемента списку:

landmark_item.xml

Визначає структуру для елементів списку визначних місць, що відображаються в RecyclerView.

3. Ресурси для різних розмірів екрану (mipmap)

Папка mipmap містить іконки додатку у різних розмірах для підтримки різних роздільних здатностей екрану.

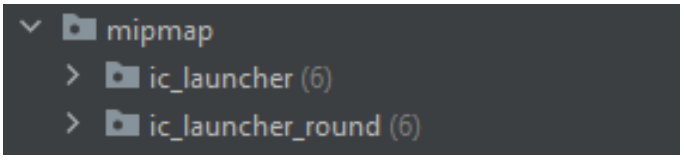


Рис. 3.26 Вміст папки mipmap

Іконки лаунчера:

ic_launcher

ic_launcher_round

Ці файли містять іконки додатку у різних розмірах для коректного відображення на різних пристроях.

4. Ресурси стилів (raw)

Папка raw містить додаткові ресурси, які можуть використовуватися в додатку.

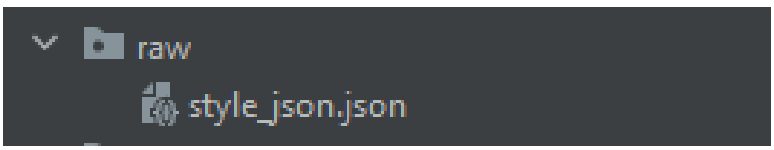


Рис. 3.27 Вміст папки raw

Файл стилю карти:

style_json.json

Визначає стиль для відображення карти в Google Maps, налаштовуючи кольори та інші параметри відображення.

5. Ресурси значень (values)

Папка values містить XML-файли з визначенням різних значень, які використовуються в додатку, таких як кольори, рядки та теми.

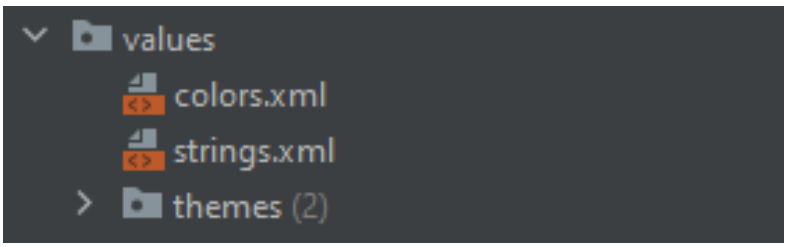


Рис. 3.28 Вміст папки values

Файл кольорів:

colors.xml

Визначає кольорову палітру для додатку.

Файл рядків:

strings.xml

Містить текстові рядки, які використовуються в додатку, що полегшує локалізацію та підтримку багатомовності.

Файли тем:

themes.xml

Визначає теми для додатку, включаючи кольорові схеми та стилі компонентів інтерфейсу.

6. Ресурси XML

Папка xml містить додаткові налаштування у форматі XML.

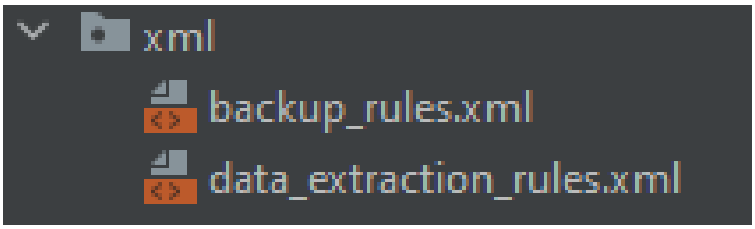


Рис. 3.29 Вміст папки xml

Файл правил резервного копіювання:

backup_rules.xml

Визначає правила резервного копіювання даних додатку.

Файл правил вилучення даних:

data_extraction_rules.xml

Містить правила для вилучення даних додатку.

Графічна складова програми забезпечує привабливість і зручність використання. Використання різноманітних ресурсів, таких як значки, маркери, макети та стилі, дозволяє створювати інтуїтивно зрозумілі та інтерактивні інтерфейси користувача. Правильна організація цих ресурсів у відповідних папках забезпечить легке обслуговування та розширення програми.

ВИСНОВКИ

Під час дослідження та розробки посібника з мобільного додатку було проведено аналіз різних архітектурних підходів, вивчено вплив архітектурних рішень на продуктивність, швидкість і рівень споживання ресурсів пристрою, а також питання безпеки. Вважається. Особливу увагу було приділено використанню мови програмування Kotlin, її перевагам, фреймворкам, бібліотекам та інтеграції з Firebase та Google Maps API сервісами.

Огляд існуючих архітектурних методів

Ми детально проаналізували існуючі архітектурні підходи для мобільних додатків. Шаблони MVVM і MVP вважаються найпопулярнішими завдяки їх гнучкості, простоті обслуговування та тестування. Також розглядаються тенденції в архітектурі мобільних додатків, що свідчить про зростання популярності реактивного програмування та компонентно-орієнтованих підходів.

Вплив архітектурних рішень на продуктивність і ресурсоємність

Визнається, що правильний вибір архітектури може значно підвищити продуктивність і знизити витрати на ресурси. Архітектурні рішення, які підтримують асинхронну обробку даних і кешування, дозволяють знизити навантаження на процесор і мережу, забезпечуючи коректну роботу додатків навіть на пристроях з механізмом обмежених ресурсів.

Захищена архітектура мобільних додатків

Безпека мобільних додатків є важливою. Було розглянуто різні аспекти безпеки, включаючи захист ключів API, автентифікацію користувачів через Firebase, безпечне зберігання даних і захист мережевих запитів. Інтеграція входу в Google і використання шифрування для зберігання конфіденційних даних дозволяє забезпечити високий рівень безпеки програми.

Використання Kotlin для розробки мобільних додатків

Kotlin зарекомендувала себе як сучасна та ефективна мова програмування для мобільних додатків. Його синтаксична зручність, розширюваність та інтеграція з існуючими бібліотеками Java роблять його ідеальним вибором для розробки на

платформі Android. Переваги Kotlin, такі як безпека типу, підтримка співпрограми для асинхронного програмування та висока продуктивність.

Інтеграція з Firebase API і Google Maps

Інтеграція з Firebase дає можливість реалізувати аутентифікацію довіреного користувача, зберігати дані в реальному часі та інші корисні функції. Використання Google Maps API допомагає відображати карти, маркери та маршрути, значно покращуючи функціональність програми.

Графічна складова програми

Ретельно розроблений графічний компонент програми забезпечує інтуїтивно зрозумілий і привабливий інтерфейс для користувачів. Використання різноманітних графічних ресурсів, таких як іконки, маркери та стилі, допомогло створити функціональний та естетично привабливий інтерфейс.

Загалом, проведене дослідження та розробка мобільного додатку-гід показали, що правильний вибір архітектурних підходів, мови програмування та інструментів для інтеграції з зовнішніми сервісами є ключовими факторами успіху. Використання Kotlin, інтеграція з Firebase та Google Maps API, а також забезпечення високого рівня безпеки та оптимізації дозволили створити продуктивний, надійний та зручний у використанні додаток.

ПЕРЕЛІК ПОСИЛАНЬ

1. Android Studio documentation
   https://developer.android.com/community
2. Kotlin documentation
   https://kotlinlang.org/docs/home.html
3. Firebase documentation
   https://firebase.google.com/docs
4. Google Maps Api documentation
   https://developers.google.com/maps/documentation/javascript?hl=ru
5. User interface design patterns
   https://www.interaction-design.org/literature/topics/ui-design-patterns#:~:text=User%20interface%20(UI)%20design%20patterns%20are%20reusable%2Frecurring%20components,problems%20in%20user%20interface%20design.&text=For%20example%2C%20the%20breadcrumbs%20design,the%20specific%20context%20of%20use.
6. Mobile program architectures
   https://decode.agency/article/mobile-app-architecture/
7. Критерії хорошої архітектури. Ієрархія принципів проектування
   https://org2.knuba.edu.ua/pluginfile.php/56721/mod_resource/content/1/Lek_4_%D0%90%D0%9F%D0%9F%D0%97.pdf

ДОДАТОК А. Лістинги програм

Лістинг 1. Код класу «CityAutocompleteAdapter»:

```
import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ArrayAdapter
import android.widget.Filter
import android.widget.Filterable
import android.widget.TextView
import com.google.android.libraries.places.api.model.AutocompletePrediction
import com.google.android.libraries.places.api.net.PlacesClient
import
com.google.android.libraries.places.api.net.FindAutocompletePredictionsRequest
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.launch
import kotlinx.coroutines.tasks.await

class CityAutocompleteAdapter(
    context: Context,
    private val placesClient: PlacesClient,
    private val coroutineScope: CoroutineScope
) : ArrayAdapter<AutocompletePrediction>(context,
android.R.layout.simple_dropdown_item_1line),
    Filterable {

    private val predictions = mutableListOf<AutocompletePrediction>()

    override fun getCount(): Int = predictions.size

    override fun getItem(position: Int): AutocompletePrediction = predictions[position]

    override fun getView(position: Int, convertView: View?, parent: ViewGroup): View
{
        val view = convertView ?: LayoutInflater.from(context)
            .inflate(android.R.layout.simple_dropdown_item_1line, parent, false)
        val item = getItem(position)
        (view.findViewById(android.R.id.text1) as TextView).text =
            item.getPrimaryText(null).toString()
        return view
    }

    override fun getFilter(): Filter {
```

```
        return object : Filter() {
            override fun performFiltering(constraint: CharSequence?): FilterResults {
                val filterResults = FilterResults()
                if (!constraint.isNullOrBlank()) {
                    coroutineScope.launch {
                        val request = FindAutocompletePredictionsRequest.builder()
                            .setQuery(constraint.toString())
                            .build()

                        val response =
placesClient.findAutocompletePredictions(request).await()
                        predictions.clear()
                        predictions.addAll(response.autocompletePredictions)
                        filterResults.values = predictions
                        filterResults.count = predictions.size
                        publishResults(constraint, filterResults)
                    }
                }
                return filterResults
            }

            override fun publishResults(constraint: CharSequence?, results: FilterResults?) {
                if (results != null && results.count > 0) {
                    notifyDataSetChanged()
                } else {
                    notifyDataSetInvalidated()
                }
            }
        }
    }
}
```

Лістинг 2. Код «Data.kt»:

```
import com.google.gson.annotations.SerializedName

data class Landmark(val id: String, val name: String, val imageUrl: String)
data class PlacesResponse(val results: List<Results>)
data class Results(val geometry: Geometry, val photos: List<Photos>, val name: String)
data class Geometry(val location: Location)
data class Location(val lat: Double, val lng: Double)
data class Photos(@SerializedName("photo_reference") val photoReference: String? =
null)
```

```
data class DirectionsResponse(val routes: List<Route>)

data class Route(val overview_polyline: Polyline)

data class Polyline(val points: String)
```

Лістинг 3. Код інтерфейсу «DirectionsApiService»:

```
import retrofit2.Response
import retrofit2.http.GET
import retrofit2.http.Query

interface DirectionsApiService {
    @GET("maps/api/directions/json")
    suspend fun getComplexRoute(
        @Query("origin") origin: String,
        @Query("destination") destination: String,
        @Query("waypoints") waypoints: String,
        @Query("mode") mode: String = "walking",
        @Query("key") apiKey: String
    ): Response<DirectionsResponse>
}
```

Лістинг 4. Код класу «LandmarksAdapter»:

```
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ImageView
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.squareup.picasso.Picasso

class LandmarksAdapter(private var landmarks: List<Landmark>) :
    RecyclerView.Adapter<LandmarksAdapter.LandmarkViewHolder>() {

    class LandmarkViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        fun bind(landmark: Landmark) {
            itemView.findViewById<TextView>(R.id.landmarkName).text =
landmark.name
            Picasso.get().load(landmark.imageUrl)
                .into(itemView.findViewById<ImageView>(R.id.landmarkImage))
        }
```

```
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
LandmarkViewHolder {
        val view =
            LayoutInflater.from(parent.context).inflate(R.layout.landmark_item, parent,
false)
        return LandmarkViewHolder(view)
    }

    override fun onBindViewHolder(holder: LandmarkViewHolder, position: Int) {
        holder.bind(landmarks[position])
    }

    override fun getItemCount() = landmarks.size

    fun updateData(newLandmarks: List<Landmark>) {
        Log.d("LandmarksAdapter", "Updating data with ${newLandmarks.size}
landmarks")
        this.landmarks = newLandmarks
        notifyDataSetChanged()
    }
}
```

Лістинг 5. Код класу «LandmarksListFragment»:

```
import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import androidx.lifecycle.lifecycleScope
import androidx.recyclerview.widget.LinearLayoutManager
import com.example.afinal.databinding.FragmentLandmarksListBinding
import com.google.android.libraries.places.api.Places
import com.google.android.libraries.places.api.model.Place
import com.google.android.libraries.places.api.net.FetchPlaceRequest
import com.google.android.libraries.places.api.net.PlacesClient
import dagger.hilt.android.AndroidEntryPoint
import javax.inject.Inject

@AndroidEntryPoint
```

```
class LandmarksListFragment : Fragment() {

    private lateinit var binding: FragmentLandmarksListBinding
    private val viewModel: LandmarksViewModel by viewModels()

    @Inject
    lateinit var placesClient: PlacesClient

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentLandmarksListBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        if (!Places.isInitialized()) {
            Places.initialize(requireContext(), getString(R.string.google_maps_key))
        }
        placesClient = Places.createClient(requireContext())

        binding.landmarksRecyclerView.layoutManager = LinearLayoutManager(context)
        val adapter = LandmarksAdapter(listOf())
        binding.landmarksRecyclerView.adapter = adapter


        val autocompleteAdapter = CityAutocompleteAdapter(
            requireContext(),
            placesClient,
            viewLifecycleOwner.lifecycleScope
        )
        binding.cityInput.setAdapter(autocompleteAdapter)
        binding.cityInput.setOnItemClickListener { _, _, position, _ ->
            val selectedPrediction = autocompleteAdapter.getItem(position)
            val placeId = selectedPrediction.placeId

            val placeFields = listOf(Place.Field.LAT_LNG)
            val request = FetchPlaceRequest.newInstance(placeId, placeFields)

            placesClient.fetchPlace(request).addOnSuccessListener { response ->
                val place = response.place
```

```
            place.latLng?.let {
                viewModel.loadLandmarks(it)
            }
        }.addOnFailureListener { exception ->
            Log.e("LandmarksListFragment", "Error fetching place details:
${exception.message}")
        }
        val cityName = selectedPrediction.getPrimaryText(null).toString()
        binding.cityInput.setText(cityName)
    }

    viewModel.landmarks.observe(viewLifecycleOwner) { landmarks ->
        adapter.updateData(landmarks)
    }

    viewModel.cityLatLng.observe(viewLifecycleOwner) { cityLatLng ->
        viewModel.landmarksLatLng.observe(viewLifecycleOwner) {
landmarksLatLng ->
            binding.showMapButton.setOnClickListener {
                val mapsFragment = MapsFragment().apply {
                    arguments = Bundle().apply {
                        putParcelable("cityLatLng", cityLatLng)
                        putParcelableArrayList("landmarksLatLng",
ArrayList(landmarksLatLng))
                    }
                }
                requireActivity().supportFragmentManager.beginTransaction()
                    .replace(R.id.container, mapsFragment)
                    .addToBackStack(null)
                    .commit()
            }
        }
    }
}
```

Лістинг 6. Код класу «LandmarksViewModel»:

```
import android.content.Context
import android.content.pm.PackageManager
import android.util.Log
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
```

```
import com.google.android.gms.maps.model.LatLng
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class LandmarksViewModel @Inject constructor(
    private val placesApiService: PlacesApiService,
    private val context: Context
) : ViewModel() {

    private val _landmarks = MutableLiveData<List<Landmark>>()
    val landmarks: LiveData<List<Landmark>> = _landmarks

    private val _cityLatLng = MutableLiveData<LatLng?>()
    val cityLatLng: LiveData<LatLng?> = _cityLatLng

    private val _landmarksLatLng = MutableLiveData<List<LatLng>>()
    val landmarksLatLng: LiveData<List<LatLng>> = _landmarksLatLng

    fun loadLandmarks(cityLatLng: LatLng?) {
        this._cityLatLng.value = cityLatLng

        cityLatLng?.let { latLng ->
            val location = "${latLng.latitude},${latLng.longitude}"
            val apiKey = getApiKey()
            Log.d("LandmarksViewModel", "Loading landmarks for location: $location")

            viewModelScope.launch {
                try {
                    val response = placesApiService.getNearbyPlaces(
                        location = location,
                        radius = 2000,
                        type = "tourist_attraction",
                        apiKey = apiKey
                    )
                    Log.d("LandmarksViewModel", "Raw API Response:
${response.body()}")

                    if (response.isSuccessful && response.body() != null) {
                        val places = response.body()!!.results
                        Log.d("LandmarksViewModel", "API Response: $places")

                        val landmarks = places.take(15).map { place ->
                            val photoReference = place.photos.firstOrNull()?.photoReference
```

```
                    val photoUrl = photoReference?.let {

"https://maps.googleapis.com/maps/api/place/photo?maxwidth=400&photoreference=$it&key=$apiKey"
                    } ?: ""

                    Landmark(
                        id = place.name.hashCode().toString(),
                        name = place.name,
                        imageUrl = photoUrl
                    )
                }
                _landmarks.postValue(landmarks)

                _landmarksLatLng.postValue(
                    places.take(15)
                        .map { LatLng(it.geometry.location.lat, it.geometry.location.lng) })
            } else {
                Log.e(
                    "LandmarksViewModel",
                    "Failed to fetch landmarks: ${response.errorBody()?.string()}"
                )
                _landmarks.postValue(emptyList())
            }
        } catch (e: Exception) {
            Log.e("LandmarksViewModel", "Exception when fetching landmarks", e)
            _landmarks.postValue(emptyList())
        }
    }

    private fun getApiKey(): String {
        val applicationInfo = context.packageManager.getApplicationInfo(
            context.packageName,
            PackageManager.GET_META_DATA
        )
        return applicationInfo.metaData.getString("com.google.android.geo.API_KEY") ?: ""
    }
}
```

Лістинг 7. Код «MainActivity.kt»:

```
import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Toast
import com.google.android.gms.auth.api.signin.GoogleSignIn
import com.google.android.gms.common.api.ApiException
import com.google.firebase.auth.FirebaseAuth
import com.google.firebase.auth.GoogleAuthProvider
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class MainActivity : AppCompatActivity(), OnAuthLaunch {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        if (savedInstanceState == null) {
            if (supportFragmentManager.findFragmentById(R.id.container) == null) {
                showListFragment()
            }
        }
    }

    override fun launch(intent: Intent) {
        startActivityForResult(intent, 1)
    }

    override fun showListFragment() {
        supportFragmentManager.beginTransaction()
            .replace(R.id.container, LandmarksListFragment())
            .commit()
    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        super.onActivityResult(requestCode, resultCode, data)
        if (requestCode == 1) {
            val task = GoogleSignIn.getSignedInAccountFromIntent(data)
            try {
                val result = task.getResult(ApiException::class.java)
                val credential =
                    GoogleAuthProvider.getCredential(result.idToken, null)
                val auth = FirebaseAuth.getInstance()
                auth.signInWithCredential(credential)
```

```
                .addOnCompleteListener {
                    if (it.isSuccessful) {
                        showListFragment()
                    } else {
                        Toast.makeText(
                            this, "Authentication failed", Toast.LENGTH_SHORT
                        ).show()
                    }
                }
            } catch (e: ApiException) {
                Toast.makeText(
                    this, "Error:${e.message}",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
    }
}

interface OnAuthLaunch {
    fun launch(intent: Intent)
    fun showListFragment()
}
```

Лістинг 8. Код класу «MapsFragment»:

```
import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.core.content.ContextCompat
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import com.google.android.gms.maps.CameraUpdateFactory
import com.google.android.gms.maps.GoogleMap
import com.google.android.gms.maps.OnMapReadyCallback
import com.google.android.gms.maps.SupportMapFragment
import com.google.android.gms.maps.model.BitmapDescriptorFactory
import com.google.android.gms.maps.model.LatLng
import com.google.android.gms.maps.model.MapStyleOptions
import com.google.android.gms.maps.model.MarkerOptions
import com.google.android.gms.maps.model.PolylineOptions
import com.google.maps.android.PolyUtil
```

```
import dagger.hilt.android.AndroidEntryPoint
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import javax.inject.Inject

@AndroidEntryPoint
class MapsFragment : Fragment(), OnMapReadyCallback {

    @Inject
    lateinit var directionsApiService: DirectionsApiService

    private lateinit var map: GoogleMap

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {

        val view = inflater.inflate(R.layout.fragment_maps, container, false)
        val mapFragment = childFragmentManager.findFragmentById(R.id.map) as
SupportMapFragment?
        mapFragment?.getMapAsync(this)
        return view
    }

    override fun onMapReady(googleMap: GoogleMap) {
        map = googleMap

        map.uiSettings.isZoomControlsEnabled = true

        context?.let { ctx ->
            map.setMapStyle(MapStyleOptions.loadRawResourceStyle(ctx,
R.raw.style_json))
        } ?: Log.e("MapsFragment", "Context is null, cannot set map style")

        arguments?.let { bundle ->
            val cityLatLng = bundle.getParcelable<LatLng>("cityLatLng")
            val landmarksLatLng =
bundle.getParcelableArrayList<LatLng>("landmarksLatLng")

            cityLatLng?.let {
                val cityMarkerOptions = MarkerOptions()
                    .position(it)
                    .title("City")
```

```
.icon(BitmapDescriptorFactory.fromResource(R.drawable.map_marker_city1))
            map.addMarker(cityMarkerOptions)
            map.animateCamera(CameraUpdateFactory.newLatLngZoom(it, 13.5F))
        }

        landmarksLatLng?.forEach { latLng ->
            val landmarkMarkerOptions = MarkerOptions()
                .position(latLng)

.icon(BitmapDescriptorFactory.fromResource(R.drawable.map_marker_icons))
            map.addMarker(landmarkMarkerOptions.position(latLng))
        }
    }

    val cityLatLng = arguments?.getParcelable<LatLng>("cityLatLng")
    val landmarksLatLng =
arguments?.getParcelableArrayList<LatLng>("landmarksLatLng")
    val routeColor = ContextCompat.getColor(requireContext(),
R.color.magenta_haze)

    lifecycleScope.launch {
        val apiKey = getString(R.string.google_maps_key)
        if (cityLatLng != null && landmarksLatLng != null &&
landmarksLatLng.isNotEmpty()) {
            try {
                val origin = "${cityLatLng.latitude},${cityLatLng.longitude}"
                val destination = origin
                val waypoints = landmarksLatLng.joinToString(separator = "|") { latLng ->
                    "${latLng.latitude},${latLng.longitude}"
                }
                val response = directionsApiService.getComplexRoute(
                    origin,
                    destination,
                    waypoints,
                    "walking",
                    apiKey
                )
                if (response.isSuccessful) {
                    val polylinePoints =
                        response.body()?.routes?.firstOrNull()?.overview_polyline?.points
                            ?: return@launch
                    val decodedPath = PolyUtil.decode(polylinePoints)
                    withContext(Dispatchers.Main) {
                        map.addPolyline(
```

```
                        PolylineOptions().addAll(decodedPath)
                            .color(routeColor)
                            .width(12f)
                    )
                }
            } else {
                Log.e(
                    "MapsFragment",
                    "Failed to fetch route: ${response.errorBody()?.string()}"
                )
            }
        } catch (e: Exception) {
            Log.e("MapsFragment", "Exception when fetching route", e)
        }
    }
  }
}
```

Лістинг 9. Код класу «MyApplication»:

```
import android.app.Application
import com.google.android.libraries.places.api.Places
import dagger.hilt.android.HiltAndroidApp

@HiltAndroidApp
class MyApplication : Application() {
  override fun onCreate() {
    super.onCreate()
    instance = this
    if (!Places.isInitialized()) {
      Places.initialize(applicationContext,
"AIzaSyDW4dB2kuQZ_YPI0YCEE6cxrnXKSphpK2I")
    }
  }

  companion object {
    lateinit var instance: MyApplication
      private set
  }
}
```

Лістинг 10. Код інтерфейсу «PlacesApiService»:

```
import retrofit2.Response
import retrofit2.http.GET
import retrofit2.http.Query

interface PlacesApiService {
    @GET("maps/api/place/nearbysearch/json?")
    suspend fun getNearbyPlaces(
        @Query("location") location: String,
        @Query("radius") radius: Int = 1000,
        @Query("type") type: String = "tourist_attraction",
        @Query("key") apiKey: String,
        @Query("language") language: String = "uk"
    ): Response<PlacesResponse>

}
```

Лістинг 11. Код класу «MyApplication»:

```
import retrofit2.Response
import retrofit2.http.GET
import retrofit2.http.Query

interface PlacesApiService {
    @GET("maps/api/place/nearbysearch/json?")
    suspend fun getNearbyPlaces(
        @Query("location") location: String,
        @Query("radius") radius: Int = 1000,
        @Query("type") type: String = "tourist_attraction",
        @Query("key") apiKey: String,
        @Query("language") language: String = "uk"
    ): Response<PlacesResponse>

}
```

Лістинг 12. Код класу «SplashFragment»:

```
import android.animation.AnimatorSet
import android.animation.ObjectAnimator
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ImageView
import androidx.fragment.app.Fragment
```

```
import com.example.afinal.databinding.SplashLoginScreenBinding
import com.google.android.gms.auth.api.signin.GoogleSignIn
import com.google.android.gms.auth.api.signin.GoogleSignInOptions
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class SplashFragment : Fragment() {

    private lateinit var binding: SplashLoginScreenBinding

    private val animatorSet = AnimatorSet()

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = SplashLoginScreenBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        startAnimation(binding.image)
        val googleSignInInOptions =
            GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
                .requestIdToken("807068749773-
94nm967ar1fb6n09rkoub3ektc6kvihk.apps.googleusercontent.com")
                .requestEmail()
                .build()
        val googleSignInClient =
            GoogleSignIn.getClient(requireContext(), googleSignInInOptions)
        val account = GoogleSignIn.getLastSignedInAccount(requireContext())
        val activity = requireActivity() as OnAuthLaunch
        if (account == null) {
            showSignInButton()
        } else {
            activity.showListFragment()
        }
        binding.signInButton.setOnClickListener {
            activity.launch(googleSignInClient.signInIntent)
        }
    }

    private fun showSignInButton() {
```

```
        binding.signInButton.visibility = View.VISIBLE
        animatorSet.cancel()
    }

    private fun startAnimation(image: ImageView) {
        val scaleXAnimation = ObjectAnimator.ofFloat(
            image, View.SCALE_X, 0.5f,
            1f
        )
        scaleXAnimation.repeatMode = ObjectAnimator.REVERSE
        scaleXAnimation.repeatCount = ObjectAnimator.INFINITE
        val scaleYAnimation = ObjectAnimator.ofFloat(
            image, View.SCALE_Y, 0.5f,
            1f
        )
        scaleYAnimation.repeatMode = ObjectAnimator.REVERSE
        scaleYAnimation.repeatCount = ObjectAnimator.INFINITE
        animatorSet.playTogether(scaleXAnimation, scaleYAnimation)
        animatorSet.duration = 1000
        animatorSet.start()
    }
}
```

Лістинг 13. Код xml файлу «ic_launcher_background»:

```
<?xml version="1.0" encoding="utf-8"?>
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="108dp"
    android:height="108dp"
    android:viewportWidth="108"
    android:viewportHeight="108">
    <path
        android:fillColor="#3DDC84"
        android:pathData="M0,0h108v108h-108z" />
    <path
        android:fillColor="#00000000"
        android:pathData="M9,0L9,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M19,0L19,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
```

```
    android:fillColor="#00000000"
    android:pathData="M29,0L29,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M39,0L39,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M49,0L49,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M59,0L59,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M69,0L69,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M79,0L79,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M89,0L89,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M99,0L99,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,9L108,9"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
```

```
    android:fillColor="#00000000"
    android:pathData="M0,19L108,19"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,29L108,29"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,39L108,39"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,49L108,49"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,59L108,59"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,69L108,69"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,79L108,79"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,89L108,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,99L108,99"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
```

```
    android:fillColor="#00000000"
    android:pathData="M19,29L89,29"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,39L89,39"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,49L89,49"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,59L89,59"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,69L89,69"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,79L89,79"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M29,19L29,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M39,19L39,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M49,19L49,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
```

```
        android:fillColor="#00000000"
        android:pathData="M59,19L59,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M69,19L69,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M79,19L79,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
</vector>
```

Лістинг 14. Код xml файлу «ic_launcher_foreground»:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:aapt="http://schemas.android.com/aapt"
    android:width="108dp"
    android:height="108dp"
    android:viewportWidth="108"
    android:viewportHeight="108">
    <path android:pathData="M31,63.928c0,0 6.4,-11 12.1,-13.1c7.2,-2.6 26,-1.4 26,-
1.4l38.1,38.1L107,108.928l-32,-1L31,63.928z">
        <aapt:attr name="android:fillColor">
            <gradient
                android:endX="85.84757"
                android:endY="92.4963"
                android:startX="42.9492"
                android:startY="49.59793"
                android:type="linear">
                <item
                    android:color="#44000000"
                    android:offset="0.0" />
                <item
                    android:color="#00000000"
                    android:offset="1.0" />
            </gradient>
        </aapt:attr>
    </path>
    <path
        android:fillColor="#FFFFFF"
        android:fillType="nonZero"
```

```
        android:pathData="M65.3,45.828l3.8,-6.6c0.2,-0.4 0.1,-0.9 -0.3,-1.1c-0.4,-0.2 -0.9,-0.1 -1.1,0.3l-3.9,6.7c-6.3,-2.8 -13.4,-2.8 -19.7,0l-3.9,-6.7c-0.2,-0.4 -0.7,-0.5 -1.1,-0.3C38.8,38.328 38.7,38.828 38.9,39.228l3.8,6.6C36.2,49.428 31.7,56.028 31,63.928h46C76.3,56.028 71.8,49.428 65.3,45.828zM43.4,57.328c-0.8,0 -1.5,-0.5 -1.8,-1.2c-0.3,-0.7 -0.1,-1.5 0.4,-2.1c0.5,-0.5 1.4,-0.7 2.1,-0.4c0.7,0.3 1.2,1 1.2,1.8C45.3,56.528 44.5,57.328 43.4,57.328L43.4,57.328zM64.6,57.328c-0.8,0 -1.5,-0.5 -1.8,-1.2s-0.1,-1.5 0.4,-2.1c0.5,-0.5 1.4,-0.7 2.1,-0.4c0.7,0.3 1.2,1 1.2,1.8C66.5,56.528 65.6,57.328 64.6,57.328L64.6,57.328z"
        android:strokeWidth="1"
        android:strokeColor="#00000000" />
</vector>
```

Лістинг 15. Код xml файлу «activity_main»:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/container"
        android:name="com.example.afinal.SplashFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Лістинг 16. Код xml файлу «fragment_landmarks_list»:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/periwinkle">

    <AutoCompleteTextView
        android:id="@+id/city_input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint_enter_city"
```

```
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:textAlignment="center"
        android:textColor="@color/magenta_haze"
        android:textSize="24sp"
        android:textStyle="bold"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/landmarksRecyclerView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/city_input" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/showMapButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:src="@android:drawable/ic_dialog_map"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Лістинг 17. Код xml файлу «fragment_maps»:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.SupportMapFragment"
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Лістинг 18. Код xml файлу «landmark_item»:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/periwinkle"
    android:padding="4dp">

    <ImageView
        android:id="@+id/landmarkImage"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:scaleType="centerCrop"
        android:paddingBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/landmarkName"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:textColor="@color/magenta_haze"
        android:textSize="16sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/landmarkImage"
        app:layout_constraintTop_toTopOf="parent" />

    <View
        android:id="@+id/divider"
        android:layout_width="409dp"
        android:layout_height="2dp"
        android:layout_marginTop="8dp"
        android:background="?android:attr/listDivider"
```

```
        android:outlineAmbientShadowColor="@color/black"
        android:outlineSpotShadowColor="@color/black"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/landmarkImage"
        tools:layout_editor_absoluteX="1dp"
        tools:layout_editor_absoluteY="74dp" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Лістинг 19. Код xml файлу «splash_login_screen»:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/periwinkle">

    <ImageView
        android:id="@+id/image"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@mipmap/ic_launcher"
        app:layout_constraintBottom_toTopOf="@+id/signInButton"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <com.google.android.gms.common.SignInButton
        android:id="@+id/signInButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:visibility="invisible"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Лістинг 20. Код xml файлу «colors»:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="payne_gray">#657380</color>
    <color name="dark_purple">#420039</color>
    <color name="magenta_haze">#932F6D</color>
    <color name="orchid">#E07BE0</color>
    <color name="teal200">#81D4FA</color>
    <color name="teal500">#3F51B5</color>
    <color name="periwinkle">#DCCCFF</color>

</resources>
```

Лістинг 21. Код xml файлу «strings»:

```
<resources>
    <string name="app_name">Final</string>
    <string name="hint_enter_city">Enter City</string>
    <string
name="google_maps_key">AIzaSyDW4dB2kuQZ_YPI0YCEE6cxrnXKSphpK2I</stri
ng>
</resources>
```

Лістинг 22. Код xml файлу «themes»:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <style name="Theme.Final" parent="Base.Theme.Final" />
</resources>
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)

Дипломна робота
на тему:

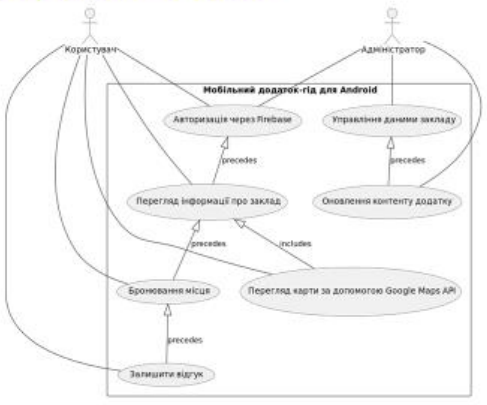
Розробка мобільного додатка-гіду для Android на основі Kotlin з використанням Firebase та Google Maps API

Виконав:
Скрипнік Богдан Іванович

студент 4 курсу, группи ІСД-42
спеціальності 126 Інформаційні системи та технології
Керівник: Миколайчук В.Р.

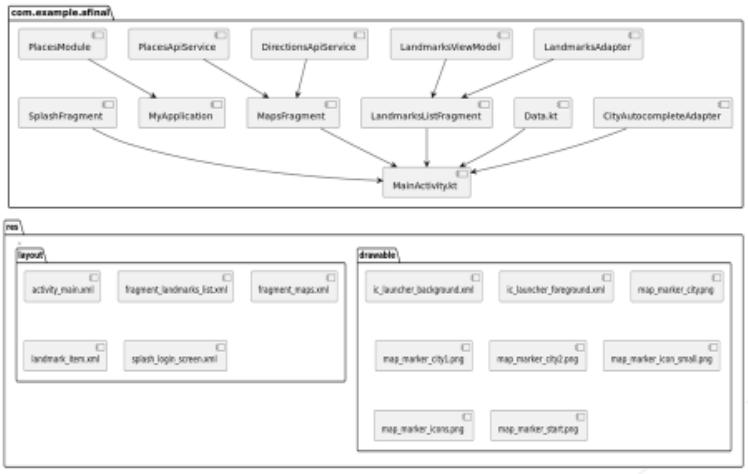
- ***Мета роботи*** - створення функціонального, зручного та безпечного мобільного додатку на основі Kotlin з використанням Firebase та Google Maps Api. Котрий зможе задовольнити потреби користувачів під час планування та здійснення поїздок і дати їм можливість максимально використовувати мобільні технології.
- **Методи дослідження** - системний аналіз, оптимізація, методики розробки мобільних додатків.
- ***Об'єкт дослідження*** -.мобільний додаток на основі Kotlin з використанням Firebase.
- ***Предмет дослідження*** - предметом дослідження є процес розробки та впровадження мобільного додатку гіду на базі Kotlin для Android з використанням Firebase та Google Maps API.

Діаграма Use-Case

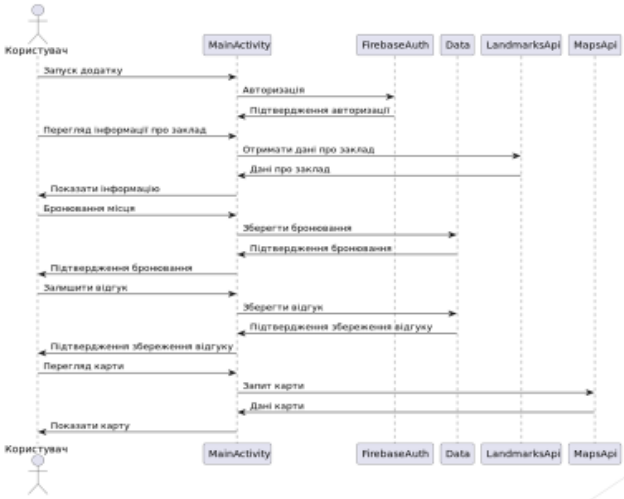


Контекстна діаграма

Діаграма компонентів



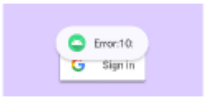
Діаграма послідовностей

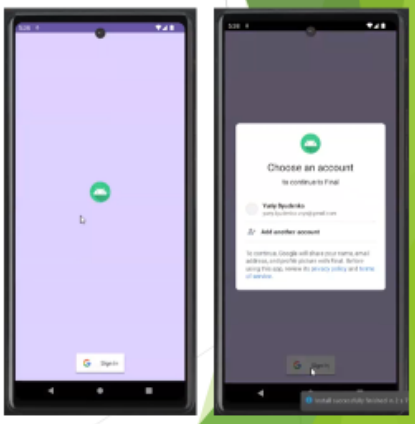


Діаграма розгортання

Візуальний вигляд мобільного додатку

- Екран авторизації з допомогою Google аккаунту, при невірних даних відображає помилку

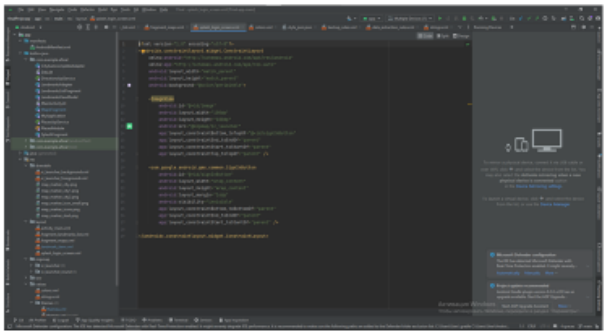




- Вибір міста та можливість обрати відоме місце поряд

- Прокладенні шляхи до місця

Інтерфейс середовища розробки Android Studio



Розробку мобільного додатку виконано у середовищі Android Studio з використанням мови програмування Kotlin

Висновки

- У першій частині були розглянуті архітектури мобільних додатків та їх вплив на оптимізацію і безпеку мобільного додатку.
- У другій частині була розглянута мова програмування Kotlin, а саме її вплив на мобільний додаток і інтеграцію з Firebase та Google Maps Api.
- У третьому розділі було здійснено розробку мобільного додатку і був проведений детальний аналіз взаємодії компонентів додатку.