ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Гібридний мобільний додаток медичного страхування на основі Kotlin Multiplatform»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

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

_____ Артем ЖУРАВЛЬОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Артем Журавльов
група КНДМ-62

Керівник: Сергій ІЩЕРЯКОВ
науковий ступінь, вчене звання к.т.н., доц.

Рецензент:
науковий ступінь, вчене звання _____
(Ім'я, ПРІЗВИЩЕ)

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність 122 Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Журавльов Артем Геннадійович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Гібридний мобільний додаток медичного страхування на основі Kotlin Multiplatform»

керівник кваліфікаційної роботи Сергій ІЩЕРЯКОВ к.т.н., доц.
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320

2. Строк подання кваліфікаційної роботи «15» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, технології кросплатформеної розробки на базі Kotlin Multiplatform, інструменти для створення інтерфейсу за допомогою Compose Multiplatform, методи інтеграції з бекенд-сервісами (Ktor, Firebase Cloud Messaging), а також підходи до оптимізації продуктивності та користувацького досвіду.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1. Дослідження підходів до розробки кросплатформених додатків на основі **Kotlin Multiplatform** та технологій для створення інтерфейсу користувача (CMP).
4.2. Проектування архітектури системи з використанням спільної бізнес-логіки та підбір інструментів для її реалізації.
4.3. Розробка прототипу мобільного додатку для медичного страхування з інтеграцією бекенд-сервісів та оптимізацією UI/UX.

5. Перелік графічного матеріалу: *презентація*
5.1. Основні методи проєктування інтерфейсу користувача з використанням **Compose Multiplatform**.
5.2. Архітектурні підходи та структура системи на основі **Kotlin Multiplatform**.
5.3. Приклади програмного коду: спільна логіка, інтеграція з бекендом (Ktor, Firebase Cloud Messaging).
5.4. Головні компоненти створеної системи та їх взаємодія.
6. Дата видачі завдання«16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.09-29.09.24	Виконано
2	Аналіз технологій кросплатформеної розробки	30.09-13.10.24	Виконано
3	Проектування архітектури додатку та підбір інструментів	14.10-27.10.24	Виконано
4	Розробка спільної бізнес-логіки додатку (KMP)	28.10-10.11.24	Виконано
5	Реалізація інтерфейсу користувача з використанням CMP	11.11-24.12.24	Виконано
6	Інтеграція з бекенд-сервісами та тестування	25.11-08.11.24	Виконано
7	Оформлення роботи: вступ, висновки, реферат	09.12-12.12.24	Виконано
8	Розробка демонстраційних матеріалів	09.12-15.12.24	Виконано

Здобувач вищої освіти _____ Артем ЖУРАВЛЬОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Сергій ІЩЕРЯКОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 77 стор., 3 табл., 9 рис., 15 джерел.

Наукове завдання – роботи полягає в комплексному дослідженні можливостей Kotlin Multiplatform для створення кросплатформених мобільних додатків на прикладі сфери медичного страхування.

Мета роботи – є оцінка ефективності Kotlin Multiplatform для розробки гібридних мобільних додатків та дослідження її впливу на процес розробки, продуктивність, стабільність і можливість повторного використання коду.

Об'єкт дослідження – є процес розробки гібридних мобільних додатків, що працюють на різних операційних системах, із використанням технології Kotlin Multiplatform.

Предмет дослідження – є ефективність застосування технології Kotlin Multiplatform у створенні мобільних додатків для медичного страхування, зокрема її вплив на швидкість розробки, продуктивність та стабільність роботи додатків на різних платформах.

Методи дослідження – аналіз та узагальнення підходів до кросплатформеної розробки; проектування архітектури додатку; інтеграція спільної бізнес-логіки на основі Kotlin Multiplatform; алгоритмізація обробки та збереження даних; тестування та оптимізація продуктивності додатку.

Короткий зміст роботи: Проведено аналіз сучасних технологій кросплатформеної розробки та їх можливостей для створення медичних додатків. Розглянуто переваги використання Kotlin Multiplatform у порівнянні з нативною розробкою та іншими кросплатформеними рішеннями (Flutter, React Native).

Розроблено архітектуру системи, що включає спільну бізнес-логіку та Compose Multiplatform для створення інтерфейсу користувача, адаптованого для платформ Android та iOS. Здійснено інтеграцію з бекенд-сервісами за допомогою Ktor та Firebase Cloud Messaging (FCM) для обробки даних і сповіщень.

Результатом роботи є прототип додатку для медичного страхування, що надає користувачам можливість:

Переглядати інформацію про медичні заклади.

Керувати даними про користувацький профіль.

Отримувати сповіщення про статус страхових полісів та оновлення.

Особливу увагу приділено оптимізації продуктивності, забезпеченню безпеки даних та покращенню користувацького досвіду.

Ключові слова: Kotlin Multiplatform, Compose Multiplatform, медичне страхування, кросплатформена розробка, інтеграція бекенд-сервісів, користувацький інтерфейс, оптимізація продуктивності.

ABSTRACT

Text part of the master's qualification work: 77 pages, 3 table, 9 pictures, 15 sources.

Scientific task is The scientific objective of the work lies in a comprehensive study of the capabilities of Kotlin Multiplatform for creating cross-platform mobile applications in the field of medical insurance.

Goal of the work is The purpose of this work is to evaluate the effectiveness of Kotlin Multiplatform for developing hybrid mobile applications and to analyze its impact on the development process, performance, stability, and code reusability.

Object of research – The object of the study is the process of developing hybrid mobile applications that operate on various operating systems using Kotlin Multiplatform technology

Subject of research – The subject of the study is the efficiency of applying Kotlin Multiplatform technology for creating mobile applications for medical insurance, particularly its impact on development speed, application performance, and stability across different platforms.

Research methods – The research methods include the analysis and generalization of cross-platform development approaches, application architecture design, integration of shared business logic based on Kotlin Multiplatform, data processing and storage algorithmization, testing, and application performance optimization.

Summary of the work: The work provides an analysis of modern cross-platform development technologies and their capabilities for creating medical insurance applications. The advantages of Kotlin Multiplatform over native development and other cross-platform solutions (such as Flutter and React Native) are considered.

The system architecture was developed, incorporating shared business logic and Compose Multiplatform for building a user interface adapted for Android and iOS platforms. Integration with backend services was implemented using Ktor and Firebase Cloud Messaging (FCM) for data processing and notifications.

The result of the work is a prototype of a medical insurance application that provides users with the ability to:

View information about medical institutions.

Manage user profile data.

Receive notifications about the status of insurance policies and updates.

Special attention was paid to performance optimization, data security, and improving the user experience.

Keywords: Kotlin Multiplatform, Compose Multiplatform, medical insurance, cross-platform development, backend services integration, user interface, performance optimization.

ЗМІСТ

ВСТУП……...12

1 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ГІБРИДНИХ МОБІЛЬНИХ ДОДАТКІВ НА БАЗІ KOTLIN MULTIPLATFORM……15

1.1 Загальна характеристика гібридних мобільних додатків……15

1.2 Порівняльний аналіз технологій розробки мобільних додатків…….22

1.3 Огляд мови програмування Kotlin та платформи Kotlin Multiplatform……..29

1.4 Особливості розробки з використанням Kotlin Multiplatform…….....33

1.5 Висновок до розділу 1……38

2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ РОЗРОБКИ ДОДАТКУ ДЛЯ МЕДИЧНОГО СТРАХУВАННЯ……..40

2.1 Аналіз предметної області медичного страхування……40

2.2 Визначення вимог до функціональності додатку……43

2.3 Опис використаних технологій та інструментів:……..47

2.3.1 Kotlin Multiplatform……..……...47

2.3.2 Compose Multiplatform (CMP)……..……..48

2.3.3 Android Studio та Xcode……..49

2.3.4 Інтеграція з бекенд-сервісами……....52

2.3.5 Бази даних та засоби збереження даних……..……..……..53

2.4 Висновок до розділу 2……..55

3 ПРОЦЕС РОЗРОБКИ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ДОДАТКУ……57

3.1 Збір вимог та проектування архітектури додатку……..…57

3.2 Розробка функціональності додатку……..59

3.2.1 Функція автентифікації користувача……....62

3.2.2 Функція перегляду страхових полісів……...65

3.2.3 Функція сповіщення та нагадування користувача……...68

3.2.4 Функція управління особистим кабінетом користувача…….68

3.2.5 Функція відображення карти клінік, які покриває страхова компанія……..70

3.3 Тестування та налагодження додатку……72

3.4 Аналіз ефективності використання Kotlin Multiplatform……..74
3.4.1 Оцінка продуктивності додатку……..…77
3.4.2 Порівняння з нативною розробкою та іншими кросплатформеними рішеннями…….79
3.5 Оптимізація додатку та покращення користувацького досвіду……....84
3.6 Висновок до розділу 3……...85
ВИСНОВКИ……87
ПЕРЕЛІК ПОСИЛАНЬ ……...89
ДОДАТКИ…….……..91
А. Програмний код……..91
Б. Скріншоти інтерфейсу додатку……....100
В. Демонстраційні матеріали (Презентація) ……...109

ВСТУП

На сьогодні мобільні додатки стали невід'ємною частиною нашого повсякденного життя. Вони активно використовуються в різних галузях, таких як банківська справа, роздрібна торгівля, освіта та охорона здоров'я. Особливого значення мобільні додатки набули в сфері медичного страхування, де швидкий доступ до інформації, оформлення послуг і контроль за своїм полісом є критичними для користувачів. Розробка таких додатків вимагає особливого підходу, який би поєднував у собі зручність використання, безпеку даних і продуктивність на різних платформах.

З появою багатоплатформних рішень, таких як Kotlin Multiplatform, з'явилися нові можливості для створення мобільних додатків, що можуть функціонувати на кількох платформах одночасно (Android та iOS). Це дозволяє зменшити витрати на розробку, скоротити час на підтримку і розвиток програмного продукту, оскільки значна частина коду є спільною для різних операційних систем. Однак, як і будь-яка технологія, Kotlin Multiplatform має свої переваги та обмеження, які вимагають глибшого вивчення та аналізу.

Актуальність дослідження полягає в тому, що сьогодні, зростання кількості мобільних пристроїв і різних операційних систем вимагає від розробників нових підходів до кросплатформеного програмування. Застосування Kotlin Multiplatform обіцяє ефективне вирішення цієї проблеми, проте важливо оцінити, наскільки дана технологія є вигідною в порівнянні з іншими методами розробки. Зокрема, важливо дослідити продуктивність додатків, створених за допомогою Kotlin Multiplatform, їхню стабільність та інші ключові аспекти, що впливають на вибір технологій для розробки мобільних додатків у сфері медичного страхування.

Об'єктом даного дослідження є процес розробки гібридних мобільних додатків, що працюють на різних операційних системах, із використанням технології Kotlin Multiplatform.

Предметом дослідження є ефективність застосування технології Kotlin Multiplatform у створенні мобільних додатків для медичного страхування, зокрема її вплив на швидкість розробки, продуктивність та стабільність роботи додатків на різних платформах.

Метою цього дослідження є оцінка ефективності Kotlin Multiplatform для розробки гібридних мобільних додатків та дослідження її впливу на процес розробки, продуктивність, стабільність і можливість повторного використання коду.

Головне завдання цієї магістерської роботи можна визначити так: розробити та оцінити ефективність мобільного додатку для медичного страхування на базі технології Kotlin Multiplatform, зокрема з точки зору продуктивності, зручності використання, стабільності та можливості повторного використання коду на різних платформах (Android та iOS).

Актуальність даного дослідження зумовлена швидким зростанням кількості мобільних додатків у різних галузях і необхідністю оптимізації процесу їх розробки. Особливо це стосується медичного страхування, де користувачі потребують швидкого доступу до своїх полісів та можливості оперативного управління послугами. Kotlin Multiplatform пропонує унікальне рішення для розробки додатків одночасно на двох основних мобільних платформах, що дає змогу компаніям скоротити час розробки та підтримки своїх додатків, а також знизити витрати. Проте, для впевненого впровадження цієї технології необхідно глибше дослідити її переваги та недоліки, що й робить це дослідження актуальним.

Наукова новизна роботи полягає в комплексному дослідженні можливостей Kotlin Multiplatform для створення кросплатформених мобільних додатків на прикладі сфери медичного страхування. У дослідженні буде представлено аналіз ефективності технології, порівняння з іншими підходами до розробки.

Практична значимість дослідження полягає в тому, що результати роботи можуть бути використані для покращення процесу розробки мобільних додатків, особливо в галузях, де критичними є безпека, продуктивність та зручність користування. Отримані результати можуть бути застосовані для оптимізації

мобільних додатків в інших сферах, а також для розробки нових рішень у медичному страхуванні та інших галузях.

1 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ГІБРИДНИХ МОБІЛЬНИХ ДОДАТКІВ НА БАЗІ KOTLIN MULTIPLATFORM

1.1 Загальна характеристика гібридних мобільних додатків

Розробка мобільних додатків за останні роки зазнала значних змін. У сучасній розробці мобільних додатків дедалі більше уваги приділяється кросплатформеній розробці, яка дозволяє створювати додатки, що працюють на різних операційних системах (Android, iOS та інших), використовуючи єдину кодову базу. Такий підхід дозволяє значно зменшити витрати часу та ресурсів на розробку, тестування та підтримку додатків. Адже раніше більшість додатків створювалися під кожну платформу окремо, використовуючи такі інструменти, як Android Studio та Xcode для iOS. Однак цей підхід вимагав значних ресурсів, оскільки кожен додаток необхідно було підтримувати та оновлювати окремо для кожної платформи. Це спричиняло високі витрати на розробку та підтримку.

З появою нових технологій, таких як Flutter, React Native, та Kotlin Multiplatform, розробники отримали можливість створювати кросплатформені рішення, які дозволяють писати більшу частину коду один раз та використовувати його на декількох платформах. Це значно скорочує час на розробку, тестування та підтримку додатків, оскільки одна і та ж кодова база може працювати як на Android, так і на iOS. Особливо важливо це для компаній, які прагнуть швидко вивести свої продукти на ринок, мінімізуючи при цьому витрати.

Одним з найперспективніших інструментів для цього є технологія Kotlin Multiplatform (KMP), яка дозволяє розробникам створювати багатоплатформенні рішення, зберігаючи при цьому високий рівень продуктивності та надійності.

Kotlin Multiplatform – це інструмент, який надає можливість розробляти спільний код для кількох платформ і використовувати його в Android, iOS, Web та інших середовищах. Це робить його надзвичайно привабливим для компаній та розробників, оскільки зменшується необхідність дублювати код для кожної

окремої платформи, що значно спрощує процес розробки. Водночас Kotlin Multiplatform дозволяє зберігати нативний досвід користувача на кожній платформі, оскільки розробники можуть інтегрувати платформо-специфічні компоненти у код на Kotlin [1].

Розробка мобільних додатків за останні роки зазнала значних змін. У сучасній розробці мобільних додатків дедалі більше уваги приділяється кросплатформеній розробці, яка дозволяє створювати додатки, що працюють на різних операційних системах (Android, iOS та інших), використовуючи єдину кодову базу. Такий підхід дозволяє значно зменшити витрати часу та ресурсів на розробку, тестування та підтримку додатків.

Раніше більшість додатків створювалися під кожну платформу окремо, використовуючи такі інструменти, як Android Studio та Xcode для iOS. Однак цей підхід вимагав значних ресурсів, оскільки кожен додаток необхідно було підтримувати та оновлювати окремо для кожної платформи. Це спричиняло високі витрати на розробку та підтримку.

Крім того, завдяки Kotlin Multiplatform значна частина інтерфейсу користувача (UI) також може мати спільний код. Наприклад, використовуючи Compose Multiplatform (CMP), розробники отримують можливість створювати UI-компоненти, які працюють одразу на різних платформах, забезпечуючи узгодженість дизайну. Це додатково скорочує витрати на розробку та підтримку інтерфейсу, дозволяючи при цьому інтегрувати платформо-специфічні компоненти для надання нативного досвіду користувача [2]

Основна перевага використання Kotlin Multiplatform полягає у здатності розділити код на дві частини: спільну частину, яка використовується на всіх платформах, та специфічну для кожної платформи, що дозволяє розробникам виконувати оптимізацію під конкретні вимоги платформи, зберігаючи при цьому багаторазовий код для інших задач, таких як бізнес-логіка, взаємодія з базами даних та мережеві операції.

Важливо зазначити, що хоча Kotlin Multiplatform є порівняно новою технологією, вона вже показує себе як перспективне рішення для побудови

сучасних мобільних додатків. Багато відомих компаній вже впровадили KMP у свої проекти для зниження витрат і підвищення ефективності [3]. Це свідчить про зростаючу роль цього інструменту у розробці кросплатформених рішень.

Окрім економії ресурсів, важливими перевагами KMP є можливість створення додатків зі спільною логікою, підтримка зручних засобів тестування, можливість інтеграції з нативними SDK кожної платформи, а також гнучкість у підходах до архітектури додатків.

Водночас, впровадження Kotlin Multiplatform вимагає глибокого розуміння архітектурних підходів до розробки додатків, оскільки розробники повинні вміти ефективно відокремлювати спільну логіку від специфічних функцій кожної платформи. Не менш важливим є також правильне використання платформо-специфічного коду для інтеграції з нативними функціями системи.

Розробка мобільних додатків є одним з найбільш затребуваних напрямків у сучасній індустрії програмного забезпечення. Це пов’язано з тим, що мобільні пристрої стали невід'ємною частиною життя людей, і компанії прагнуть охопити якомога більше користувачів через свої мобільні додатки. Сучасний ринок вимагає від розробників швидких, якісних рішень, які можуть працювати на кількох платформах одночасно, зберігаючи при цьому високий рівень продуктивності та зручності використання [1].

Традиційно, розробка мобільних додатків велася на нативних платформах, таких як Android або iOS, кожна з яких вимагала окремого підходу та інструментарію. Це означало, що для кожної операційної системи розроблялися окремі додатки, що призводило до збільшення витрат часу та ресурсів на розробку і підтримку. З огляду на це, виникли потреби в пошуку рішень, які б дозволяли спростити процес розробки та зменшити витрати, одночасно підтримуючи кілька платформ.

Гібридні мобільні додатки стали відповіддю на ці виклики. Ці додатки розробляються на основі кросплатформених технологій, які дозволяють створювати єдину кодову базу для кількох операційних систем. Одним із таких

рішень, що отримало широке поширення, є Kotlin Multiplatform (KMP), розроблене компанією JetBrains.

Kotlin Multiplatform дозволяє розробляти спільний код для бізнес-логіки додатків, а також забезпечує інтеграцію платформо-специфічних компонентів. Однак із розвитком технологій з'явилася можливість розробки спільного інтерфейсу користувача (UI), що додатково спрощує процес створення гібридних додатків. Одним із найбільш популярних інструментів для цієї мети є Compose Multiplatform (CMP).

Compose Multiplatform — це бібліотека, яка дозволяє створювати UI-компоненти, що працюють одразу на різних платформах, таких як Android, iOS, Desktop і Web. Використовуючи CMP, розробники можуть писати єдиний код для користувацького інтерфейсу, забезпечуючи узгодженість дизайну та поведінки на різних платформах. Такий підхід зменшує час та витрати на розробку, дозволяючи сконцентруватися на основній функціональності додатка [2].

Водночас CMP дозволяє зберігати нативний досвід користувача, оскільки розробники можуть налаштовувати окремі елементи під специфічні вимоги кожної платформи. Це робить технологію Kotlin Multiplatform разом із CMP надзвичайно ефективним інструментом для розробки сучасних гібридних додатків, які відповідають потребам бізнесу та користувачів [5].

Kotlin Multiplatform є інноваційним інструментом, який дозволяє розробляти багатоплатформені мобільні додатки, використовуючи єдину мову програмування - Kotlin. Відмінною особливістю цієї технології є її гнучкість і можливість спільного використання бізнес-логіки між платформами, зберігаючи при цьому доступ до нативних можливостей кожної з них.

Ідея полягає в тому, що більшість функцій додатку, таких як робота з мережею, бізнес-логіка або взаємодія з базами даних, може бути написана один раз і використовуватися на різних платформах (Android, iOS, Web тощо). Водночас, специфічний для кожної платформи код, який взаємодіє з нативними API, пишеться окремо. Це дозволяє зберігати гнучкість та оптимізувати додаток під кожну платформу, не жертвуючи продуктивністю.

Основні переваги використання Kotlin Multiplatform:

Одна з ключових переваг Kotlin Multiplatform полягає в тому, що ця технологія підтримує "дійсно спільний код" (спільна бізнес- логіка у KMP охоплює взаємодію з мережевими запитами, обробку баз даних і загальні розрахунки, що є критичними для багатоплатформних додатків. Це спрощує обслуговування коду і дозволяє відстежувати помилки в одному місці, оскільки оновлення або виправлення у спільній логіці автоматично застосовуються до всіх платформ)[6].

Kotlin Multiplatform також надає гнучкість у підході до вибору архітектури додатку. Наприклад, розробники можуть використовувати KMP для написання лише окремих частин програми, таких як бізнес-логіка або робота з даними, при цьому залишаючи нативну розробку інтерфейсу для кожної платформи. Це дозволяє розробникам адаптувати додаток під потреби користувачів кожної платформи, не жертвуючи кросплатформеною сумісністю.

Ще однією ключовою перевагою Kotlin Multiplatform є можливість створення уніфікованого інтерфейсу користувача (UI) за допомогою Compose Multiplatform (CMP). На відміну від традиційних підходів, CMP пропонує інструментарій, який спрощує процес розробки UI-компонентів завдяки єдиному коду, що працює на різних платформах від Android і IOS до Desktop і Web [2].

CMP не тільки скорочує час на створення інтерфейсу, але й забезпечує адаптивність дизайну до особливостей кожної платформи. Завдяки цьому розробники можуть інтегрувати специфічні функції, такі як використання жестів на iOS або підтримка специфічних сервісів Android, без необхідності дублювати основний код. Це дозволяє забезпечити баланс між кросплатформовістю і збереженням нативного досвіду користувача.

Крім того, CMP активно підтримує компонентний підхід до розробки. Це дає змогу створювати модульні та масштабовані UI-рішення, які легко адаптувати для майбутніх оновлень. Наприклад, дизайн авторизації або екрана профілю може бути реалізований як окремий модуль, який інтегрується в додаток на всіх платформах із мінімальними змінами.

Також важливо зазначити, що CMP дозволяє уникнути типових викликів, пов'язаних із різною поведінкою UI на різних пристроях. Завдяки власному механізму рендерингу, CMP забезпечує високу стабільність і відповідність дизайну вимогам платформ, незалежно від розміру екрана чи специфікацій пристрою.

Таким чином, інтеграція CMP із Kotlin Multiplatform відкриває нові можливості для створення продуктивних, масштабованих і зручних у підтримці багатоплатформних рішень. Це робить цю технологію універсальним вибором як для стартапів, так і для великих компаній, які прагнуть оптимізувати витрати та час на розробку.

Ще однією значною перевагою Kotlin Multiplatform є можливість поступової інтеграції з існуючими нативними проєктами. Це означає, що компанії можуть почати використовувати KMP для конкретних частин додатка, таких як бізнес-логіка або окремі модулі, без необхідності повного переписування програми. Подібний підхід дозволяє розробникам експериментувати з кросплатформеною розробкою, поступово оцінюючи її ефективність, і мінімізувати ризики, пов’язані з впровадженням нових технологій [5].

Однак перехід на Kotlin Multiplatform вимагає ретельного планування архітектури додатка. Хоча KMP дозволяє створювати спільну кодову базу, розробникам важливо структурувати проєкт так, щоб забезпечити максимальну повторюваність коду та уникнути дублювання логіки. Це особливо стосується складних систем, де залежності від платформного коду можуть ускладнювати підтримку і розширення функціональності. Наприклад, інтеграція платформо-специфічних компонентів, таких як UI або робота з апаратним забезпеченням, може вимагати додаткового часу для налаштування.

Хоча Kotlin Multiplatform має багато переваг, існують певні проблеми та обмеження, які необхідно враховувати. Однією з основних проблем є необхідність глибокого розуміння архітектурних підходів до розробки кросплатформених додатків. Оскільки KMP дозволяє створювати спільну кодову базу, важливо правильно структурувати додаток, щоб забезпечити максимальну повторюваність коду та зменшити залежності від платформи.

Також KMP вимагає підтримки окремих частин коду для кожної платформи, коли йдеться про специфічні для платформи функції (наприклад, робота з камерою або інтерфейсом користувача).

Наприклад, якщо розробники обирають підхід із використанням нативного коду для інтерфейсу користувача, то їм доведеться створювати окремі UI-компоненти для кожної платформи, що може збільшити витрати часу на розробку та підтримку.

Це може призводити до додаткових витрат часу та зусиль на підтримку таких компонентів, особливо якщо додаток має складну структуру. У таких випадках використання спільного коду для інтерфейсу за допомогою Compose Multiplatform може суттєво спростити розробку, але все одно потребує належного планування архітектури. Наприклад, розробники повинні враховувати особливості адаптивності інтерфейсу для різних розмірів екранів і платформ. CMP пропонує гнучкі механізми для створення кастомізованих дизайнів, але це також може вимагати певних зусиль для коректної інтеграції платформо-специфічних елементів.

Крім того, технологія Kotlin Multiplatform все ще перебуває у стадії активної розробки та вдосконалення. Хоча вона вже широко використовується і підтримується спільнотою розробників, можуть виникати труднощі, пов'язані з обмеженою документацією або відсутністю підтримки деяких специфічних функцій. Це стосується і CMP, оскільки, попри її значний потенціал, деякі функціональні можливості можуть ще не мати стабільної реалізації, особливо для менш поширених платформ, таких як Web чи Desktop.

Для розробників, які тільки починають працювати з Kotlin Multiplatform і Compose Multiplatform, важливим фактором успіху є активне вивчення доступної документації, спільноти та прикладів застосування. Водночас, варто враховувати, що використання CMP дозволяє уникнути багатьох проблем, які зазвичай виникають при окремій розробці інтерфейсу для кожної платформи. Це робить технологію CMP не лише засобом для прискорення розробки, але й інструментом,

що сприяє ефективному вирішенню архітектурних викликів у багатоплатформних проєктах [2].

1.2 Порівняльний аналіз технологій розробки мобільних додатків

У сучасній розробці мобільних додатків існує кілька підходів до створення програмного забезпечення, які відрізняються за своїми технологічними рішеннями та підходами до реалізації. Основними категоріями є нативні, кросплатформені та гібридні технології розробки. Кожен з цих підходів має свої переваги, недоліки та сфери застосування, які впливають на вибір технології для певного типу проєктів.

Нативна розробка орієнтована на конкретну платформу, що дозволяє використовувати всі оптимізації та можливості апаратного забезпечення, доступні в операційній системі. Цей підхід забезпечує найвищу продуктивність і стабільність, а також повний доступ до інструментів для створення складного інтерфейсу. Наприклад, для Android це Kotlin або Java з Android SDK, а для iOS – Swift з Xcode."[6].

Переваги нативної розробки:

1. Максимальна продуктивність. Оскільки нативні додатки безпосередньо працюють з апаратними можливостями пристрою, вони мають вищу продуктивність порівняно з іншими підходами.

2. Повний доступ до функцій ОС. Нативні додатки можуть використовувати всі можливості платформи, включаючи доступ до API, сенсорів, камери, GPS та інших апаратних компонентів.

3. Нативний інтерфейс користувача (UI). Можливість створення інтерфейсу, який повністю відповідає гайдлайнам кожної платформи (Android Material Design, Apple Human Interface Guidelines), що дозволяє забезпечити високий рівень користувацького досвіду (UX).

Недоліки нативної розробки:

1. Високі витрати на розробку та підтримку. Для кожної платформи потрібна окрема команда або спеціалісти, оскільки код, написаний для Android, не може бути використаний для iOS і навпаки. Це значно збільшує вартість розробки.

2. Триваліший час на розробку. Оскільки додатки розробляються окремо для кожної платформи, цей процес потребує більше часу на проєктування, програмування та тестування.

3. Важка підтримка. Всі оновлення та нові функції потрібно реалізовувати у двох окремих кодових базах, що ускладнює підтримку та розвиток продукту.

Кросплатформені технології розробки дозволяють створювати додатки, які працюють на декількох платформах (наприклад, Android та iOS), використовуючи єдину кодову базу. Серед найбільш популярних технологій кросплатформеної розробки можна виділити React Native, Flutter та Xamarin.

1. React Native. Розроблений компанією Facebook. React Native дозволяє створювати мобільні додатки, використовуючи JavaScript. Більшість коду може бути спільною для обох платформ, проте при необхідності можна використовувати нативний код для специфічних функцій.

- Переваги: швидкий процес розробки завдяки великій кількості бібліотек і модулів, підтримка нативних компонентів, активна спільнота.

- Недоліки: обмежена продуктивність для складних або ресурсоємних додатків, залежність від бібліотек третьої сторони, труднощі з інтеграцією нових API або специфічних функцій платформи. [табл.1.1]

2. Flutter. Це фреймворк від Google, який використовує мову програмування Dart. Flutter дозволяє створювати високопродуктивні додатки з єдиною кодовою базою, які виглядають однаково на різних платформах.

- Переваги: висока продуктивність додатків, завдяки використанню власного рендерингу UI, гарна підтримка від Google, єдина кодова база для Android та iOS.

- Недоліки: обмежена кількість нативних компонентів, необхідність вивчення мови Dart, великі розміри додатків. [табл.1.1]

3. Xamarin. Ця технологія, розроблена Microsoft, дозволяє створювати додатки на мові C#. Xamarin дає змогу розробляти спільну бізнес-логіку, використовуючи нативні інтерфейси для кожної платформи.

- Переваги: інтеграція з екосистемою Microsoft, спільний код для логіки, можливість створення нативного інтерфейсу.

- Недоліки: менша продуктивність порівняно з нативними додатками, обмежена підтримка нативних API [9].

Гібридні додатки є проміжним рішенням між веб-додатками і нативними. Вони використовують веб-технології (HTML, CSS, JavaScript) всередині нативного контейнера, що дозволяє запускати додаток на будь-якій платформі. Відомі фреймворки для гібридної розробки включають Cordova та Ionic. [табл.1.1]

Переваги гібридної розробки:

1. Швидкість розробки. Завдяки використанню веб-технологій можна швидко створити простий мобільний додаток.

2. Використання єдиної кодової бази. Код пишеться один раз і може бути використаний для всіх платформ.

3. Простота підтримки. Легке оновлення через єдину базу коду.

Недоліки гібридної розробки:

1. Низька продуктивність. Гібридні додатки зазвичай повільніші, ніж нативні або навіть кросплатформені рішення, оскільки вони використовують браузерні технології для відображення інтерфейсу.

2. Обмежений доступ до апаратних можливостей. Гібридні додатки не завжди можуть ефективно використовувати ресурси пристрою.

3. Гірший UX. Інтерфейс гібридних додатків може виглядати неприродно і не відповідати нативним стандартам платформ.

Таблиця 1.1 Порівняльна таблиця між Kotlin Multiplatform (KMP), Flutter та React Native

Критерій	**Kotlin Multiplatform (KMP) Compose Multiplatform (CMP)**	**Flutter**	**React Native**
Мова програмування	Kotlin	Dart	JavaScript
Спільний код	Бізнес-логіка (KMP) та UI (CMP)	Спільний код для UI та логіки	Спільний код з можливістю додавання нативного коду
Інтерфейс користувача (UI)	Спільний UI з можливістю кастомізації для кожної платформи	Власний рендеринг UI через Skia	JavaScript-компоненти, частково нативний UI
Продуктивність	Висока (завдяки нативному коду)	Висока, але рендеринг важчий	Помірна; нижча для складних додатків
Інтеграція з нативним кодом	Повна інтеграція з платформеними API	Обмежена інтеграція	Можливість інтеграції через мост (Bridge)
Використання у наявних проєктах	Легке впровадження у нативний код	Повний перепис коду додатку	Часткове впровадження можливе
Підтримка платформ	Android, iOS, Web, Desktop	Android, iOS, Web, Desktop	Android, iOS
Великі корпорації	Google, JetBrains	Google	Meta (Facebook)
Документація та підтримка	Середній рівень, активний розвиток	Високий рівень документації	Високий рівень підтримки
Розмір додатку	Менший розмір завдяки нативному підходу	Більший розмір додатків	Середній розмір додатків
Вивчення технології	Середня складність (знання Kotlin та CMP)	Висока складність (вивчення Dart)	Легка для розробників JavaScript

Дана таблиця була додана для наочності та кращого розуміння ключових відмінностей між цими технологіями у контексті кросплатформеної розробки.

Kotlin Multiplatform (KMP) є унікальною технологією, оскільки поєднує переваги як нативної, так і кросплатформеної розробки. На відміну від інших кросплатформених рішень, KMP дозволяє розробляти спільну кодову базу для бізнес-логіки, при цьому залишаючи можливість використовувати нативний інтерфейс для кожної платформи.

Переваги Kotlin Multiplatform:

1. Спільна кодова база. Можливість писати спільний код для різних платформ, що дозволяє суттєво зменшити час і витрати на розробку.

2. Нативний UX. Завдяки використанню нативного інтерфейсу на кожній платформі, додатки, розроблені з використанням KMP, виглядають і функціонують як нативні.

3. Гнучкість. Розробники можуть інтегрувати KMP у вже існуючі нативні проекти або створювати нові додатки з повною підтримкою кросплатформеної логіки.

Недоліки Kotlin Multiplatform:

1. Потрібно більше часу на вивчення. Хоча розробка з KMP забезпечує гнучкість, для новачків ця технологія може бути складною через необхідність розуміння як нативного, так і кросплатформеного коду.

2. Ранні стадії розвитку. KMP все ще знаходиться на стадії активної розробки, що може означати обмежену підтримку та документацію для деяких функцій.

Кожен з підходів до розробки мобільних додатків має свої сильні та слабкі сторони, що робить вибір технології критично важливим залежно від конкретних вимог проєкту, доступних ресурсів і цілей бізнесу [5].

Переваги використання CMP у KMP проектах:

Спільний код для UI: CMP дозволяє створювати спільний інтерфейс користувача, що значно скорочує час розробки та витрати. Завдяки цьому розробники можуть уникнути дублювання коду для кожної платформи, що є

особливо важливим у проєктах зі складною структурою або великим обсягом екранів.

Узгодженість дизайну: Використовуючи CMP, компанії можуть забезпечити єдиний стиль дизайну на всіх платформах, що покращує користувацький досвід (UX) та зменшує зусилля на забезпечення відповідності гайдлайнам.

Гнучкість адаптації: CMP дозволяє налаштовувати платформо-специфічні компоненти UI без необхідності повного дублювання інтерфейсу для кожної платформи. Це забезпечує збереження нативного вигляду і поведінки.

Недоліки використання CMP у KMP проектах:

Обмежена продуктивність на малопотужних пристроях: Оскільки CMP використовує спільний рендеринг для UI, його продуктивність може бути нижчою порівняно з повністю нативними інтерфейсами, особливо на застарілих або малопотужних пристроях.

Ранній етап розвитку: CMP, як і KMP, перебуває у стадії активної розробки. Це може спричинити обмежену підтримку або відсутність деяких функціональних можливостей, зокрема для менш поширених платформ, таких як Web або Desktop.

Додаткові зусилля на інтеграцію: У проєктах, де вже існують нативні інтерфейси, впровадження CMP може вимагати значних зусиль для адаптації та перенесення існуючого UI-коду.

Нативна розробка забезпечує найвищий рівень продуктивності та інтеграції з операційними системами, пропонуючи користувачам найбільш оптимізований досвід (UX). Вона залишається найкращим вибором для проєктів, де важливі висока продуктивність, доступ до апаратного забезпечення та можливість створення складних інтерфейсів. Однак, цей підхід має високі витрати на розробку та підтримку через необхідність дублювати код для кожної платформи [6].

Кросплатформені рішення дозволяють значно зменшити витрати на розробку, надаючи можливість використовувати єдину кодову базу для кількох платформ. Попри деякі компроміси у продуктивності порівняно з нативними рішеннями, технології, такі як React Native та Flutter, забезпечують достатньо високу продуктивність і гнучкість для багатьох застосувань [9].

Це робить їх чудовим вибором для стартапів або компаній, які хочуть швидко випустити продукт на ринок з мінімальними витратами.

Завдяки Kotlin Multiplatform (KMP) розробники можуть комбінувати спільну логіку для основних функцій із нативними компонентами для кожної платформи. Це дозволяє одночасно знижувати витрати на розробку і підвищувати продуктивність. KMP підтримує expect/actual механізм, що дозволяє інтегрувати специфічні функції для кожної ОС, забезпечуючи гнучкість у реалізації бізнес-логіки та доступ до API на рівні платформи."

Kotlin Multiplatform пропонує компроміс між нативною та кросплатформенною розробкою, дозволяючи використовувати спільну бізнес-логіку між платформами, зберігаючи нативний інтерфейс користувача.

Це робить KMP привабливим варіантом для проєктів, де необхідна кросплатформена розробка з мінімізацією технічного боргу, але збереженням максимальної якості для кожної платформи.

Завдяки використанню Compose Multiplatform (CMP) у рамках Kotlin Multiplatform, розробники отримують потужний інструмент для створення інтерфейсу користувача. CMP забезпечує значну економію часу на розробку UI, дозволяючи створювати спільний код для різних платформ. Це не тільки зменшує витрати, але й забезпечує узгодженість дизайну на всіх пристроях.

Compose Multiplatform (CMP) є перспективною технологією, однак її використання вимагає врахування певних технічних викликів. Одним із таких є залежність від інструментів та бібліотек, які поки що перебувають на стадії активного розвитку. Наприклад, деякі популярні компоненти для роботи з UI або анімацією ще не підтримують повної інтеграції з CMP, що може обмежувати можливості розробників у реалізації унікального дизайну [2].

Також варто враховувати, що CMP орієнтована на уніфікацію процесу створення інтерфейсів, однак це може ускладнити оптимізацію специфічних рішень для окремих платформ. У таких випадках розробникам доводиться створювати додаткові налаштування або доповнення, щоб забезпечити необхідну продуктивність і відповідність платформним стандартам.

Окрім цього, відсутність зрілої екосистеми може впливати на швидкість розробки, оскільки деякі проблеми або питання інтеграції доводиться вирішувати вручну. Це особливо актуально для команд, які лише починають використовувати Kotlin Multiplatform і CMP, адже потрібен час для освоєння нових підходів до роботи.

Попри ці виклики, CMP залишається одним із найперспективніших інструментів для створення багатоплатформних додатків. Його подальший розвиток сприятиме розширенню підтримки платформ і полегшенню інтеграції зі складними архітектурними рішеннями.

Загалом, CMP у поєднанні з KMP забезпечує відмінний баланс між швидкістю розробки та збереженням нативного досвіду користувача, що робить цю технологію перспективним вибором для багатоплатформних проєктів.

Використання Kotlin Multiplatform дозволяє зменшити витрати на розробку та пришвидшити випуск продукту на ринок, водночас забезпечуючи високу продуктивність і можливість гнучкої інтеграції з нативними функціями ОС.

Загалом, вибір технології залежить від конкретних вимог та особливостей проєкту. Якщо важлива продуктивність і нативний досвід, краще використовувати нативну розробку. Якщо ж головними критеріями є швидкість розробки і зниження витрат, кросплатформені або гібридні рішення можуть бути більш прийнятними. Kotlin Multiplatform надає унікальну можливість поєднати переваги обох підходів, дозволяючи створювати продуктивні додатки з мінімізацією витрат і часу на розробку.

1.3 Огляд мови програмування Kotlin та платформи Kotlin Multiplatform

Kotlin розроблений компанією JetBrains у 2011 році, є сучасною мовою програмування, що орієнтована на лаконічність, безпечність і ефективність, зокрема для Android-платформи. Її повна сумісність із Java дозволяє плавно інтегрувати Kotlin у існуючі Java-проєкти, забезпечуючи поступовий перехід без необхідності повного переписування коду. Це особливо важливо для великих

команд і проектів, що прагнуть підвищити продуктивність, зберігаючи при цьому наявні розробки.

Kotlin має низку суттєвих переваг порівняно з іншими мовами програмування, які зробили її популярною серед розробників мобільних додатків:

1. Лаконічність та зручність синтаксису. Kotlin має значно компактніший синтаксис порівняно з Java, що дозволяє писати менше коду для виконання тих самих завдань. Наприклад, у Kotlin доступні більш гнучкі варіанти оголошення змінних, функцій та класів, а також зручні механізми для роботи з колекціями та функціями вищого порядку.

2. Зменшення часу на розробку. Завдяки можливості використовувати спільну кодову базу, розробники можуть суттєво зменшити час, необхідний для створення додатків для кількох платформ. Це дозволяє компаніям швидше випускати нові продукти та оновлення.

3. Безпека типів. Kotlin вирішує одну з найбільш поширених проблем Java – можливість виникнення помилок, пов'язаних з використанням значення `null`. Завдяки механізму нулебезпеки (null safety), Kotlin не дозволяє розробникам використовувати `null` без явної вказівки цього у коді, що дозволяє уникнути типових помилок під час виконання програми (NullPointerException).

4. Покращення якості коду. Спільний код дозволяє зменшити кількість дублікатів, що позитивно впливає на якість програмного забезпечення. Також зменшується ймовірність виникнення помилок під час розробки, оскільки код тестується і використовується на кількох платформах.

5. Гнучкість у використанні нативного коду. Kotlin Multiplatform надає можливість писати нативний код для кожної платформи там, де це необхідно. Це забезпечує максимальну продуктивність та інтеграцію з нативними API операційних систем, зберігаючи при цьому переваги кросплатформеного підходу.

6. Легка інтеграція в існуючі проєкти. Kotlin Multiplatform дозволяє інтегрувати спільний код у вже існуючі нативні проєкти. Це дає змогу компаніям поступово переходити до кросплатформеної розробки без необхідності повного переписування додатків.

7. Функціональний підхід. Kotlin поєднує об'єктно-орієнтований та функціональний підходи до програмування, дозволяючи розробникам використовувати функції вищого порядку, лямбда-вирази та інші функціональні конструкції. Це робить код більш читабельним та зрозумілим.

8. Повна сумісність із Java. Одна з ключових переваг Kotlin полягає в тому, що він повністю сумісний з існуючим кодом на Java. Розробники можуть використовувати Kotlin у поєднанні з Java в одному проєкті, що дозволяє поступово інтегрувати Kotlin у вже існуючі системи.

9. Підтримка Android. У 2017 році компанія Google оголосила Kotlin офіційно підтримуваною мовою для розробки Android-додатків, що призвело до її масового впровадження в цій галузі. Kotlin інтегрується з Android SDK і забезпечує простіший і ефективніший спосіб розробки Android-додатків порівняно з Java.

10. Потужна інфраструктура. Завдяки підтримці JetBrains та активній спільноті, Kotlin має добре розвинену інфраструктуру, яка включає в себе потужний набір інструментів для розробки (IDE, компілятори, бібліотеки) та підтримку великих корпорацій, таких як Google.

Недоліки Kotlin Multiplatform:

1. Відносна складність навчання. Оскільки KMP включає елементи як нативної, так і кросплатформеної розробки, для нових розробників ця технологія може здатися складнішою в освоєнні порівняно з більш традиційними підходами. Вивчення правил і принципів побудови спільного і платформо-специфічного коду вимагає часу та практичного досвіду.

2. Обмежена підтримка деяких бібліотек. Хоча екосистема Kotlin Multiplatform активно розвивається, не всі популярні бібліотеки підтримують кросплатформені рішення на 100%. Це може вимагати написання специфічного коду для кожної платформи або пошуку альтернативних бібліотек, що може ускладнювати розробку.

3. Порівняно нова технологія. Kotlin Multiplatform все ще знаходиться в стадії активного розвитку. Це означає, що існують обмеження у функціональності, можливі зміни в майбутніх версіях і недостатньо зріла документація для деяких

аспектів платформи. Через це розробники можуть стикатися з технічними викликами або відсутністю стабільних інструментів для певних платформ.

Kotlin Multiplatform (KMP) – це розширення мови Kotlin, яке дозволяє використовувати одну і ту саму кодову базу для розробки додатків на кілька платформ, таких як Android, iOS, Web, Windows, Linux, та інші. Основна ідея KMP полягає в тому, щоб розділити код на дві частини: спільний код для кількох платформ і специфічний для кожної платформи код.

Kotlin Multiplatform дозволяє розробникам зменшити час і витрати, необхідні для створення багатоплатформних додатків, за рахунок єдиної кодової бази. Спільний код може включати бізнес-логіку, роботу з мережею та базами даних. Проблема окремого створення UI для кожної платформи частково вирішується за допомогою Compose Multiplatform (CMP). CMP інтегрується зі спільним кодом KMP, дозволяючи створювати уніфікований інтерфейс користувача для кількох платформ, зберігаючи можливість адаптації до специфіки кожної ОС.

Compose Multiplatform (CMP) є важливим компонентом екосистеми Kotlin Multiplatform. CMP дозволяє створювати спільний код для інтерфейсу користувача (UI), що значно спрощує розробку та забезпечує узгодженість дизайну між платформами. CMP інтегрується зі спільним кодом бізнес-логіки, що дозволяє зменшити дублювання зусиль і витрат на розробку інтерфейсу для Android, iOS та інших платформ [2].

Завдяки CMP, розробники можуть використовувати уніфікований підхід до створення UI, одночасно адаптуючи дизайн під особливості кожної платформи. Наприклад, використання нативних компонентів та API для окремих платформ дозволяє зберегти нативний досвід користувача, забезпечуючи баланс між уніфікованим дизайном і специфікою платформи.

Це спрощує процес оновлення і підтримки додатків, оскільки внесені зміни в спільний код застосовуються до всіх платформ.

Мова програмування Kotlin і платформа Kotlin Multiplatform (KMP) є інноваційними інструментами для розробки мобільних додатків, які поєднують у собі простоту, гнучкість і ефективність. Kotlin завоював популярність завдяки своїй

лаконічності, безпеці типів та повній сумісності з Java, що робить його ідеальним вибором для Android-розробки. Kotlin Multiplatform відкриває нові можливості для кросплатформеної розробки, дозволяючи розробникам використовувати спільний код для кількох платформ, зберігаючи при цьому можливість гнучкої інтеграції з нативними функціями кожної операційної системи.

1.4 Особливості розробки з використанням Kotlin Multiplatform

Розробка з використанням Kotlin Multiplatform (KMP) відкриває нові можливості для створення кросплатформених мобільних додатків з ефективною та гнучкою кодовою базою. Це підхід, який забезпечує переваги обох світів: нативної розробки, що надає високий рівень інтеграції з платформою, та кросплатформеної, яка дозволяє повторно використовувати більшість коду для різних операційних систем.

Розробка на базі KMP відрізняється кількома важливими аспектами, які роблять її ефективною та привабливою для розробників, зокрема у випадках, коли проєкт потребує підтримки кількох платформ одночасно.

Kotlin Multiplatform дозволяє розробникам розділяти код на дві основні частини:

1. Спільний (common) код. Спільний код – це та частина програми, яка може використовуватися на кількох платформах без змін. Зазвичай це бізнес-логіка, робота з мережею, взаємодія з базами даних та інші загальні для всіх платформ функції. Цей код пишеться один раз і компілюється для кожної підтримуваної платформи.

Спільний код у KMP дозволяє абстрагувати більшість загальних для всіх платформ операцій, таких як:

- Обробка HTTP-запитів та відповіді з серверів.
- Робота з базами даних через спільні інтерфейси.
- Логіка бізнес-процесів.
- Математичні обчислення та логічні операції.

2. Платформо-залежний код, що містить специфічніна функції для кожної платформи, наприклад, для взаємодії з нативним інтерфейсом користувача або доступу до апаратних можливостей пристрою (камери, сенсорів тощо).

Ця модульна архітектура дозволяє розробникам використовувати загальну частину програми для більшості логічних операцій, що суттєво знижує обсяги коду та спрощує підтримку. Платформо-залежні частини, навпаки, зберігаються для специфічної оптимізації під конкретні платформи, де це необхідно [6].

3. Спільні бібліотеки. Kotlin Multiplatform підтримує використання спільних бібліотек для декількох платформ. Наприклад, популярні бібліотеки для роботи з мережею або базами даних (наприклад, Ktor, SQLDelight) можуть використовуватися як у Android, так і в iOS-додатках без необхідності написання окремого коду для кожної платформи [14].

4. Нативні модулі. У випадках, коли певна функціональність специфічна для однієї платформи, можна створювати нативні модулі для кожної операційної системи. Це дозволяє зберегти максимальну гнучкість та адаптувати додаток під конкретні потреби кожної платформи.

Для реалізації спільного коду розробники використовують стандартні бібліотеки та інтерфейси, які підтримують мультиплатформеність. Популярні бібліотеки, такі як Ktor (для роботи з мережею), SQLDelight (для роботи з базами даних), Kotlinx Serialization (для серіалізації даних), дозволяють написати спільний код один раз і використовувати його як на Android, так і на iOS чи інших платформах [4].

Хоча KMP дозволяє значно зменшити кількість кодових дублікатів завдяки спільному коду, важливою складовою є також можливість інтеграції з платформо-залежними API. Для цього кожна платформа може мати власну реалізацію для специфічних компонентів, таких як:

- Інтерфейс користувача (UI).
- Апаратні можливості пристроїв (GPS, камера, біометрія).
- Локальні ресурси пристрою (системні повідомлення, оповіщення, сенсори).

У KMP цей підхід реалізовано за допомогою expect/actual механізму. Розробники можуть оголосити спільний інтерфейс у common-коді, використовуючи ключове слово `expect`, і визначити конкретну реалізацію для кожної платформи за допомогою ключового слова `actual`. Це дозволяє зберегти гнучкість і забезпечити відповідну нативну інтеграцію для кожної платформи.

Compose Multiplatform (CMP) виступає важливим доповненням до екосистеми Kotlin Multiplatform, яке значно полегшує розробку інтерфейсу користувача (UI). Завдяки CMP розробники отримують можливість створювати спільний код для UI, який автоматично адаптується до особливостей різних платформ, таких як Android, iOS, Web та Desktop. Це усуває необхідність дублювати код для кожної платформи та забезпечує узгодженість дизайну.

На відміну від традиційного підходу, де для кожної операційної системи доводилося створювати окремі UI-компоненти, CMP дозволяє реалізувати універсальний інтерфейс із мінімальними зусиллями. Наприклад, елементи, такі як списки, форми введення чи кнопки, можуть бути створені один раз і використовуватися на всіх платформах, зберігаючи єдиний стиль і функціональність.

Водночас CMP підтримує налаштування специфічних компонентів для окремих платформ, що дозволяє розробникам адаптувати ключові UI-елементи під унікальні вимоги кожної операційної системи. Це особливо актуально для додатків із високими вимогами до дизайну чи специфічної поведінки, наприклад, інтерактивних жестів на iOS або роботи з повідомленнями Android.

Таким чином, CMP інтегрує переваги кросплатформеного підходу зі збереженням гнучкості, дозволяючи розробникам балансувати між уніфікацією інтерфейсу та специфікою кожної платформи.

Тестування є важливою складовою будь-якого процесу розробки, і Kotlin Multiplatform також забезпечує ефективний механізм для цього. Розробники можуть писати тести для спільної логіки (common-коду), які працюватимуть для всіх підтримуваних платформ. Це дає можливість автоматизувати перевірку бізнес-логіки додатку, незалежно від платформи, для якої вона використовується.

Також можна реалізовувати платформо-залежні тести, що дозволяють перевіряти специфічні функції для кожної платформи. Для цього використовуються стандартні інструменти тестування для Android та iOS, такі як JUnit для Android або XCTest для iOS.

Однією з ключових особливостей Kotlin Multiplatform є можливість інтеграції з існуючими нативними проєктами. Це особливо актуально для компаній, які мають уже розроблені мобільні додатки для Android або iOS, але прагнуть скоротити витрати на розробку нових функцій.

KMP дозволяє інтегрувати спільний код без необхідності переписувати весь додаток. Наприклад, можна поступово мігрувати бізнес-логіку до спільного коду та залишити платформо-залежні частини без змін. Це забезпечує плавний перехід до кросплатформеної розробки без серйозних ризиків для існуючого функціоналу.

Переваги використання Kotlin Multiplatform у розробці:

1. Оптимізація витрат на розробку. Однією з головних переваг є можливість значно скоротити витрати часу і ресурсів на розробку додатків для кількох платформ. Це досягається завдяки повторному використанню спільної кодової бази.

2. Швидше випуск оновлень. Використання спільної кодової бази дозволяє одночасно впроваджувати зміни на декількох платформах. Це скорочує час на підтримку та тестування окремих версій додатка.

3. Модульність та гнучкість. Kotlin Multiplatform дозволяє будувати архітектуру додатка модульним способом, розділяючи код на незалежні частини, що полегшує його розвиток та тестування.

4. Нативний досвід користувача. Завдяки використанню нативних компонентів інтерфейсу користувача для кожної платформи, додатки, розроблені з KMP, забезпечують високий рівень користувацького досвіду, який відповідає очікуванням користувачів на кожній платформі.

5. Покращення якості коду. Оскільки спільний код використовується на кількох платформах, це дозволяє розробникам зосередитися на створенні якіснішого та безпечнішого коду, який проходить перевірку в різних умовах [6].

Compose Multiplatform (CMP) є важливим компонентом екосистеми KMP, який суттєво спрощує розробку інтерфейсу користувача. Використовуючи CMP, розробники можуть створювати спільний код для UI, що автоматично адаптується до різних платформ, таких як Android, iOS, Web та Desktop. Цей підхід дозволяє реалізувати узгодженість дизайну та поведінки додатків, мінімізуючи витрати на створення інтерфейсу для кожної платформи.

CMP особливо ефективний для додатків, які потребують однакових або схожих елементів UI на різних платформах, наприклад, форм авторизації, списків чи інформаційних екранів. Він також підтримує платформо-специфічні налаштування, що дозволяє додавати унікальні елементи для окремих платформ, наприклад, підтримку жестів на iOS або інтеграцію з системними сервісами Android.

Недоліки та виклики:

1. Відносна новизна технології. Kotlin Multiplatform все ще розвивається і не всі функції є достатньо зрілими. Це може створювати труднощі з підтримкою певних специфічних функцій або бібліотек.

2. Обмежена підтримка бібліотек. Хоча кількість бібліотек для KMP постійно зростає, деякі специфічні інструменти або рішення можуть не бути доступними для мультиплатформеного використання, що вимагає додаткових зусиль для реалізації власних рішень.

3. Необхідність підтримки нативного коду. Для платформо-залежних частин все одно потрібно писати специфічний код, що може ускладнити розробку у випадках, коли потрібна глибока інтеграція з платформою.

Kotlin Multiplatform надає потужний інструмент для розробки кросплатформених додатків, дозволяючи повторно використовувати більшість бізнес-логіки та суттєво зменшувати витрати на підтримку декількох платформ. Основною особливістю KMP є його гнучкість, яка дозволяє зберігати нативний досвід користувача для кожної платформи, водночас оптимізуючи процес розробки завдяки спільній кодовій базі. Попри певні виклики, пов'язані з відносною новизною технології та необхідністю підтримки платформо-залежного коду, KMP

є перспективним вибором для проєктів, які потребують кросплатформеного підходу з високою продуктивністю та надійністю.

1.5 ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було детально розглянуто основні аспекти теоретичних основ розробки гібридних мобільних додатків з використанням Kotlin Multiplatform (KMP). Спочатку я проаналізував сучасні тенденції в розробці мобільних додатків, акцентуючи увагу на різниці між нативними, кросплатформеними та гібридними підходами.

Огляд технологій розробки мобільних додатків показав, що Kotlin Multiplatform має значну перевагу над іншими підходами завдяки можливості використання спільного коду для кількох платформ із збереженням нативного інтерфейсу користувача та продуктивності. Це дозволяє ефективно поєднувати переваги кросплатформеного підходу зі збереженням високої якості нативного досвіду користувача.

Детальний аналіз мови програмування Kotlin та платформи KMP показав, що Kotlin є однією з найсучасніших і найефективніших мов для розробки мобільних додатків завдяки своїй лаконічності, безпеці та повній сумісності з Java. Kotlin Multiplatform дозволяє розробникам використовувати потужності мови Kotlin для створення багатоплатформенних додатків, що значно зменшує витрати часу на розробку та підтримку.

Додатково, було розглянуто можливості використання Compose Multiplatform (CMP) як інтегрованого рішення для створення інтерфейсу користувача. CMP забезпечує значну перевагу, дозволяючи розробникам створювати уніфікований UI-код для кількох платформ. Це не лише зменшує час і витрати на розробку, але й забезпечує узгодженість дизайну та поведінки на різних пристроях. CMP також підтримує налаштування специфічних компонентів для кожної платформи, що дозволяє зберегти нативний досвід користувача.

Попри певні виклики, такі як потреба в додатковій оптимізації для застарілих пристроїв або обмеження через ранню стадію розвитку, CMP у поєднанні з KMP демонструє значний потенціал для вирішення актуальних завдань кросплатформеної розробки. Інструменти CMP та KMP розширюють можливості розробників, дозволяючи створювати продуктивні та гнучкі мобільні додатки з мінімальними витратами часу та ресурсів.

Таким чином, Kotlin Multiplatform у поєднанні з Compose Multiplatform є перспективним рішенням для сучасних потреб мобільної розробки. Ця технологія забезпечує ефективний баланс між швидкістю розробки, оптимізацією витрат і забезпеченням високої якості додатків, що працюють на різних платформах. Її впровадження може стати ключовою конкурентною перевагою для компаній, які прагнуть адаптуватися до швидких змін на ринку та розширювати свій вплив через багатоплатформові рішення.

2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ РОЗРОБКИ ДОДАТКУ ДЛЯ МЕДИЧНОГО СТРАХУВАННЯ

2.1 Аналіз предметної області медичного страхування

Медичне страхування є однією з найважливіших складових системи охорони здоров’я, оскільки забезпечує фінансовий захист громадян у випадку захворювання, нещасного випадку або потреби в отриманні медичних послуг. У сучасному світі страхові компанії пропонують різноманітні програми медичного страхування, які можуть покривати різні види медичних послуг, починаючи від профілактичних оглядів і закінчуючи дорогими операціями. Аналіз предметної області медичного страхування допомагає визначити ключові функціональні вимоги до розробки додатку, що забезпечить ефективне управління медичними полісами, комунікацію з користувачами та зручність користування.

Медичне страхування – це система взаємовідносин між страхувальником (страховою компанією) та застрахованою особою, яка передбачає надання медичних послуг або компенсацію витрат на лікування в разі виникнення страхового випадку. Для управління цими процесами компанії використовують різні технологічні рішення, зокрема мобільні додатки, що надають користувачам зручний доступ до полісів, інформації про доступні послуги, мережу клінік і медичних закладів.

Основні поняття, що стосуються медичного страхування, включають:

1. Страхувальник – компанія, яка надає послуги страхування.

2. Застрахована особа – фізична або юридична особа, яка отримує послуги страхування і є власником страхового полісу.

3. Страховий поліс – документ, який підтверджує договір між страхувальником і застрахованою особою та визначає обсяг медичних послуг, які покриваються.

Існують різні види програм медичного страхування, які можуть мати різні рівні покриття та специфіку надання послуг:

- Державне медичне страхування. Це базовий рівень страхування, який зазвичай забезпечується державою для всіх громадян. У рамках цього страхування покриваються основні медичні послуги, такі як діагностика, лікування та профілактика поширених захворювань.

- Приватне медичне страхування. Приватне страхування зазвичай пропонує розширений набір медичних послуг, включаючи спеціалізовані огляди, процедури, дорогі операції, стоматологічні послуги тощо.

- Добровільне медичне страхування (ДМС). Ця форма страхування пропонує індивідуальні програми для тих, хто хоче отримати додаткові медичні послуги, які не покриваються державним чи стандартним приватним полісом. Програми ДМС дозволяють вибирати додаткові опції, наприклад, страхування в разі нещасних випадків або покриття витрат на лікування за кордоном.

Основними учасниками процесу медичного страхування є:

1. Страхові компанії – надають послуги страхування, пропонуючи різні страхові продукти і програми. Страхова компанія відповідає за оцінку ризиків, управління полісами, здійснення виплат у разі настання страхових випадків та обслуговування клієнтів.

2. Медичні заклади – лікарні, клініки, лабораторії та інші заклади охорони здоров'я, що надають медичні послуги за рахунок страхових полісів.

3. Клієнти (застраховані особи) – фізичні або юридичні особи, які придбають страхові поліси та користуються медичними послугами.

4. Посередники (агенти, брокери) – особи або організації, що здійснюють продаж страхових полісів, надають консультації та допомагають клієнтам з вибором програм страхування.

Медичне страхування передбачає виконання ряду ключових процесів, що мають бути враховані при розробці мобільного додатку:

1. Інформація про страхові послуги. Клієнт має доступ до повного переліку страхових послуг і програм, що надаються. У додатку представлені детальні описи

страхових умов, покриття та переваг кожної програми, що дозволяє користувачеві ознайомитися з усією необхідною інформацією.

2. Подача заявки на страхування. Користувач може подати заявку на оформлення страхового полісу безпосередньо через додаток. У заявці вказуються особисті дані та необхідна інформація для подальшого узгодження деталей із представником страхової компанії.

3. Доступ до мережі медичних закладів. Застраховані особи повинні мати доступ до інформації про клініки та лікарні, з якими співпрацює страхова компанія. Додаток має надавати можливість перегляду карти клінік, їх контактної інформації та переліку доступних послуг.

4. Подача заяв на страхові виплати. У разі виникнення страхового випадку користувач повинен мати можливість подати заявку на страхову виплату через додаток, завантажуючи необхідні документи і отримуючи інформацію про стан обробки заявки.

5. Сповіщення про оновлення полісу та терміни оплати. Клієнти повинні отримувати нагадування про терміни закінчення полісу, можливість продовження страхування та інші важливі повідомлення.

Медичне страхування має кілька специфічних викликів та вимог, які повинні бути враховані під час розробки мобільного додатку:

1. Безпека даних. Медична інформація є конфіденційною, тому додатки, що працюють з нею, повинні відповідати високим стандартам захисту даних, включаючи використання шифрування, автентифікації та надійних способів зберігання інформації.

2. Швидкість доступу до інформації. Додаток повинен забезпечувати швидкий доступ до інформації про поліси, медичні заклади, історію лікування та страхові випадки, щоб користувачі могли оперативно отримувати необхідну інформацію.

3. Підтримка користувачів. Оскільки медичне страхування може бути складною темою для багатьох користувачів, додаток має передбачати можливість отримання консультацій, відповіді на поширені питання та підтримку у вирішенні

Мобільний додаток для медичного страхування грає важливу роль у забезпеченні зручного доступу до страхових послуг та управління полісами. Користувачі можуть отримувати всю необхідну інформацію про свої страхові поліси, записуватися на прийоми до лікарів та отримувати консультації в будь-який зручний час і з будь-якого місця.

Застосування сучасних технологій у сфері медичного страхування дозволяє підвищити якість обслуговування клієнтів, знизити витрати на адміністрування страхових послуг та забезпечити більшу прозорість процесів. Це, в свою чергу, підвищує довіру клієнтів до страхових компаній і стимулює їх до активнішого використання страхових послуг.

Аналіз предметної області медичного страхування показує, що мобільні додатки можуть суттєво спростити взаємодію між страхувальником та застрахованою особою, забезпечуючи зручний доступ до послуг, управління полісами та комунікацію з медичними закладами. Врахування особливостей предметної області дозволяє чітко визначити вимоги до функціональності додатку та забезпечити його ефективне використання у сфері медичного страхування.

2.2 Визначення вимог до функціональності додатку

Визначення вимог до функціональності є ключовим етапом у процесі розробки мобільного додатку для медичного страхування. Цей етап включає визначення основних завдань додатку, які будуть забезпечувати потреби користувачів, спрощувати взаємодію зі страховими компаніями та медичними закладами, а також гарантувати зручне управління медичними полісами.

Функціональні вимоги до додатку можна поділити на кілька ключових категорій, які включають базові, специфічні для медичного страхування функції, а також вимоги до зручності використання та безпеки даних.

1. Авторизація користувачів:

-Авторизація: користувачі можуть увійти до додатку, ввівши свою електронну адресу та пароль. Додаток підтримує збереження сесії для забезпечення зручного доступу без необхідності повторного введення даних.

-Відновлення доступу: у разі втрати паролю користувач може скористатися функцією відновлення доступу, отримавши тимчасовий пароль або посилання для зміни паролю на зареєстровану електронну адресу.

2. Управління полісами:

- Користувачі мають можливість переглядати всі доступні страхові поліси, включаючи детальну інформацію про кожен поліс.

3. Перегляд історії страхових випадків:

- Користувачі мають доступ до історії страхових випадків, що включає інформацію про медичні послуги, отримані на основі їхнього страхового полісу, а також про виплати за страховими випадками.

- Історія повинна містити деталі кожного випадку, включаючи дату, тип послуг, суму покриття та статус обробки запиту.

4. Подача заяв на страхові виплати:

- Додаток повинен забезпечувати можливість подачі заяв на страхові виплати прямо з мобільного пристрою. Користувачі повинні мати змогу заповнити форму заявки, завантажити необхідні документи (довідки з лікарні, рахунки тощо) та відправити її для розгляду.

– користувачі повинні отримувати сповіщення про етапи розгляду заявки (прийнято, обробляється, схвалено або відхилено).

5. Пошук та навігація до медичних закладів

- Користувачі мають доступ до списку медичних закладів, з якими співпрацює страхова компанія в радіусі 5 км.

- Кожен медичний заклад повинен бути представлений детальною інформацією, включаючи контактні дані, години роботи, доступні послуги та рейтинг.

6. Онлайн-консультації:

- Додаток має функцію онлайн-консультацій з медичними спеціалістами. Це може включати текстові чати, відеоконференції.

- Консультації можуть бути частиною страхового покриття або надаватися як додаткова послуга за окрему оплату.

7. Нагадування та сповіщення:

- Користувачі повинні отримувати автоматичні сповіщення про наближення закінчення терміну дії страхового полісу, необхідність оплати рахунків, подачу заявок на виплати або наявність нових пропозицій страхових полісів.

- Сповіщення повинні бути налаштовуваними, щоб користувач міг вибирати, які саме нагадування йому потрібні, і в якому форматі їх отримувати (push-сповіщення, SMS, email).

8. Доступ до медичних документів:

- Користувачі повинні мати можливість зберігати та переглядати медичні документи, пов’язані з їхнім страхуванням: медичні звіти, рецепти, результати аналізів тощо. Це дозволить їм мати весь необхідний пакет документів під рукою у будь-який момент.

- Важливо забезпечити можливість завантаження цих документів з медичних закладів, які є партнерами страхової компанії.

Окрім функціональних вимог, важливо забезпечити зручність використання додатку, що є критичним фактором для медичних страхових рішень, оскільки вони повинні бути простими у використанні для всіх вікових груп користувачів.

1. Інтуїтивний інтерфейс. Додаток повинен мати простий і зрозумілий інтерфейс з чітко структурованими розділами. Користувачі повинні легко знаходити потрібну інформацію і швидко виконувати основні операції.

2. Адаптивність. Додаток повинен підтримувати різні розміри екранів, включаючи смартфони, планшети, а також надавати якісний інтерфейс для користувачів з обмеженими можливостями (підтримка великих шрифтів, голосових команд тощо).

3. Швидка робота. Додаток повинен швидко завантажуватися і працювати без затримок навіть при великій кількості інформації або запитах до серверів. Оскільки медичні дані є критично важливими, затримки у їхньому отриманні можуть негативно вплинути на користувача.

Безпека є критично важливим аспектом при роботі з медичними і фінансовими даними користувачів.

1. Шифрування даних. Усі дані, що передаються між додатком і сервером, повинні бути захищені за допомогою сучасних протоколів шифрування (TLS/SSL). Це забезпечить захист персональних даних від несанкціонованого доступу.

2. Аутентифікація та авторизація. Додаток повинен забезпечувати надійні механізми аутентифікації користувачів, включаючи підтримку багатофакторної автентифікації для критичних операцій, таких як зміна особистих даних або подача заяв на страхові виплати.

3. Захист медичних документів. Медичні документи користувачів повинні зберігатися у захищеному вигляді і бути доступні лише для тих, хто має на це право. Необхідно забезпечити захист від витоків інформації і відповідати вимогам

Додаток повинен передбачати можливість звернення до служби підтримки для вирішення технічних проблем або отримання консультацій з питань страхування.

Визначення функціональних вимог до мобільного додатку для медичного страхування є важливим етапом розробки, який дозволяє забезпечити повну відповідність потребам користувачів. Основні функції, такі як управління полісами, подача заяв на виплати, доступ до медичних закладів, сповіщення і захист даних, є ключовими для успішного використання додатку. Правильна реалізація цих вимог забезпечить зручний доступ до страхових послуг і підвищить рівень задоволеності клієнтів.

2.3 Опис використаних технологій та інструментів:

Для розробки додатку для медичного страхування використовуються сучасні технології та інструменти, що забезпечують ефективність, продуктивність та безпеку мобільного додатку. Основними технологіями та інструментами, які були використаними в цьому проекті, є Kotlin Multiplatform для кросплатформеної розробки, Android Studio та Xcode для розробки під Android та iOS відповідно, інтеграція з бекенд-сервісами для зберігання та обробки даних, а також локальні та хмарні бази даних для управління медичними полісами та іншою інформацією.

Крім того, для створення уніфікованого інтерфейсу користувача було використано Compose Multiplatform (CMP). Ця технологія дозволила реалізувати спільний код для UI, що забезпечує узгодженість дизайну та поведінки додатку на різних платформах, скоротивши час на розробку інтерфейсу [15].

2.3.1 Kotlin Multiplatform

Kotlin Multiplatform (KMP) — це технологія, яка дозволяє розробляти кросплатформені додатки, використовуючи єдину кодову базу для бізнес-логіки, але з можливістю створення нативних інтерфейсів для кожної окремої платформи (Android, iOS). KMP розроблена компанією JetBrains і підтримується Google, що робить її оптимальним вибором для розробки мобільних додатків з можливістю одночасної підтримки кількох платформ.

Головна перевага KMP полягає в тому, що значна частина коду (наприклад, робота з API, логіка обробки даних) пишеться один раз і використовується для всіх платформ. Однак інтерфейси користувача залишаються нативними, що дозволяє забезпечити якісний користувацький досвід (UX), характерний для кожної конкретної платформи.

Особливо в контексті медичного страхування, KMP дозволяє оптимізувати процес розробки додатку, повторно використовуючи код для ключових функцій,

таких як управління страховими полісами, перевірка статусу заявок, сповіщення та взаємодія з бекендом [6].

Основні переваги Kotlin Multiplatform:

- Спільний код: значна частина логіки додатку пишеться один раз і використовується на всіх платформах.

- Нативні інтерфейси: інтерфейси залишаються нативними для кожної платформи, що забезпечує кращий користувацький досвід.

- Гнучкість: KMP дозволяє інтегрувати кросплатформений код у вже існуючі нативні проекти без необхідності повного переписування додатку.

2.3.2 Compose Multiplatform (CMP)

Compose Multiplatform (CMP) — це сучасна бібліотека для створення кросплатформеного інтерфейсу користувача (UI) у межах екосистеми Kotlin Multiplatform. CMP дозволяє розробникам писати спільний код для UI-компонентів, який працює на різних платформах, таких як Android, iOS, Desktop і Web. CMP базується на принципах Jetpack Compose для Android, що гарантує зручний декларативний підхід до створення інтерфейсу.

Головною перевагою CMP є можливість створення узгодженого дизайну та поведінки додатку на різних платформах, що значно спрощує процес розробки. Завдяки CMP розробники можуть створювати елементи інтерфейсу, такі як форми, списки чи кнопки, один раз і використовувати їх на всіх платформах без дублювання коду.

Особливо в контексті додатків для медичного страхування CMP дозволяє ефективно реалізовувати користувацький інтерфейс для таких функцій, як відображення інформації про страхові поліси, моніторинг статусу заявок чи надання доступу до переліку медичних закладів [табл.1.2].

Основні переваги Compose Multiplatform:

Спільний код для UI: CMP дозволяє писати спільний інтерфейс для кількох платформ, що зменшує витрати на розробку та забезпечує узгодженість дизайну.

Можливість кастомізації: Для кожної платформи можна адаптувати UI-компоненти, щоб відповідати специфічним вимогам операційної системи.

Простота інтеграції: CMP можна легко інтегрувати в проекти, побудовані на базі Kotlin Multiplatform, для створення сучасного, узгодженого інтерфейсу користувача [табл.1.2].

Таблиця 1.2 Порівняння розробки UI

Критерій	**Традиційна розробка UI**	**Compose Multiplatform (CMP)**
Розробка UI	Окремий код для кожної платформи	Спільний код для UI на всіх платформах
Кількість кодової бази	Декілька (Android, iOS, Desktop, Web)	Одна спільна кодова база
Швидкість розробки	Більш повільна (потрібно дублювати роботу)	Швидка завдяки єдиному підходу
Користувацький інтерфейс	Нативний для кожної платформи	Уніфікований, але з можливістю кастомізації
Підтримка платформ	Потрібна розробка для кожної ОС окремо	Android, iOS, Desktop, Web одночасно
Витрати на підтримку	Високі (багато окремих кодів для UI)	Низькі (спільна кодова база)
Гнучкість	Висока для платформи, але обмежена загалом	Висока із можливістю кастомізації
Інтеграція з нативними функціями	Повна, але складна в реалізації	Легка інтеграція з нативними API

CMP у поєднанні з Kotlin Multiplatform стає потужним інструментом для розробки багатоплатформних додатків, забезпечуючи баланс між спільним кодом і платформо-залежними налаштуваннями.

2.3.3 Android Studio та Xcode

Android Studio — це офіційне середовище розробки для Android-додатків, засноване на IntelliJ IDEA. Android Studio надає всі необхідні інструменти для

написання, тестування, налагодження та профілювання додатків для Android. Вона також підтримує Kotlin Multiplatform, що робить його ідеальним вибором для розробки кросплатформеного додатку з нативним інтерфейсом на Android.

Основні функції Android Studio:

- Емулятор Android для тестування додатків на різних пристроях і версіях Android.

- Gradle — система збірки, яка дозволяє автоматизувати процеси збірки, тестування і доставки додатку.

- Layout Editor для створення та редагування інтерфейсу користувача через інтуїтивний візуальний редактор.

- Інструменти для відлагодження: профілювання продуктивності додатку, управління пам’яттю та інше.

Додаткові аспекти Android Studio:

Підтримка Jetpack Compose:

Android Studio повністю підтримує Jetpack Compose — сучасний інструмент для декларативної розробки UI на Android. Це є важливим, якщо в проєкті використовується Compose Multiplatform (CMP), оскільки інструмент дозволяє розробникам ефективно тестувати та налагоджувати спільний UI-код для Android.

Можливості інтеграції з бекендом:

Android Studio має вбудовані інструменти для роботи з REST API через бібліотеки, такі як Retrofit або Ktor. Це особливо корисно для медичного додатку, який інтегрується з сервером для отримання даних про поліси та заявки [4].

Інструменти для тестування:

Android Studio підтримує написання UI-тестів (Espresso) та юніт-тестів (JUnit). Це дозволяє перевіряти якість бізнес-логіки та інтерфейсу додатку в умовах, максимально наближених до реальних [табл.1.3].

Xcode — це офіційне інтегроване середовище розробки для додатків під iOS, macOS, tvOS та watchOS. Воно надає інструменти для розробки, тестування і відлагодження додатків на iOS. У контексті розробки додатку для медичного

страхування, Xcode використовується для створення нативних інтерфейсів під iOS та інтеграції з бекендом [табл.1.3].

Основні функції Xcode:

- Interface Builder для створення інтерфейсів користувача через візуальний редактор.

- Instruments для профілювання продуктивності додатку на рівні процесора, пам’яті та споживання ресурсів.

- Інтеграція з iOS SDK для доступу до нативних можливостей iOS.

Додаткові аспекти Xcode:

SwiftUI та його взаємодія з CMP:

Xcode підтримує SwiftUI — фреймворк для декларативного створення інтерфейсу користувача. Це дозволяє розробникам створювати UI-компоненти для iOS, які можуть бути доповненням до CMP для платформо-залежних частин [15].

Симулятор iOS:

Симулятор Xcode забезпечує можливість тестування додатків на різних моделях iPhone та iPad. Це особливо корисно для перевірки, як додаток виглядає і працює на пристроях із різними розмірами екранів.

Інтеграція з нативними сервісами:

Xcode дозволяє використовувати специфічні сервіси iOS, наприклад, CloudKit для хмарного зберігання даних або HealthKit для обробки медичної інформації. Ці інструменти можуть бути важливими для зберігання й обробки страхових даних.

Таблиця 1.3 Порівняння можливостей Android Studio та Xcode

Критерій	Android Studio	Xcode
Платформа	Android	iOS, macOS, watchOS, tvOS
Мова програмування	Kotlin, Java	Swift, Objective-C
Інструменти для UI	Layout Editor	Interface Builder
Профілювання продуктивності	Android Profiler	Instruments
Вбудовані емулятори	Так (Android Virtual Device)	Так (iOS Simulator)
Підтримка інтеграції KMP	Повна підтримка	Обмежена підтримка

2.3.4 Інтеграція з бекенд-сервісами

Для роботи додатку з медичними полісами, управління даними користувачів та подачею заяв на страхові виплати необхідно забезпечити надійну інтеграцію з бекенд-сервісами. Бекенд-сервіси відіграють ключову роль у зберіганні та обробці даних, оскільки вони забезпечують:

- Зберігання інформації про поліси. Усі дані про страхові поліси користувачів (типи полісів, терміни дії, страхові виплати) повинні зберігатися на сервері, а додаток повинен мати доступ до цих даних для відображення в режимі реального часу.

- Обробка заяв на страхові виплати. Користувачі можуть подавати заявки через мобільний додаток, але їхня обробка здійснюється на серверній стороні. Додаток повинен бути здатен відправляти документи і отримувати статус заявки.

- Сповіщення та нотифікації. Бекенд-сервіс також відповідальний за відправку сповіщень користувачам про важливі події, такі як закінчення терміну дії полісу або статус обробки заявок.

Для інтеграції з бекендом було використано такі технології: Ktor — це легковаговий фреймворк, розроблений компанією JetBrains, який використовується для створення клієнтської та серверної частин додатків. У контексті цього проєкту Ktor застосовується для взаємодії з бекендом через REST

API або GraphQL. Завдяки своїй підтримці Kotlin Multiplatform, Ktor дозволяє писати спільний код для роботи з мережею, що зменшує кількість платформо-залежних рішень і спрощує процес розробки [15].

Основні переваги Ktor:

- Підтримка клієнтської та серверної частин, що робить його універсальним для розробки.

- Легке налаштування для роботи з HTTP-запитами, автентифікацією та обробкою помилок.

- Ідеальна інтеграція з Kotlin Multiplatform, що дозволяє повторно використовувати код на різних платформах [4].

Firebase Cloud Messaging (FCM) — це сервіс, який дозволяє надсилати push-сповіщення на мобільні пристрої. У додатку для медичного страхування FCM може використовуватися для інформування користувачів про важливі події, такі як оновлення статусу їхніх заявок, нагадування про термін дії полісу чи сповіщення про нові пропозиції [7].

Основні переваги FCM:

- Надійність і масштабованість — сервіс здатний обробляти великий обсяг повідомлень у реальному часі.

- Підтримка персоналізованих повідомлень, що дозволяє надсилати релевантний контент залежно від дій користувача.

- Інтеграція з бекендом для автоматичного надсилання сповіщень на основі тригерів (наприклад, подання заявки чи зміна статусу).

2.3.5 Бази даних та засоби збереження даних

Для зберігання даних у додатку використовуються як локальні, так і хмарні бази даних. Збереження даних є ключовою функцією для додатку медичного страхування, оскільки потрібно зберігати інформацію про поліси, страхові випадки, документи, профілі користувачів тощо.

- Realm — це сучасна база даних для роботи з локальними даними, яка забезпечує високу продуктивність і простоту використання. Realm дозволяє зберігати інформацію про користувацькі налаштування, історію запитів і інші дані без необхідності написання складного SQL-коду. Завдяки своїй гнучкості та підтримці автоматичної синхронізації, Realm є ефективним вибором для додатків, які потребують роботи з великими обсягами локальних даних [8].

- SharedPreferences — інструмент для збереження простих даних (налаштування, токени доступу) у форматі ключ-значення. Це рішення підходить для легких налаштувань користувача, таких як мова додатку або стан авторизації.

- Keychain — це інструмент для безпечного зберігання конфіденційних даних на пристроях iOS. Використовуючи Keychain, додаток може зберігати такі дані, як токени авторизації, логіни чи інші чутливі дані, які необхідно захистити від несанкціонованого доступу. У контексті мобільного додатку для медичного страхування Keychain забезпечує додатковий рівень безпеки для управління даними користувачів, що є критично важливим у медичній сфері.

2. Хмарні бази даних:

- Firebase Firestore — це хмарна база даних від Google, яка дозволяє зберігати дані про користувачів, медичні поліси та заявки на страхові виплати в реальному часі. Вона забезпечує зручну синхронізацію даних між різними пристроями, що дозволяє користувачам отримувати доступ до актуальної інформації незалежно від того, де вони знаходяться [7].

3. Захист даних:

- Важливо забезпечити шифрування даних як на рівні локального збереження, так і під час передачі інформації між додатком і сервером. Шифрування на стороні клієнта і сервера дозволяє захистити медичні дані та конфіденційну інформацію про користувачів.

Для розробки мобільного додатку для медичного страхування використовуються сучасні технології та інструменти, які забезпечують ефективність, гнучкість і безпеку. Kotlin Multiplatform дозволяє розробляти додаток для різних платформ із мінімальними витратами на дублювання коду.

Android Studio та Xcode надають повний набір інструментів для розробки нативних інтерфейсів і функцій. Інтеграція з бекенд-сервісами забезпечує надійний обмін даними між додатком і сервером, а бази даних забезпечують збереження даних і можливість їхнього доступу в реальному часі.

2.4 ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було детально розглянуто технології та інструменти, які використовуються для розробки мобільного додатку для медичного страхування. Основною метою цього розділу було описати технічні рішення, що забезпечують ефективність, продуктивність і безпеку додатку.

Аналіз предметної області медичного страхування дозволив визначити ключові вимоги до функціональності додатку: управління страховими полісами, подача заяв на виплати, інтеграція з медичними закладами та реалізація додаткових функцій, таких як інформування користувачів через нотифікації. Ці функції є основними для користувачів додатку, забезпечуючи зручний доступ до медичних послуг і управління інформацією про страхування.

Одним із важливих технічних рішень, описаних у розділі, є використання Kotlin Multiplatform (KMP). Ця технологія дозволяє розробляти додаток для кількох платформ одночасно, використовуючи спільну кодову базу для бізнес-логіки. Це значно зменшує витрати на розробку та підтримку, забезпечуючи при цьому нативний користувацький досвід для кожної платформи. Доповненням до цього підходу є використання Compose Multiplatform (CMP), яке дозволяє створювати уніфікований інтерфейс користувача для кількох платформ, зберігаючи узгодженість дизайну та гнучкість кастомізації.

Android Studio та Xcode залишаються ключовими інструментами для розробки платформо-залежних компонентів додатку. Вони забезпечують створення, налагодження та тестування додатків, а також інтеграцію з бекенд-сервісами. Завдяки цим середовищам розробки вдається підтримувати стабільність

роботи додатку на різних пристроях і забезпечувати доступ до нативних можливостей платформ.

Інтеграція з бекенд-сервісами є критичним компонентом функціональності додатку. У розділі описано використання технологій, таких як Ktor для взаємодії з REST API, що забезпечує надійний і безпечний обмін даними між сервером і додатком. Firebase Cloud Messaging (FCM) використовується для відправки push-сповіщень, що дозволяє оперативно інформувати користувачів про статус заявок, оновлення полісів або інші важливі події.

Для збереження даних у додатку використовуються сучасні бази даних. Для локального збереження даних було обрано Realm, що забезпечує високу продуктивність і простоту використання. Для хмарного зберігання даних застосовуються рішення на базі Firebase Firestore, які дозволяють синхронізувати інформацію між пристроями користувачів та гарантують її безпеку через шифрування та захищений доступ.

Отже, технології та інструменти, описані у цьому розділі, забезпечують ефективну, безпечну та продуктивну реалізацію мобільного додатку для медичного страхування. Використання Kotlin Multiplatform, Compose Multiplatform, Ktor, Firebase та сучасних баз даних дозволяє створити гнучкий, масштабований додаток, що відповідає вимогам сучасного ринку та забезпечує надійну роботу з даними користувачів.

3 ПРОЦЕС РОЗРОБКИ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ДОДАТКУ

3.1 Збір вимог та проектування архітектури додатку

Процес розробки мобільного додатку починається зі збору вимог і проектування архітектури, оскільки ці етапи є основою для подальших рішень щодо розробки, тестування та підтримки додатку. Збір вимог визначає, які функціональні та нефункціональні характеристики має реалізувати додаток, а проектування архітектури дозволяє оптимально організувати структуру коду, забезпечуючи його гнучкість, продуктивність і зручність у підтримці.

1. Збір вимог

Збір вимог є першим етапом процесу розробки, який допомагає визначити ключові функції додатку та очікування користувачів. Це важливо для створення мобільного додатку, який максимально відповідає бізнес-цілям та потребам кінцевих користувачів. Збір вимог включає кілька основних етапів:

- аналіз цільової аудиторії: визначення потреб користувачів та їхніх очікувань щодо функціональності та користувацького досвіду. Наприклад, для медичного страхового додатку основними вимогами можуть бути зручність доступу до страхових полісів, швидка подача заявок та інтеграція з медичними закладами.

- аналіз бізнес-вимог: визначення ключових бізнес-функцій, таких як управління страховими полісами, подача заявок на виплати, доступ до історії страхових випадків тощо. Ці вимоги забезпечують відповідність додатку бізнес-процесам компанії та підвищують його цінність для користувачів.

- визначення функціональних вимог: створення списку основних функцій додатку, наприклад, функції авторизації, управління профілем, перегляду страхових полісів, подачі заявок на страхові виплати та підтримки сповіщень [11].

- визначення нефункціональних вимог: окрім основних функцій, додаток повинен відповідати вимогам продуктивності, безпеки, зручності використання та масштабованості. Наприклад, медичний страховий додаток повинен забезпечувати

безпеку особистих даних користувачів, мати швидкий час відгуку та підтримувати масштабованість для великої кількості користувачів.

2. Проектування архітектури додатку

Після збору вимог наступним етапом є проектування архітектури, яка допоможе забезпечити ефективне управління кодом та підтримувати високу продуктивність додатку. Архітектура має включати вибір технологій, структурних компонентів і патернів, які найкраще відповідають цілям проекту.

Архітектурні підходи для мобільних додатків на Kotlin Multiplatform:

1.MVVM (Model-View-ViewModel): MVVM є одним із найбільш популярних архітектурних шаблонів для мобільних додатків на Kotlin Multiplatform. Він забезпечує чіткий поділ між бізнес-логікою та відображенням даних, що спрощує тестування та підтримку проєкту. У цьому підході:

-Model відповідає за обробку даних і реалізацію бізнес-логіки.

-ViewModel управляє станом інтерфейсу та взаємодіє з Model для отримання або оновлення даних.

-View відображає дані на екрані та реагує на зміни у ViewModel.

2.Clean Architecture: Clean Architecture забезпечує високу модульність та структурованість додатку, розділяючи його на незалежні рівні. Цей підхід дозволяє легко додавати нові функції, модифікувати існуючі компоненти або інтегрувати інші рішення.

Основні принципи Clean Architecture включають поділ коду на шари:

-Інтерфейс користувача (UI): відповідає за візуальне представлення даних.

-Бізнес-логіка: реалізує основні функціональні можливості додатку.

-Дані: забезпечує зберігання та обробку інформації.

Поєднання Clean Architecture та MVVM дозволяє створити гнучкий і тестований проєкт, де бізнес-логіка і представлення інтерфейсу користувача чітко розмежовані.

Компоненти архітектури KMP-додатків:

1. Спільний модуль (Common Module): містить бізнес-логіку, мережеві запити та взаємодію з базами даних, яка використовується на всіх платформах.

2. Платформо-залежні модулі: містять код, що використовує нативні API для кожної платформи, наприклад, Android SDK для Android і UIKit для iOS. Ці модулі забезпечують інтеграцію з функціями ОС, такими як камера, GPS, пуш-сповіщення тощо.

3. Комунікація між модулями: зв’язок між спільним та нативними модулями здійснюється через механізм expect/actual, що дозволяє викликати нативний код із спільного модуля. Це забезпечує одночасно спільність бізнес-логіки та можливість доступу до платформозалежних функцій.

3.2 Розробка функціональності додатку

Розробка функціональності є одним із ключових етапів створення мобільного додатку, що включає реалізацію основних функцій, визначених на етапі збору вимог. Основна мета цього етапу — забезпечити повний набір функцій, що відповідають потребам користувачів та бізнес-вимогам, а також забезпечити зручність користування, продуктивність і безпеку додатку.

1. Авторизація є однією з основних функцій, яка забезпечує безпечний доступ користувачів до функціоналу додатку.

Авторизація: користувач вводить свою електронну пошту та пароль для входу в додаток. Для зручності використовується функція збереження сесії, що дозволяє уникнути повторного введення даних при наступних входах.

Захист даних: під час авторизації застосовується шифрування даних користувачів для забезпечення безпеки під час передачі та зберігання інформації.

2. Відображення профілю користувача

Відображення профілю дозволяє користувачам переглядати свої особисті дані та інформацію, пов’язану з їхнім обліковим записом.

Перегляд профілю: користувачі мають доступ до інформації, такої як ім’я, контакти, статус страхового полісу та інша персоналізована інформація.

Безпека даних: дані користувача відображаються у захищеному середовищі, з використанням шифрування для забезпечення конфіденційності.

3. Управління страховими полісами

Для додатку медичного страхування ця функціональність є критично важливою:

- Перегляд доступних полісів: користувачам надається доступ до інформації про страхові поліси, що включає види страхування, терміни дії, умови покриття та інші параметри.

4. Подання заявок на страхові виплати

Одна з важливих функцій для користувачів — можливість подавати заявки на страхові виплати прямо з мобільного додатку:

- Форма заявки: користувач заповнює форму, додає опис випадку та необхідні документи.

- Завантаження документів: додаток підтримує завантаження сканів медичних довідок, квитанцій та інших необхідних файлів.

- Відстеження статусу: користувачі отримують сповіщення про статус заявки (прийнято, обробляється, відхилено), що полегшує контроль над процесом розгляду заявки.

5. Карта медичних закладів

Карта медичних закладів надає користувачам зручний спосіб візуалізації місць, які співпрацюють зі страховою компанією.

Карта закладів: інтерактивна карта, яка відображає розташування всіх доступних медичних закладів.

Інформація про заклади: на карті можна переглядати основні дані про заклади, зокрема їхню адресу, контактну інформацію та години роботи.

6. Нагадування та сповіщення

Сповіщення дозволяють користувачам бути в курсі важливих подій та нагадувань:

- Пуш-сповіщення: нагадування про закінчення терміну полісу, можливість оформити новий поліс або сповіщення про нові послуги.

- Налаштування сповіщень: користувачі можуть обирати, чи хочуть вони отримувати сповіщення.

7. Онлайн-консультації

Для підвищення зручності користувачів важливо реалізувати функціональність онлайн-консультацій:

- Текстові чати: можливість звернутися до медичних консультантів, поставити запитання, отримати відповіді у реальному часі.

- Запити на консультації: можливість надсилання запитів у клініки-партнери з подальшим відстеженням статусу [рисунок 3.1].

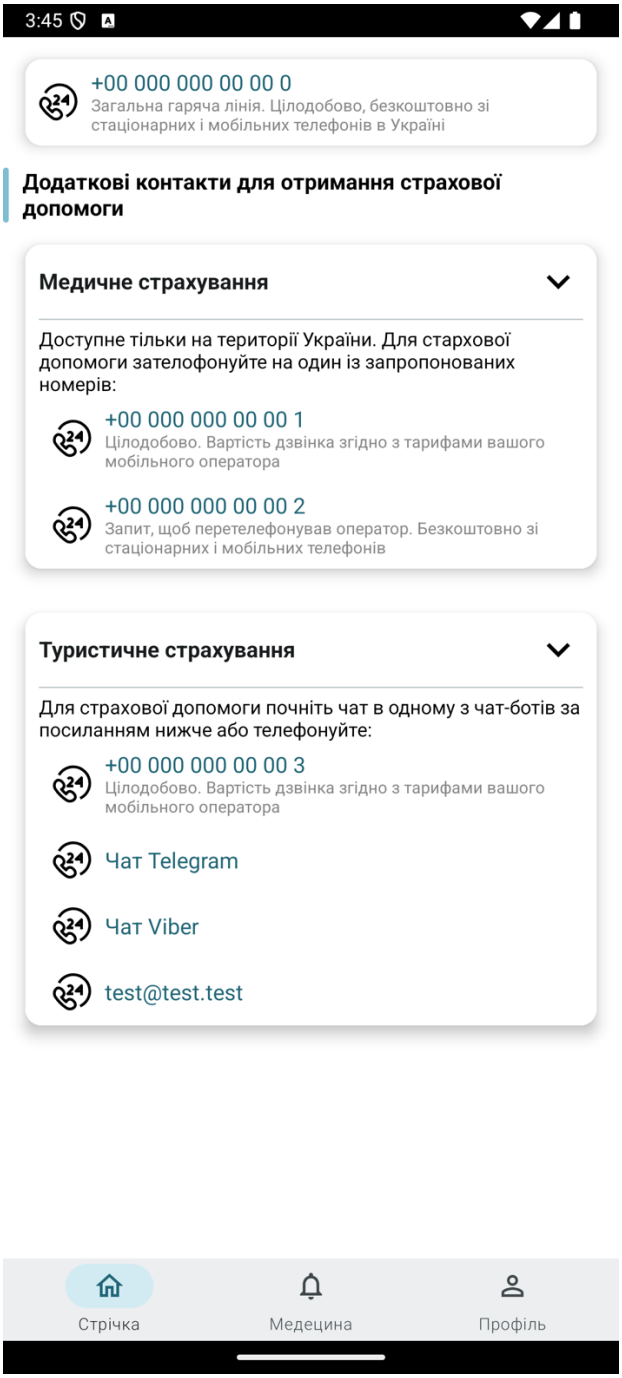


Рисунок 3.1 Контакти для отримання страхової допомоги

3.2.1 Функція автентифікації користувача

Функція автентифікації користувача є основною для додатку, оскільки забезпечує безпечний доступ до персональних даних і функцій. Важливо, щоб користувачі мали зручні та надійні способи входу, а додаток гарантував захист їхніх даних.

Основні аспекти реалізації:

Способи автентифікації: додаток підтримує автентифікацію через електронну пошту або номер телефону, а також реалізує багатофакторну автентифікацію (2FA) для підвищення рівня захисту облікових записів користувачів [11].

Процес автентифікації: після введення логіну та пароля сервер перевіряє дані користувача. У разі успішної перевірки користувач отримує доступ до персональних даних, страхових полісів та інших функцій додатку. Цей процес є ключовим для забезпечення безпеки, оскільки гарантує, що доступ до облікового запису має лише авторизований користувач [рисунок 3.2].

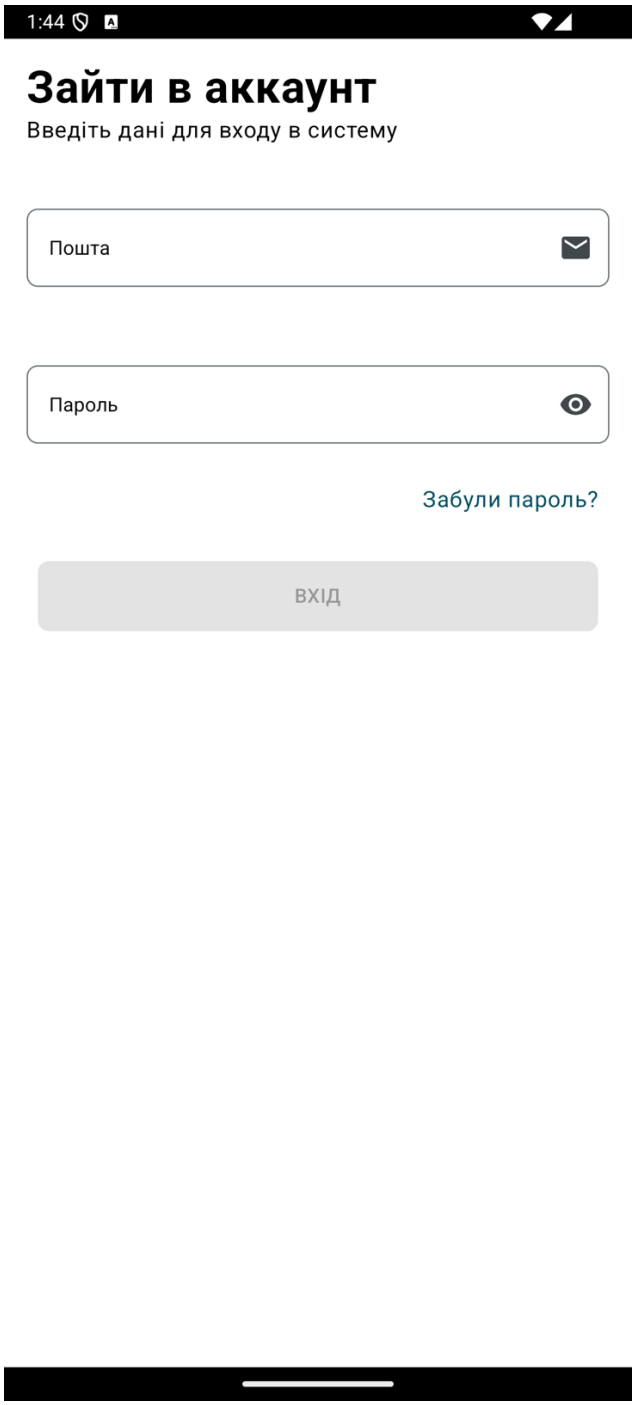


Рисунок 3.2 Успішний вхід у додаток через форму авторизації.

Відновлення доступу: у разі втрати пароля користувач може скористатися функцією відновлення доступу, отримавши тимчасовий пароль або посилання на електронну пошту чи номер телефону [рисунок 3.3].

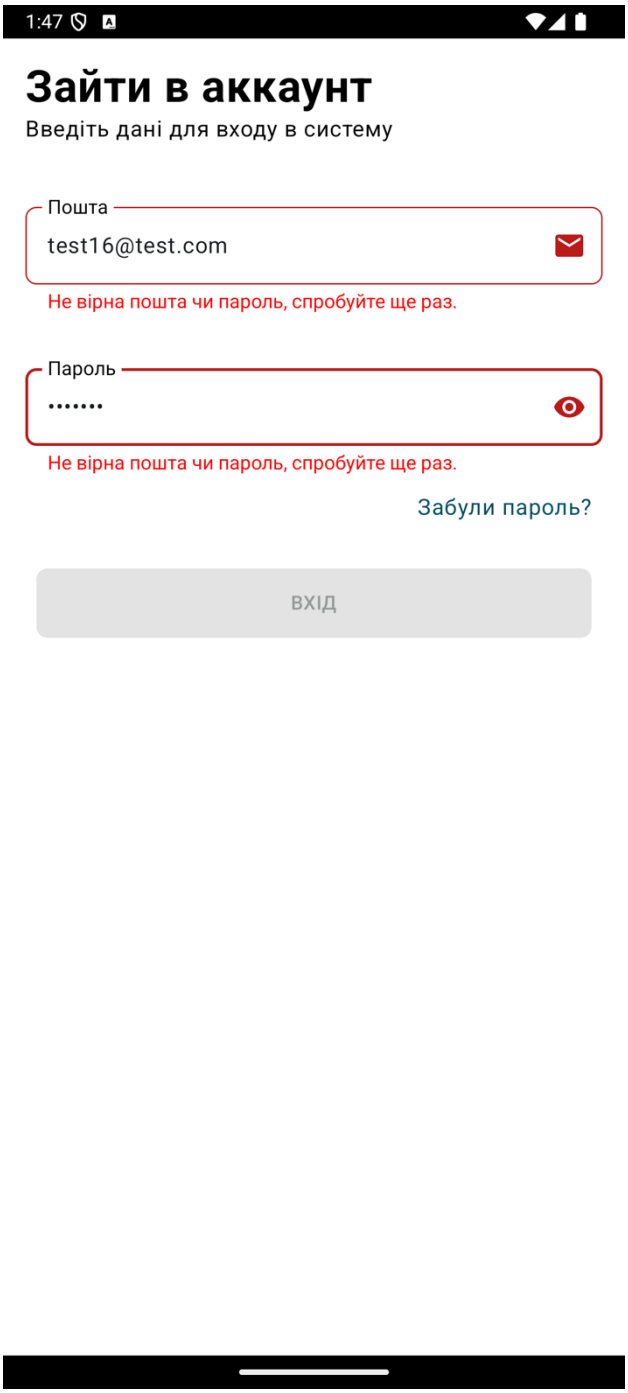


Рисунок 3.3 Повідомлення про помилку у разі невірного вводу даних.

Безпека даних: для гарантування захисту даних користувачів застосовується шифрування інформації під час передачі та зберігання паролів у хешованому вигляді (наприклад, із використанням SHA-256 або bcrypt). Використання токенів автентифікації на основі JSON Web Token (JWT) забезпечує безпечне керування сесіями користувачів [додаток А].

3.2.2 Функція перегляду страхових полісів

Функція перегляду страхових полісів дозволяє користувачам отримувати повну інформацію про доступні страхові продукти, що співпрацюють зі страховою компанією. Завдяки цій функції користувачі мають можливість детально ознайомитися з умовами страхування.

Основні аспекти функціональності:

Основні вимоги: у додатку відображається список доступних полісів, що містять детальну інформацію про кожен продукт. Це забезпечує зручний доступ до інформації для користувачів.

Деталі полісів: кожен поліс містить ключову інформацію, таку як термін дії, сума покриття та умови страхування. Це дозволяє користувачам отримати повне уявлення про характеристики полісів.

Категоризація полісів: додаток надає можливість групувати поліси за категоріями, наприклад, «Здоров'я», «Нещасний випадок». Це допомагає користувачам швидше орієнтуватися серед різних видів страхування.

Однією з основних функцій додатку є можливість перегляду актуальних медичних полісів користувача. В залежності від вибраного страхового плану, користувачі можуть бачити детальну інформацію про свої поліси: тип полісу (базовий або преміум), терміни дії, унікальний медичний ID та інші важливі дані. Інтерфейс додатку реалізовано з використанням **Compose Multiplatform**, що забезпечує єдиний стиль оформлення для обох платформ (Android та iOS), зберігаючи при цьому продуктивність та зручність користування" [рисунок 3.4], [рисунок 3.5].

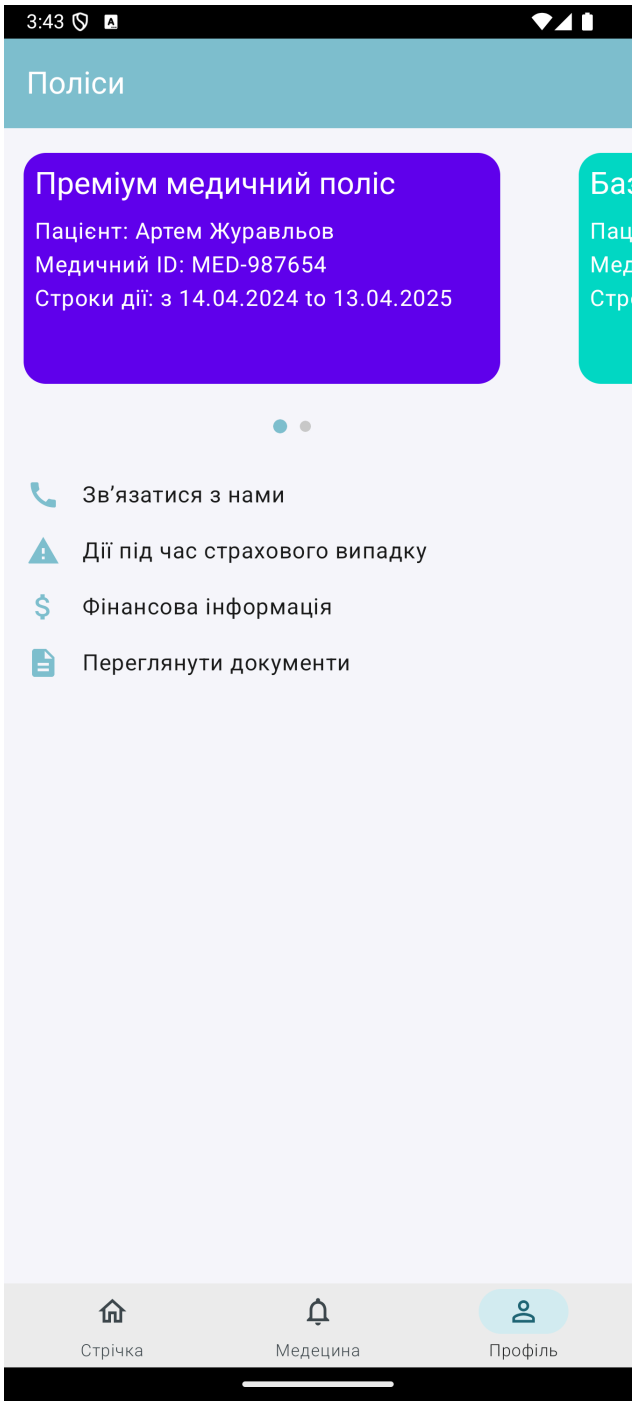


Рисунок 3.4 Відображення преміум медичного полісу з детальною інформацією про пацієнта та терміни дії полісу

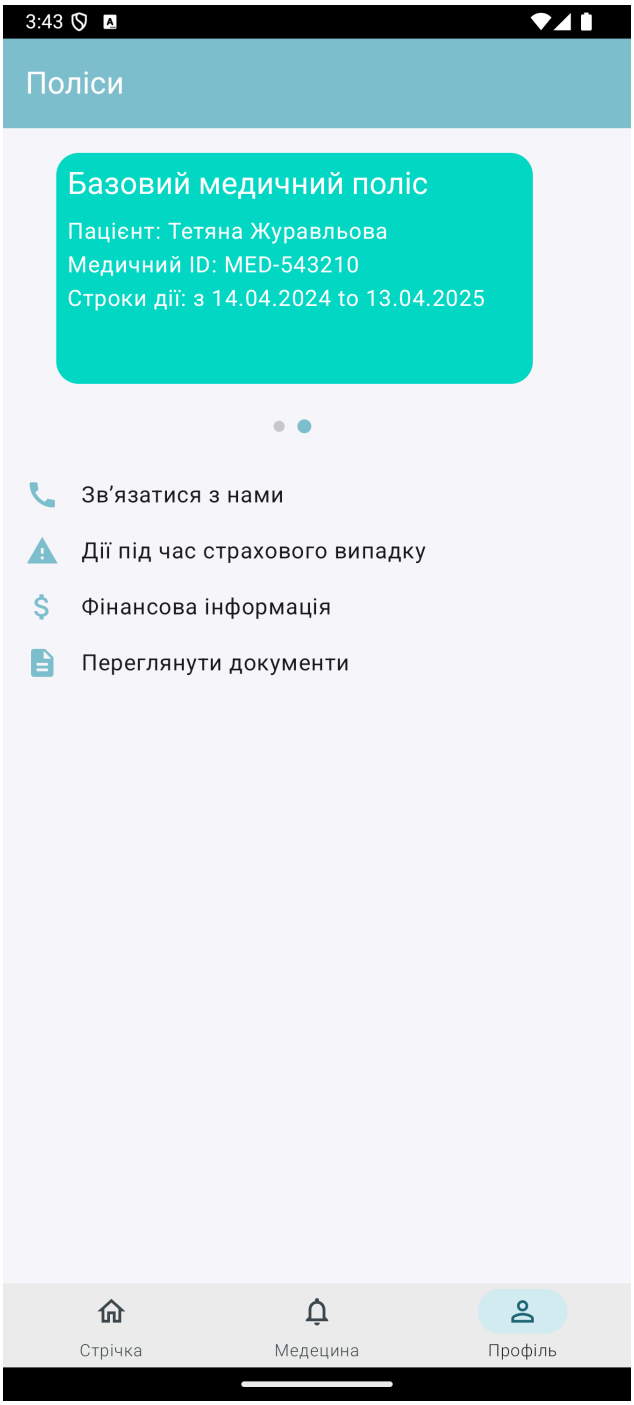


Рисунок 3.5 Відображення базового медичного полісу з інформацією про пацієнта та терміни дії полісу.

Інформація без фільтрації: користувачі переглядають повний перелік полісів без додаткових фільтрів чи персоналізованих рекомендацій, зосереджуючись лише на загальній інформації [додаток А].

3.2.3 Функція сповіщення та нагадування користувача

Ця функція дозволяє користувачам своєчасно отримувати важливі повідомлення та нагадування, пов’язані з використанням їхніх страхових полісів, а також з новими пропозиціями та оновленнями умов страхування. Завдяки сповіщенням користувачі завжди залишаються в курсі актуальної інформації, що підвищує рівень обслуговування та зручність у використанні додатку.

Основні аспекти функціональності:

Основні вимоги: для того щоб забезпечити оперативну комунікацію, додаток підтримує push-сповіщення, які використовуються для нагадувань про закінчення терміну дії полісів та повідомлень про нові можливості або зміни умов страхування. Це дозволяє користувачам бути в курсі важливих подій і змін без необхідності перевіряти додаток вручну.

Типи сповіщень: у додатку передбачені різні види сповіщень, включаючи нагадування про оновлення полісу, повідомлення про нові акції або пропозиції, а також оновлення статусу поданих заявок на страхові виплати. Це допомагає користувачам своєчасно реагувати на необхідні дії, зокрема щодо продовження страхового покриття.

Налаштування сповіщень: користувачі можуть обирати, чи хочуть вони отримувати сповіщення. Налаштування сповіщень: користувачі можуть обирати, чи хочуть вони отримувати сповіщення. Це дозволяє адаптувати функцію сповіщень до індивідуальних потреб кожного користувача.

3.2.4 Функція управління особистим кабінетом користувача

Особистий кабінет користувача є важливою частиною додатку, оскільки він дозволяє користувачам переглядати та змінювати особисті дані, а також забезпечує безпеку та конфіденційність інформації. Ця функціональність створює зручність і гнучкість для користувачів у керуванні своїми даними, а також сприяє персоналізації досвіду використання додатку.

Основні аспекти функціональності:

Безпека даних: додаток використовує надійні методи шифрування для збереження персональної інформації користувачів, що гарантує захист даних від несанкціонованого доступу. Це є особливо важливим для медичного страхування, де обробляється чутлива інформація [рисунок 3.6].

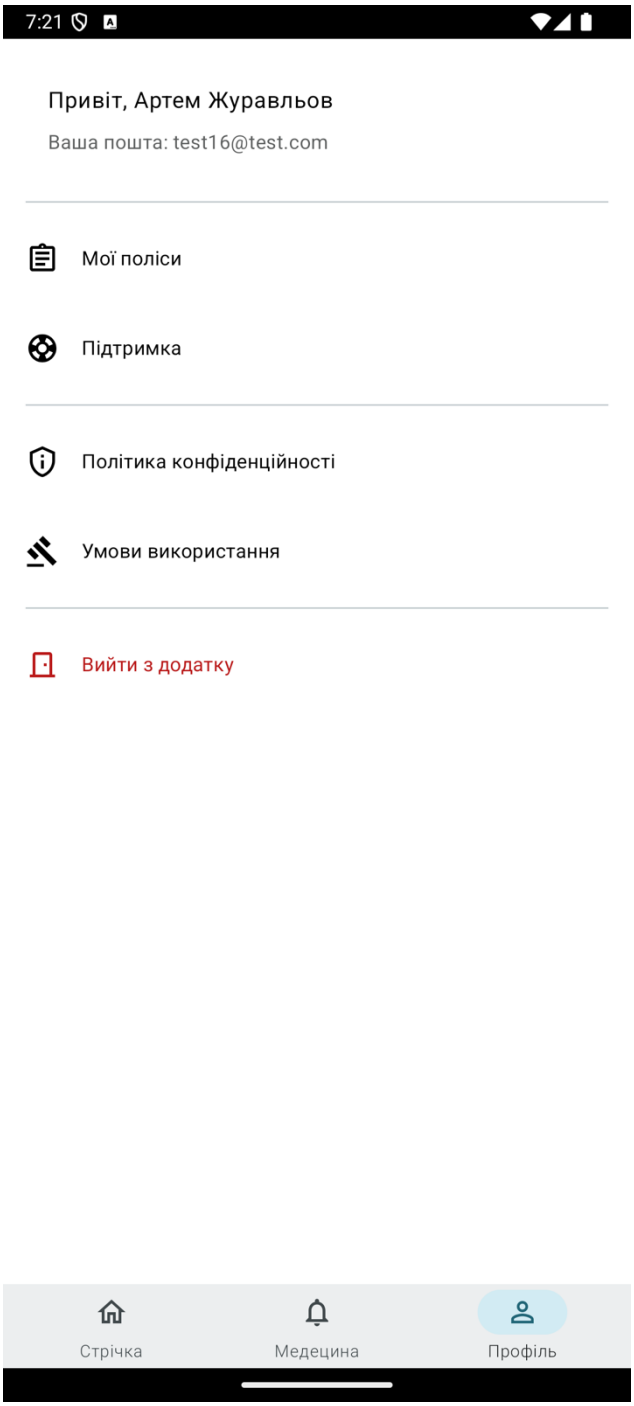


Рисунок 3.6 Особистий кабінет користувача.

3.2.5 Функція відображення карти клінік, які покриває страхова компанія

Функція карти клінік дозволяє користувачам зручно знаходити медичні заклади, які входять до страхового покриття. Вона допомагає користувачам дізнаватися про найближчі клініки, що надають необхідні медичні послуги, а також планувати візит до лікаря з використанням інтегрованих навігаційних сервісів.

Основні аспекти функціональності:

Основні вимоги: список найближчих клінік в радіусі 5км, що дозволяє переглядати найближчі клініки, з урахуванням розташування користувача [рисунок 3.7].

Рисунок 3.7 Відображення карти з медичними закладами для навігації користувача.

Деталі клінік: інформація про клініки, що входять до покриття, включаючи адреси, розклад роботи та доступні спеціалізації [рисунок 3.8].

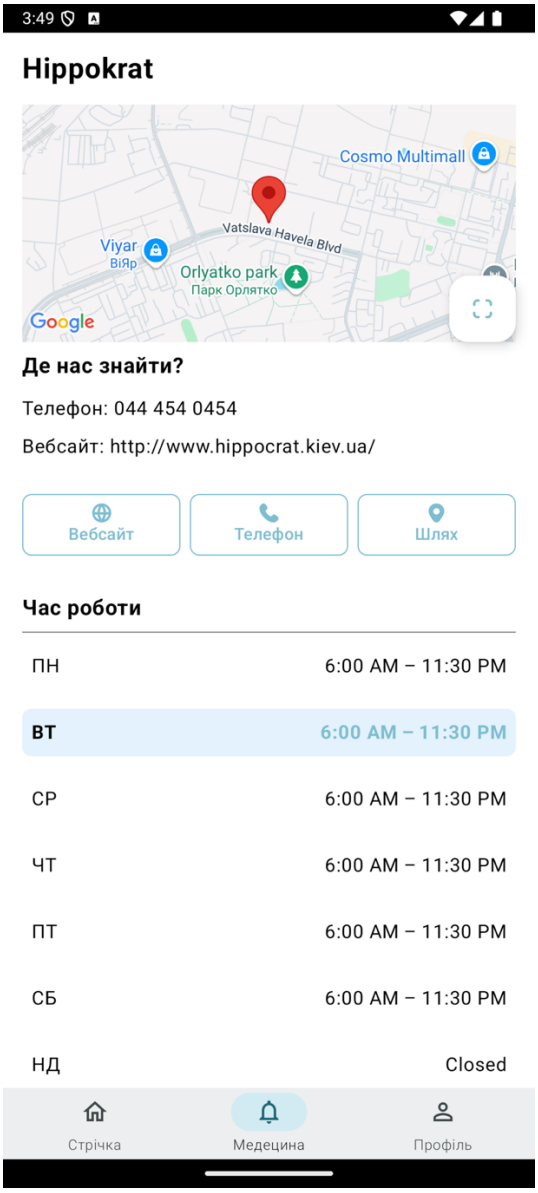


Рисунок 3.8 Детальна інформація про медичний заклад із контактами та часом роботи.

Це допомагає користувачам вибирати заклади, що найкраще відповідають їхнім потребам.

Інтерактивна навігація: завдяки інтеграції з такими навігаційними інструментами, як Google Maps API для Android та MapKit для iOS, користувачі можуть створювати маршрути до клінік із урахуванням поточного місця перебування [рисунок 3.9], [додаток А].

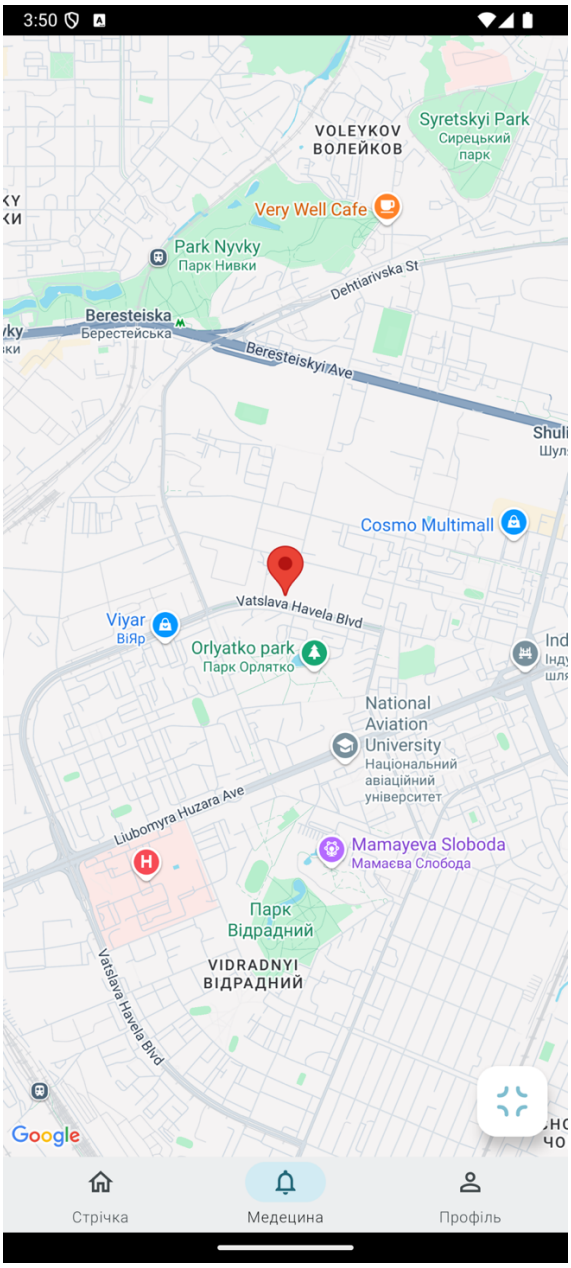


Рисунок 3.9 Список медичних закладів, доступних у додатку для вибору користувачем

3.3 Тестування та налагодження додатку

Тестування та налагодження є невід'ємною частиною процесу розробки мобільного додатку, оскільки вони гарантують його стабільність, надійність і відповідність початковим вимогам. Етап тестування передбачає різні методи перевірки, зокрема функціональні тести.

1. Функціональне тестування

Функціональне тестування перевіряє, чи працюють всі функції додатку згідно із заданими вимогами та очікуваннями користувача. Воно охоплює всі основні функції, включаючи авторизацію, перегляд полісів, а також сповіщення та управління профілем користувача.

Ручне тестування: певні функції потребують додаткової уваги, особливо при першому тестуванні додатку, коли потрібно переконатися, що всі сценарії правильно обробляються та працюють без помилок. Ручне тестування особливо корисне для перевірки коректності роботи інтерфейсу та зручності використання додатку.

2. Налагодження

Налагодження є важливим процесом після тестування, який дозволяє ідентифікувати та виправити знайдені помилки чи проблеми. Це включає перевірку логів, аналіз коду, виправлення багів та ретестування для забезпечення повної функціональності.

Інструменти налагодження: для налагодження додатків Kotlin Multiplatform можна використовувати Android Studio та Xcode, що дозволяють здійснювати відстеження помилок як на Android, так і на iOS. Додаткова інтеграція з інструментами збору логів (наприклад, Firebase Crashlytics) дозволяє відстежувати помилки, які виникають у користувачів під час використання додатку [7].

Ретестування: після виправлення помилок усі модулі проходять повторне тестування, щоб переконатися, що усунення багів не вплинуло на інші частини додатку. Це забезпечує стабільність і надійність всього продукту.

3. Залучення кінцевих користувачів до тестування (бета-тестування)

Бета-тестування передбачає залучення невеликої групи кінцевих користувачів для оцінки роботи додатку. Це дозволяє отримати реальні відгуки щодо роботи функцій, зручності використання та надійності додатку [12].

Збір зворотного зв'язку: користувачі можуть повідомляти про можливі помилки або пропонувати покращення, що дозволяє врахувати реальний досвід використання.

Аналіз відгуків: відгуки від бета-тестерів можуть допомогти зрозуміти, які аспекти потребують доопрацювання або змін, щоб забезпечити кращий користувацький досвід [15].

3.4 Аналіз ефективності використання Kotlin Multiplatform

Kotlin Multiplatform (KMP) є одним з інноваційних рішень для кросплатформеної розробки, яке дозволяє використовувати спільну кодову базу для створення додатків, що працюють на різних платформах, таких як Android, iOS, Web тощо. Аналіз ефективності KMP включає в себе оцінку її продуктивності, економії ресурсів на розробку та підтримку, а також порівняння з іншими кросплатформеними та нативними підходами. Це допомагає зрозуміти, чи є KMP оптимальним вибором для вашого додатку та які її переваги й недоліки в реальних умовах.

1. Продуктивність додатку

Одним з ключових аспектів аналізу є оцінка продуктивності додатків, створених на KMP, зокрема швидкості виконання та споживання ресурсів.

Швидкість виконання: KMP забезпечує хорошу продуктивність для бізнес-логіки, яка реалізована в спільному модулі, оскільки цей код компілюється для кожної цільової платформи окремо. Таким чином, Kotlin Multiplatform дозволяє досягти високої продуктивності для критичних функцій додатку, хоча деякі елементи інтерфейсу, написані нативно, можуть мати певні розбіжності у швидкодії на різних платформах [12].

Споживання ресурсів: оскільки KMP дозволяє уникнути дублювання коду для кожної платформи, це знижує обсяги використовуваного додатком коду, що сприяє оптимізації використання пам'яті. Крім того, продуктивність додатку може бути підвищена за рахунок налаштування пам’яті та ресурсоємних функцій залежно від платформи. [6].

2. Скорочення витрат на розробку

Kotlin Multiplatform дозволяє значно знизити витрати на розробку та підтримку, що робить його ефективним рішенням для створення багатоплатформених додатків. Ось основні аспекти, які впливають на економію ресурсів:

Зменшення часу на розробку: використання єдиної кодової бази для бізнес-логіки та повторюваних компонентів дозволяє значно зменшити час, необхідний для розробки додатку. Завдяки цьому розробники можуть працювати над новими функціями та оновленнями, не відволікаючись на окрему розробку для кожної платформи [13].

Підтримка та оновлення: оновлення бізнес-логіки або фікси помилок можуть бути впроваджені у спільному модулі, що автоматично застосовується для всіх платформ, де використовується цей код. Це забезпечує швидше виправлення помилок і полегшує підтримку додатку в довгостроковій перспективі.

3. Гнучкість у виборі архітектури

Однією з головних переваг KMP є гнучкість у виборі архітектури додатку. Розробники можуть налаштовувати проєкт під конкретні вимоги, вибираючи, які частини коду будуть спільними для всіх платформ, а які — нативними для кожної платформи окремо.

Спільний та платформо-залежний код: KMP дозволяє розділяти код на спільний (common) та платформо-залежний, що дозволяє оптимізувати додаток під конкретні потреби користувачів кожної платформи. Наприклад, бізнес-логіка, обробка даних, робота з мережею можуть бути реалізовані в загальному модулі, тоді як специфічні функції інтерфейсу можуть бути нативними для кожної платформи.

Модульність та повторне використання: KMP підтримує модульний підхід до розробки, що дозволяє легше керувати кодом та повторно використовувати готові компоненти в різних частинах проєкту. Це особливо корисно для великих проєктів, де є можливість відокремлювати функціональні модулі та використовувати їх у кількох додатках одночасно.

4. Порівняння з нативною розробкою та іншими кросплатформеними рішеннями

Порівняння KMP з нативною розробкою та іншими кросплатформеними рішеннями (такими як Flutter або React Native) дозволяє оцінити, наскільки ефективною є ця технологія в конкретних умовах розробки.

Переваги над нативною розробкою: KMP дозволяє об’єднати бізнес-логіку для кількох платформ, тоді як у нативній розробці для кожної платформи необхідно розробляти код окремо. Це знижує витрати на підтримку та оновлення, а також скорочує час виходу продукту на ринок.

Порівняння з іншими кросплатформеними фреймворками: у порівнянні з Flutter чи React Native, KMP забезпечує більшу гнучкість у використанні нативного інтерфейсу кожної платформи, що дозволяє створювати додатки з нативним користувацьким досвідом. Однак у випадках, коли потрібен повністю спільний інтерфейс, KMP може поступатися іншим рішенням у швидкості розробки інтерфейсу [9].

5. Відгуки та досвід розробників

Ключовий фактор ефективності будь-якої технології — це відгуки і досвід розробників, які її використовують. Kotlin Multiplatform активно підтримується JetBrains, а також має велике ком’юніті розробників, яке ділиться власним досвідом і рішеннями.

Спільнота та підтримка: завдяки активному розвитку та підтримці з боку спільноти розробників, KMP стає все більш популярним та отримує оновлення з новими функціями, що робить його перспективною технологією для кросплатформеної розробки.

Виклики та можливості для оптимізації: оскільки KMP ще розвивається, деякі розробники стикаються з обмеженнями щодо підтримки бібліотек або документації. Проте з кожним оновленням ці недоліки зменшуються, а KMP стає доступнішим і зрозумілішим для більшої кількості розробників [6].

6. Висновок щодо ефективності KMP

Аналіз ефективності використання Kotlin Multiplatform показує, що цей підхід до розробки багатоплатформених додатків може значно знизити витрати на розробку та забезпечити високу продуктивність додатків. Основними перевагами KMP є гнучкість у виборі архітектури, можливість створення нативного користувацького досвіду та скорочення часу на розробку. Однак слід також враховувати, що KMP ще розвивається, тому можуть виникати труднощі з підтримкою деяких функцій чи бібліотек. Загалом, KMP є ефективним інструментом для розробки багатоплатформених додатків, особливо коли важливі продуктивність і можливість створення нативного інтерфейсу для кожної платформи [13].

3.4.1 Оцінка продуктивності додатку

Оцінка продуктивності додатку, розробленого на Kotlin Multiplatform (KMP), є важливим аспектом для визначення його ефективності, стабільності та здатності відповідати очікуванням користувачів на різних платформах. Продуктивність аналізується з точки зору швидкості виконання бізнес-логіки, ефективності використання ресурсів пристроїв (таких як пам’ять та процесор), а також стабільності під навантаженням [13].

1. Швидкість виконання бізнес-логіки

Бізнес-логіка, реалізована в спільному коді KMP, має значну перевагу завдяки тому, що компілюється для кожної цільової платформи окремо (наприклад, у байт-код для JVM на Android або в нативний код для iOS). Це забезпечує високу швидкість виконання загальних функцій, зокрема обробки даних, мережевих запитів та взаємодії з базою даних.

Показники швидкодії: для оцінки швидкості виконання бізнес-логіки можна використовувати час відгуку на виклики до серверу, швидкість обробки транзакцій чи інших складних операцій, які критичні для функціональності додатку.

2. Використання ресурсів (пам’яті та процесора)

Однією з ключових характеристик продуктивності додатку є ефективне використання ресурсів пристроїв. Це стосується обсягу пам'яті, який споживає додаток, а також навантаження на процесор, особливо під час виконання складних обчислень або обробки графічних елементів.

Тестування використання пам’яті: KMP дозволяє уникнути дублювання коду на різних платформах, що сприяє економії пам'яті, оскільки велика частина бізнес-логіки не повторюється. Для оцінки можна вимірювати середнє та пікове використання пам'яті в моменти високих навантажень.

Навантаження на процесор: ефективність обробки даних та інших обчислювальних задач можна оцінювати за допомогою аналізу використання процесора під час виконання певних функцій. Це особливо важливо для пристроїв з обмеженими ресурсами.

3. Стабільність під навантаженням

Стабільність додатку під великим навантаженням, зокрема в умовах одночасного доступу багатьох користувачів, є критично важливим показником продуктивності. Навантажувальні тести дозволяють перевірити, як додаток реагує на велику кількість запитів або одночасне виконання різних процесів.

Тестування на багатозадачність: KMP забезпечує стабільну роботу при одночасній обробці запитів у спільному модулі. Для цього використовуються навантажувальні тести, щоб виявити слабкі місця у роботі програми та налаштувати ефективне управління ресурсами.

4. Порівняння продуктивності з нативними рішеннями

Щоб оцінити ефективність Kotlin Multiplatform у порівнянні з нативними рішеннями, важливо проводити паралельні тести продуктивності. Це дозволяє визначити, чи є значна різниця в часі відгуку, навантаженні на процесор та споживанні пам'яті при виконанні однакових операцій.

Еталонні тести: створення серії еталонних тестів для порівняння продуктивності дозволяє отримати об’єктивні дані щодо відмінностей у роботі нативного додатку та додатку на базі KMP. Наприклад, порівняння часу відгуку при виконанні обробки даних або швидкості запуску додатку на Android та iOS[13].

5. Інструменти для моніторингу продуктивності

Для проведення точного аналізу продуктивності додатку слід використовувати спеціалізовані інструменти моніторингу та відстеження ресурсів.

Android Profiler для аналізу продуктивності на Android: дозволяє відстежувати використання процесора, пам'яті та частоти кадрів.

Instruments для iOS: забезпечує контроль ресурсів, вимірювання часу виконання та виявлення вузьких місць у продуктивності.

Firebase Performance Monitoring: забезпечує моніторинг продуктивності додатку на обох платформах, зокрема відстеження часу відгуку, навантаження на процесор та використання пам'яті під час реального використання додатку [7].

6. Висновок щодо оцінки продуктивності

Аналіз продуктивності додатку на основі KMP показує, що цей підхід здатний забезпечити високу ефективність, зокрема для бізнес-логіки та обробки даних. Завдяки компіляції спільного коду для кожної платформи додаток може виконуватися швидко і стабільно. Проте для досягнення максимальних результатів важливо оптимізувати нативний код для інтерфейсу кожної платформи та враховувати специфіку роботи з ресурсами пристроїв.

3.4.2 Порівняння з нативною розробкою та іншими кросплатформеними рішеннями

Порівняння Kotlin Multiplatform (KMP) з нативною розробкою та іншими кросплатформеними рішеннями (такими як Flutter і React Native) дозволяє зрозуміти переваги, недоліки та основні відмінності кожного підходу. Це допомагає вибрати оптимальну технологію для реалізації багатоплатформених додатків залежно від цілей проєкту, ресурсів та вимог до функціональності. [9].

1. Порівняння з нативною розробкою

Нативна розробка передбачає створення окремих додатків для кожної платформи (Android і iOS) з використанням платформо-залежних інструментів і мов програмування: Kotlin або Java для Android і Swift або Objective-C для iOS.

Переваги нативної розробки: нативні додатки мають максимальну продуктивність і доступ до всіх можливостей платформи, що забезпечує високу швидкість виконання, плавність інтерфейсу та відповідність платформеним гайдлайнам. Крім того, нативна розробка дає повний контроль над дизайном інтерфейсу та доступ до найновіших API [13].

Недоліки нативної розробки: основним недоліком є висока вартість та тривалість розробки, оскільки кожна платформа вимагає окремої команди та дублювання бізнес-логіки. Це збільшує витрати на підтримку та оновлення додатку, оскільки будь-які зміни потрібно вносити у кожну кодову базу окремо.

Порівняння з KMP: на відміну від нативної розробки, KMP дозволяє використовувати спільну бізнес-логіку для обох платформ, що скорочує час розробки і знижує витрати. При цьому KMP дозволяє реалізовувати нативні інтерфейси для кожної платформи, що зберігає нативний досвід користувача, а спільний код значно спрощує підтримку додатку.

2. Порівняння з Flutter

Flutter — це фреймворк для кросплатформеної розробки від Google, який використовує мову програмування Dart. Він дозволяє створювати додатки з єдиним кодом для обох платформ, що робить розробку швидшою і більш економічною.

Переваги Flutter: основною перевагою є можливість створення єдиного коду, який працює однаково на Android та iOS. Flutter використовує власний механізм рендерингу для відображення інтерфейсу, що забезпечує високий рівень гнучкості у дизайні та швидке тестування UI. Flutter також підтримує велику кількість бібліотек, які спрощують розробку [9].

Недоліки Flutter: оскільки Flutter рендерить інтерфейс, використовуючи власний механізм, додаток може не повністю відповідати нативному стилю кожної платформи. Це може знизити користувацький досвід на платформах із відмінними дизайн-гайдлайнами, особливо для iOS. Крім того, розміри додатків на Flutter часто більші, а продуктивність може бути обмеженою для ресурсомістких завдань [9].

Порівняння з KMP: у той час як Flutter дозволяє створювати єдиний інтерфейс для обох платформ, KMP зосереджується на поділі бізнес-логіки,

залишаючи інтерфейс нативним для кожної платформи. Це забезпечує кращий користувацький досвід і оптимізує додаток для кожної ОС, особливо для складних додатків, які потребують специфічної функціональності [6].

3. Порівняння з React Native

React Native — це популярний фреймворк для кросплатформеної розробки від Meta (Facebook), що використовує JavaScript і дозволяє створювати спільну кодову базу для Android та iOS. React Native забезпечує часткову підтримку нативного інтерфейсу через JavaScript-компоненти.

Переваги React Native: основна перевага полягає у використанні JavaScript, що полегшує входження у розробку. React Native дозволяє створювати компоненти, які легко адаптуються до кожної платформи, і підтримує нативний код, якщо необхідно інтегрувати специфічні для платформи API.

Недоліки React Native: продуктивність React Native може бути нижчою порівняно з нативною розробкою, особливо для складних або графічно інтенсивних додатків. Крім того, користувачам потрібно інтегрувати окремі модулі для більшої продуктивності, що може призводити до ускладнень в розробці.

Порівняння з KMP: на відміну від React Native, який дозволяє створювати гібридний інтерфейс з елементами нативного коду, KMP підтримує повністю нативні інтерфейси. Це дозволяє створювати додатки з високою продуктивністю та нативним користувацьким досвідом, а спільний код зосереджується на бізнес-логіці.

4. Основні переваги та недоліки KMP

Завдяки використанню спільної кодової бази, KMP забезпечує значне скорочення часу та ресурсів на розробку, особливо для проєктів з великою кількістю бізнес-логіки. Основні переваги та недоліки KMP у порівнянні з іншими підходами:

Переваги KMP:

Спільна бізнес-логіка для кількох платформ, що знижує витрати на підтримку та оновлення.

Підтримка нативного інтерфейсу для кожної платформи, що дозволяє зберігати високий користувацький досвід.

Гнучкість у виборі архітектури та можливість легко інтегрувати KMP у наявні проєкти.

Недоліки KMP:

Обмежена кількість бібліотек і документації порівняно з Flutter або React Native, особливо для новачків [9].

5. Compose Multiplatform (CMP) — це інструмент для розробки кросплатформених інтерфейсів користувача, заснований на Jetpack Compose. CMP дозволяє створювати спільний код для UI, який може працювати одразу на кількох платформах, таких як Android, iOS, Desktop і Web [10]

Переваги CMP:

Уніфікований підхід до UI. CMP дозволяє розробникам писати один спільний код для користувацького інтерфейсу, що значно зменшує час і витрати на створення UI для різних платформ.

Гнучкість кастомізації. CMP забезпечує можливість інтеграції платформо-залежних компонентів у загальний інтерфейс, зберігаючи нативний досвід користувача там, де це необхідно.

Інтеграція з Kotlin Multiplatform. CMP гармонійно доповнює KMP, дозволяючи створювати єдине рішення для бізнес-логіки та інтерфейсу.

Простота адаптації дизайну. Використання декларативного підходу до створення UI полегшує створення адаптивних інтерфейсів для різних типів пристроїв і розмірів екранів.

Відкритість до спільнот. CMP активно розвивається, що сприяє розширенню його можливостей і появі нових інструментів для підтримки додатків.

Недоліки CMP:

Обмеження в продуктивності. Для деяких складних або графічно насичених інтерфейсів продуктивність CMP може поступатися нативним рішенням.

Новизна технології. CMP ще перебуває на стадії активного розвитку, через що можливі обмеження у стабільності та підтримці деяких функцій.

Розмір додатків. Використання спільного рендерера для UI може збільшити розмір додатку порівняно з нативними інтерфейсами.

Порівняння з Kotlin Multiplatform (KMP):

CMP є логічним доповненням KMP, зосереджуючись на спрощенні розробки інтерфейсів користувача. На відміну від KMP, який орієнтований на спільну бізнес-логіку, CMP дозволяє створювати єдиний код для UI, який буде виглядати однаково на різних платформах. CMP підходить для проєктів, де важливі зручність і швидкість створення інтерфейсів, але не потрібна максимальна оптимізація під нативні дизайн-гайдлайни.

Порівняння з іншими технологіями:

Flutter: Подібно до CMP, Flutter дозволяє створювати єдиний UI для кількох платформ. Однак, Flutter використовує власний механізм рендерингу, що може не повністю відповідати платформо-залежним гайдлайнам, тоді як CMP дозволяє інтегрувати нативні компоненти, забезпечуючи більш природний користувацький досвід.

React Native: React Native дозволяє створювати інтерфейси з використанням JavaScript і гібридного підходу до UI, але CMP забезпечує повністю декларативний підхід у поєднанні зі спільним кодом на Kotlin, що спрощує роботу в проєктах із Kotlin Multiplatform.

6. Висновок щодо порівняння

Аналіз показує, що вибір між нативною розробкою, KMP та іншими кросплатформеними фреймворками залежить від цілей проєкту. KMP є оптимальним вибором, коли потрібно забезпечити спільну бізнес-логіку, зберігаючи при цьому нативний користувацький досвід для кожної платформи. Flutter підходить для швидкої розробки та візуально складних додатків, які не потребують строгого дотримання нативних стандартів. React Native може бути корисним для створення простих додатків із частковим доступом до нативного коду [9].

Kotlin Multiplatform забезпечує збалансований підхід, який дозволяє знизити витрати та оптимізувати розробку, не жертвуючи якістю та продуктивністю.

3.5 Оптимізація додатку та покращення користувацького досвіду

Одним із ключових аспектів оптимізації мобільного додатку для медичного страхування є впровадження ефективного механізму кешування. Кешування дозволяє зменшити час завантаження додатку, мінімізувати кількість запитів до серверу та забезпечити швидкий доступ до даних навіть за умови слабкого або відсутнього інтернет-з’єднання.

Основні аспекти кешування:

Локальне кешування даних. Інформація про страхові поліси, медичні заклади та користувацькі дані може зберігатися локально на пристрої. Це дозволяє швидко відображати інформацію без необхідності очікування відповіді від сервера. Інструменти, такі як Realm, SharedPreferences (для Android) або Keychain (для iOS), забезпечують ефективне зберігання кешованих даних.

Кешування мережевих запитів. Використання бібліотеки, такої як Ktor, дозволяє кешувати результати API-запитів, що значно зменшує навантаження на сервер і покращує продуктивність. Наприклад, інформація про статус заяви користувача може бути оновлена лише при відповідному запиті або після закінчення терміну дії кешу [8].

Стратегія інтелектуального оновлення. Важливо реалізувати механізм автоматичного оновлення кешованих даних після певного періоду або коли додаток підключається до мережі. Це забезпечує доступ до актуальної інформації без додаткових дій з боку користувача.

Вплив на користувацький досвід:

Швидке завантаження інтерфейсу: завдяки кешуванню додаток відображає дані миттєво, навіть при слабкому інтернет-з’єднанні.

Зменшення споживання мобільного трафіку: оптимізація запитів до сервера дозволяє зекономити ресурси користувача, що є особливо важливим для клієнтів із лімітованим мобільним інтернетом.

Підвищення стабільності додатку: навіть у випадках тимчасових проблем із мережею користувачі матимуть доступ до останньої кешованої версії даних, що знижує ризик негативного користувацького досвіду.

Інструменти для впровадження:

Kotlin Multiplatform: забезпечує єдиний підхід до реалізації кешування для кількох платформ.

Firebase Firestore: може використовуватися для синхронізації кешованих даних між локальним сховищем і хмарою, забезпечуючи актуальність інформації на різних пристроях користувача.

Переваги кешування:

Зменшення часу завантаження додатку.

Покращення стабільності роботи у разі нестабільного інтернету.

Зменшення навантаження на бекенд-сервіси [15].

Кешування є невід'ємною складовою сучасних мобільних додатків, яке не тільки підвищує швидкість роботи, але й забезпечує комфортний та безперервний користувацький досвід. У додатках для медичного страхування це особливо важливо, адже доступність даних про поліси чи статус заяв має бути максимально швидкою та надійною.

3.6 ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було детально проаналізовано ключові аспекти розробки мобільного додатку для медичного страхування, включаючи особливості впровадження Kotlin Multiplatform (KMP), порівняння з іншими технологіями та оптимізацію додатку для покращення користувацького досвіду.

Розділ почався з аналізу функціональних можливостей додатку, зокрема управління страховими полісами, відображення профілю користувача, інтеграції з медичними закладами та відправки сповіщень. Було підкреслено, що ефективна реалізація цих функцій залежить від вибору оптимальних технологій та інструментів.

Важливу роль у розробці відіграє Kotlin Multiplatform, який дозволяє розділяти бізнес-логіку між платформами, зберігаючи при цьому нативний досвід користувача для Android та iOS. Порівняння KMP з нативною розробкою та іншими кросплатформеними рішеннями, такими як Flutter і React Native, підтвердило, що KMP є ефективним рішенням для багатоплатформних проєктів, які потребують оптимального балансу між швидкістю розробки, якістю інтерфейсу та зручністю підтримки.

Окрему увагу було приділено Compose Multiplatform (CMP), який доповнює KMP, забезпечуючи створення спільного коду для користувацького інтерфейсу. Це дозволяє зменшити витрати на розробку UI та забезпечити узгодженість дизайну на різних платформах. CMP є цінним інструментом для покращення продуктивності розробки та скорочення часу на створення інтерфейсу [10].

Розділ також включав детальний огляд підходів до оптимізації додатку. Основними методами було обрано кешування даних, оптимізацію мережевих запитів, забезпечення стабільності роботи додатку у випадках слабкого інтернет-з’єднання та використання сучасних інструментів для збереження конфіденційності даних користувачів.

Таким чином, у цьому розділі були визначені ефективні стратегії для забезпечення високої продуктивності, безпеки та покращення користувацького досвіду в мобільному додатку для медичного страхування. Використання Kotlin Multiplatform і Compose Multiplatform у поєднанні з сучасними інструментами для оптимізації та інтеграції забезпечує створення масштабованого, гнучкого і зручного для користувачів додатку, що відповідає вимогам сучасного ринку та сфери медичного страхування.

ВИСНОВКИ

У ході виконання магістерської роботи було детально досліджено ефективність використання Kotlin Multiplatform (KMP) для розробки багатоплатформних мобільних додатків на прикладі медичного страхування. Робота складалася з трьох основних етапів: теоретичного аналізу, огляду технологій та інструментів, а також реалізації оптимізаційних рішень і порівняння з іншими підходами.

У першому розділі було визначено теоретичні основи розробки мобільних додатків, включаючи порівняння нативних, кросплатформених і гібридних рішень. Особливу увагу приділено Kotlin Multiplatform як технології, що поєднує переваги кросплатформеного підходу (спільна бізнес-логіка) із збереженням нативного досвіду користувача. Також було розглянуто Compose Multiplatform (CMP), який забезпечує спільну розробку інтерфейсів користувача для різних платформ, зменшуючи витрати часу та ресурсів. Аналіз показав, що використання KMP дозволяє ефективно створювати багатоплатформні рішення з високою продуктивністю та гнучкістю.

У другому розділі було описано технології та інструменти, які використовувалися для розробки додатку. Зокрема:

Kotlin Multiplatform для спільної бізнес-логіки;

Compose Multiplatform для спільного інтерфейсу користувача;

Android Studio та Xcode для нативної розробки та тестування;

Інтеграція з бекенд-сервісами через Ktor і Firebase Cloud Messaging (FCM) для роботи з API, обробки даних та сповіщень;

Локальні бази даних, такі як Realm, SharedPreferences (для Android) або Keychain (для iOS), для збереження конфіденційних даних користувачів.

Розгляд сучасних інструментів підтвердив, що їхнє використання сприяє створенню продуктивних і безпечних додатків, які відповідають потребам бізнесу та користувачів.

У третьому розділі було проаналізовано особливості оптимізації додатку для покращення користувацького досвіду. Основну увагу приділено:

Кешуванню даних для забезпечення швидкого доступу до інформації без зайвих запитів до сервера;

Зменшенню навантаження на мережу через оптимізацію запитів;

Використанню сучасних технологій шифрування для захисту медичних даних;

Впровадженню адаптивних інтерфейсів через CMP, що забезпечують узгоджений дизайн на різних платформах [14].

Було також проведено порівняння Kotlin Multiplatform із нативною розробкою та іншими кросплатформеними рішеннями (Flutter і React Native). Це дозволило висвітлити переваги KMP, такі як спільна кодова база для бізнес-логіки, збереження нативного інтерфейсу користувача та зниження витрат на розробку.

Основні висновки:

Kotlin Multiplatform є перспективною технологією для багатоплатформної розробки, яка забезпечує високу продуктивність, гнучкість і зменшення витрат на створення та підтримку додатків.

Використання Compose Multiplatform дозволяє оптимізувати розробку інтерфейсів, забезпечуючи узгодженість дизайну та зменшення часу на розробку.

Інтеграція з сучасними бекенд-сервісами, такими як Ktor і Firebase, значно підвищує функціональність додатку, забезпечуючи обробку даних, сповіщення та безпеку.

Оптимізаційні рішення, такі як кешування та адаптивні інтерфейси, сприяють покращенню користувацького досвіду та стабільності роботи додатку.

Таким чином, результати магістерської роботи показали, що використання Kotlin Multiplatform і пов’язаних інструментів є ефективним підходом до розробки сучасних мобільних додатків. Цей підхід дозволяє створювати багатоплатформні рішення, які відповідають вимогам бізнесу, забезпечують високий рівень продуктивності та задовольняють очікування користувачів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kotlin Multiplatform Mobile: A New Approach to Cross-Platform Development», JetBrains, [https://kotlinlang.org/docs/multiplatform-mobile.html].

2. Compose Multiplatform Overview», JetBrains, [https://kotlinlang.org/docs/compose-multiplatform.html].

3. https://www.jetbrains.com/kotlin-multiplatform/.

4. Ktor: Building Connected Systems», JetBrains, [https://ktor.io/].

5. Petrov I., Smith A. Cross-Platform Mobile Development with Kotlin Multiplatform and Compose Multiplatform. – Berlin: Springer, 2022. – 320 p.

6. Johnson T. Enhancing User Experience in Medical Apps through Optimization Techniques. // International Conference on Software Engineering, 2023. – C.201-210.

7. Firebase Cloud Messaging for Mobile App Development», Google Developers, [https://firebase.google.com/docs/cloud-messaging].

8. Room vs Realm: Choosing the Best Local Database for Your App», Dev.to, [https://dev.to/].

9. A Performance Comparison of Flutter, React Native, and Kotlin Multiplatform», International Journal of Software Engineering and Knowledge Engineering, Volume 33, Issue 5, 2023. – C.457-468.

10. Das A. «How to Develop Multiplatform apps using Compose Multiplatform for iOS, Android, Desktop and Web», Medium, [https://medium.com/developers-inc/how-to-develop-multiplatform-apps-using-compose-multiplatform-for-ios-android-desktop-and-web-7f4b3e7180cf].

11. Nassar A. «I look at you, and… I'm home. One click compose multiplatform template», Medium, [https://medium.com/@ranger163/i-look-at-you-and-im-home-one-click-compose-multiplatform-template-cbb6c7712675].

12. Witaszak A. «Kotlin Multiplatform Android, iOS and Desktop Apps with shared UI: React Native killer?», Medium, [https://medium.com/@adrianwitaszak/kotlin-multiplatform-android-ios-and-desktop-apps-with-shared-ui-react-native-killer-9b689b2a91fd].

13. «Exploring Kotlin Multiplatform Mobile (KMM): Bridging the Gap between Android and iOS Development», Medium, [https://medium.com/@abdulbasitoduk/exploring-kotlin-multiplatform-mobile-kmm-bridging-the-gap-between-android-and-ios-development-c365ebef1ecc].
14. Mastering Networking with Ktor in Kotlin Multiplatform and Compose Multiplatform Apps [Електронний ресурс] – Режим доступу: https://academy.droidcon.com/course/mastering-networking-with-ktor-in-kotlin-multiplatform-kmp-and-compose-multiplatform-apps. –
15. Compose Multiplatform: Getting Started & Initializing Firebase [Електронний ресурс] – Режим доступу: https://johnoreilly.dev/posts/vertex-ai-kmp/.
16. Код додатку: https://github.com/Artemius-dev/Clarity

ДОДАТОК А

Програмний код

LoginRepository.kt

```
package org.artemzhuravlov.clarity.data.repository.login

import dev.gitlive.firebase.Firebase
import dev.gitlive.firebase.auth.FirebaseAuth
import dev.gitlive.firebase.auth.auth
import org.artemzhuravlov.clarity.data.model.user.AuthenticatedUser
import
org.artemzhuravlov.clarity.data.model.user.toFirebaseAuthenticatedUs
er

class LoginRepository(private val auth: FirebaseAuth) :
ILoginRepository {
    override suspend fun login(
        email: String,
        password: String,
        onSuccess: ((AuthenticatedUser?) -> Unit)?,
        onError: ((Exception) -> Unit)?
    ) {
        try {
            val signInResult =
auth.signInWithEmailAndPassword(email, password)
            val firebaseUser = signInResult.user
onSuccess?.invoke(firebaseUser?.toFirebaseAuthenticatedUser())
        } catch (signInException: Exception) {
            onError?.invoke(signInException)
        }
    }

    override suspend fun logout(
        onSuccess: ((AuthenticatedUser?) -> Unit)?,
        onError: ((Exception?) -> Unit)?
    ) {
        try {
            auth.signOut()
            val firebaseUser = auth.currentUser

onSuccess?.invoke(firebaseUser?.toFirebaseAuthenticatedUser())
        } catch (signOutException: Exception) {
            onError?.invoke(signOutException)
        }
    }
```

```
    override suspend fun checkIsUserLoggedIn(isLoggedIn: (() ->
Unit)?) {
        val auth = Firebase.auth
        if (auth.currentUser != null) {
            isLoggedIn?.invoke()
        }
    }
}
```

ILoginRepository.kt

```
package org.artemzhuravlov.clarity.data.repository.login

import org.artemzhuravlov.clarity.data.model.user.AuthenticatedUser

interface ILoginRepository {
    suspend fun login(
        email: String,
        password: String,
        onSuccess: ((AuthenticatedUser?) -> Unit)? = null,
        onError: ((Exception) -> Unit)? = null
    )

    suspend fun logout(
        onSuccess: ((AuthenticatedUser?) -> Unit)? = null,
        onError: ((Exception?) -> Unit)? = null
    )

    suspend fun checkIsUserLoggedIn(isLoggedIn: (() -> Unit)? =
null)
}
```

PoliciesScreen.kt

```
package org.artemzhuravlov.clarity.ui.screens.policies

import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.pager.HorizontalPager
import androidx.compose.foundation.pager.rememberPagerState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.AttachMoney
import androidx.compose.material.icons.filled.Call
import androidx.compose.material.icons.filled.Description
import androidx.compose.material.icons.filled.Warning
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
```

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.unit.dp
import com.arkivanov.decompose.extensions.compose.subscribeAsState
import org.artemzhuravlov.clarity.data.model.policies.InsuranceCard
import
org.artemzhuravlov.clarity.navigation.policies.IPoliciesScreenCompon
ent
import
org.artemzhuravlov.clarity.navigation.policies.PoliciesScreenEvent
import org.artemzhuravlov.clarity.resources.Res
import org.artemzhuravlov.clarity.resources.contact_us
import org.artemzhuravlov.clarity.resources.financial_information
import
org.artemzhuravlov.clarity.resources.financial_information_url
import org.artemzhuravlov.clarity.resources.insurance_event_actions
import org.artemzhuravlov.clarity.resources.insurance_event_url
import org.artemzhuravlov.clarity.resources.medical_id
import org.artemzhuravlov.clarity.resources.patient
import org.artemzhuravlov.clarity.resources.policies_title
import org.artemzhuravlov.clarity.resources.validity_period
import org.artemzhuravlov.clarity.resources.view_documents
import org.artemzhuravlov.clarity.ui.composables.PagerIndicator
import org.artemzhuravlov.clarity.ui.utils.openExternalWebPage
import org.artemzhuravlov.clarity.ui.utils.openPdf
import org.jetbrains.compose.resources.stringResource

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun PoliciesScreen(component: IPoliciesScreenComponent) {
    val uiState by component.uiState.subscribeAsState()

    var isOpenInsuranceEvent by remember {
        mutableStateOf(false)
    }

    var isOpenFinancialInformation by remember {
        mutableStateOf(false)
    }

    if (isOpenInsuranceEvent) {

openExternalWebPage(stringResource(Res.string.insurance_event_url))
        isOpenInsuranceEvent = false
    }

    if (isOpenFinancialInformation) {

openExternalWebPage(stringResource(Res.string.financial_information_
url))
        isOpenFinancialInformation = false
```

```
    }
    if (uiState.documentUri != null) {
        openPdf(component.uiState.value.documentUri!!)

component.handleEvent(PoliciesScreenEvent.DismissDocumentUri)
    }

    if (uiState.error != null && uiState.error!!.isNotEmpty()) {
        androidx.compose.material.AlertDialog(
            onDismissRequest = {

component.handleEvent(PoliciesScreenEvent.DismissErrorDialog)
            },
            text = {
                Text(text = uiState.error!!)
            },
            buttons = {
                Button(onClick = {

component.handleEvent(PoliciesScreenEvent.DismissErrorDialog)
                }) {
                    Row(
                        modifier = Modifier.fillMaxWidth(),
                        horizontalArrangement = Arrangement.End
                    ) {
                        Text(text = "OK")
                    }
                }
            }
        )
    }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(text =
stringResource(Res.string.policies_title)) },
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor =
MaterialTheme.colorScheme.primary,
                    titleContentColor =
MaterialTheme.colorScheme.onPrimary
                )
            )
        }
    ) { padding ->
        if (uiState.isLoading) {
            Box(modifier = Modifier.fillMaxSize(), contentAlignment
= Alignment.Center) {
                CircularProgressIndicator()
            }
        } else {
            Column(
```

```
                modifier = Modifier
                    .fillMaxSize()
                    .padding(padding)
            ) {
                val policiesPagerState =
rememberPagerState(pageCount = {
                    component.uiState.value.insuranceCards.size
                })

                HorizontalPager(
                    state = policiesPagerState,
                    beyondViewportPageCount = 2,
                    pageSpacing = (-24).dp,
                    contentPadding = PaddingValues(horizontal =
14.dp),
                    modifier = Modifier
                        .fillMaxWidth()
                        .height(200.dp)
                ) { page ->
                    val card = uiState.insuranceCards[page]

                    Box(
                        modifier = Modifier
                            .width(340.dp)
                            .height(165.dp)
                            .clip(RoundedCornerShape(16.dp))
                            .background(Color(card.backgroundColor))
                    ) {
                        PolicyCard(card)
                    }
                }
                Row(
                    modifier =
Modifier.fillMaxWidth().padding(horizontal = 16.dp),
                    horizontalArrangement = Arrangement.Center
                ) {
                    PagerIndicator(pagerState = policiesPagerState)
                }
                Spacer(modifier = Modifier.height(16.dp))
                Column(modifier = Modifier.padding(horizontal =
16.dp)) {
                    ActionItem(
                        icon = Icons.Default.Call,
                        label =
stringResource(Res.string.contact_us),
                        onClick = {

component.handleEvent(PoliciesScreenEvent.NavigateToContacts)
                        }
                    )
                    ActionItem(
                        icon = Icons.Default.Warning,
```

```
                            label =
stringResource(Res.string.insurance_event_actions),
                            onClick = {
                                isOpenInsuranceEvent = true
                            }
                        )
                        ActionItem(
                            icon = Icons.Default.AttachMoney,
                            label =
stringResource(Res.string.financial_information),
                            onClick = {
                                isOpenFinancialInformation = true
                            }
                        )
                        ActionItem(
                            icon = Icons.Default.Description,
                            label =
stringResource(Res.string.view_documents),
                            onClick = {

component.handleEvent(PoliciesScreenEvent.GetDocumentForPolicy(uiSta
te.insuranceCards[policiesPagerState.currentPage].policyNumber))
                            }
                        )
                    }
                }
            }
        }
    }

    @Composable
    private fun PolicyCard(card: InsuranceCard) {
        Column(modifier = Modifier.padding(8.dp)) {
            Text(
                text = card.program,
                style = MaterialTheme.typography.titleLarge,
                color = MaterialTheme.colorScheme.onPrimary
            )
            Spacer(modifier = Modifier.height(8.dp))
            Text(
                text = stringResource(Res.string.patient,
card.patientName),
                style = MaterialTheme.typography.bodyLarge,
                color = MaterialTheme.colorScheme.onPrimary
            )
            Text(
                text = stringResource(Res.string.medical_id,
card.medicalId),
                style = MaterialTheme.typography.bodyLarge,
                color = MaterialTheme.colorScheme.onPrimary
            )
            Text(
```

```
            text = stringResource(Res.string.validity_period,
card.validityPeriod),
            style = MaterialTheme.typography.bodyLarge,
            color = MaterialTheme.colorScheme.onPrimary
        )
    }
}

@Composable
private fun ActionItem(icon: ImageVector, label: String, onClick: ()
-> Unit) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .clip(RoundedCornerShape(8.dp))
            .clickable(onClick = onClick),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Icon(
            imageVector = icon,
            contentDescription = null,
            tint = MaterialTheme.colorScheme.primary
        )
        Spacer(modifier = Modifier.width(16.dp))
        Text(text = label, style =
MaterialTheme.typography.bodyLarge)
    }
}
```

MapComposable.kt

```
package org.artemzhuravlov.clarity.ui.screens.maps.fullscreen_map

import androidx.compose.runtime.Composable
import org.artemzhuravlov.clarity.data.model.place.LatLng

@Composable
expect fun MapComposable(latLng: LatLng)
```

LatLng.kt

```
package org.artemzhuravlov.clarity.data.model.place

import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class LatLng(
    @SerialName("lat")
    val latitude: String = "",
    @SerialName("lng")
    val longitude: String = ""
)
```

MapComposable.android.kt

```
package org.artemzhuravlov.clarity.ui.screens.maps.fullscreen_map

import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import com.google.android.gms.maps.model.CameraPosition
import com.google.android.gms.maps.model.LatLng
import com.google.maps.android.compose.GoogleMap
import com.google.maps.android.compose.MapUiSettings
import com.google.maps.android.compose.Marker
import com.google.maps.android.compose.rememberCameraPositionState
import com.google.maps.android.compose.rememberMarkerState
import org.artemzhuravlov.clarity.data.model.place.LatLng as
LatLngOfPlace

@Composable
actual fun MapComposable(latLng: LatLngOfPlace) {
    Box(
        modifier = Modifier.fillMaxSize(),
    ) {
        val coordinates = LatLng(latLng.latitude.toDouble(),
latLng.longitude.toDouble())
        val markerState = rememberMarkerState(position =
coordinates)
        val cameraPositionState = rememberCameraPositionState {
            position = CameraPosition.fromLatLngZoom(coordinates,
14f)
        }
        GoogleMap(
            modifier = Modifier.fillMaxSize(),
            cameraPositionState = cameraPositionState,
            uiSettings = MapUiSettings(zoomControlsEnabled = false)
        ) {
            Marker(
                state = markerState,
            )
        }
    }
}
```

MapComposable.ios.kt

```
package org.artemzhuravlov.clarity.ui.screens.maps.fullscreen_map

import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.viewinterop.UIKitViewController
import org.artemzhuravlov.clarity.data.places.LatLng
import org.artemzhuravlov.clarity.mapViewController
```

```
@Composable
actual fun MapComposable(latLng: LatLng) {
    UIKitViewController(
        factory = { mapViewController(latLng) },
        modifier = Modifier.fillMaxSize(),
    )
}
```

GoogleMap.swift

```
import SwiftUI
import GoogleMaps

struct GoogleMapView: UIViewRepresentable {
    var latLng: LatLng

    func makeUIView(context: Context) -> GMSMapView {
        let options = GMSMapViewOptions()
        options.camera = GMSCameraPosition.camera(
            withLatitude: latLng.latitude,
            longitude: latLng.longitude,
            zoom: 14.0
        )

        let mapView = GMSMapView(options: options)
        mapView.isZoomEnabled = false
        mapView.showsZoomControls = false

        let marker = GMSMarker()
        marker.position = CLLocationCoordinate2D(latitude:
latLng.latitude, longitude: latLng.longitude)
        marker.map = mapView

        return mapView
    }

    func updateUIView(_ uiView: GMSMapView, context: Context) {
        // Update the map's camera position if the LatLng changes
        let newPosition = GMSCameraPosition.camera(
            withLatitude: latLng.latitude,
            longitude: latLng.longitude,
            zoom: uiView.camera.zoom
        )
        uiView.animate(to: newPosition)

        // Update the marker's position
        if let marker = uiView.selectedMarker {
            marker.position = CLLocationCoordinate2D(latitude:
latLng.latitude, longitude: latLng.longitude)
        }
    }
}
```

ДОДАТОК Б

Скріншоти інтерфейсу додатку

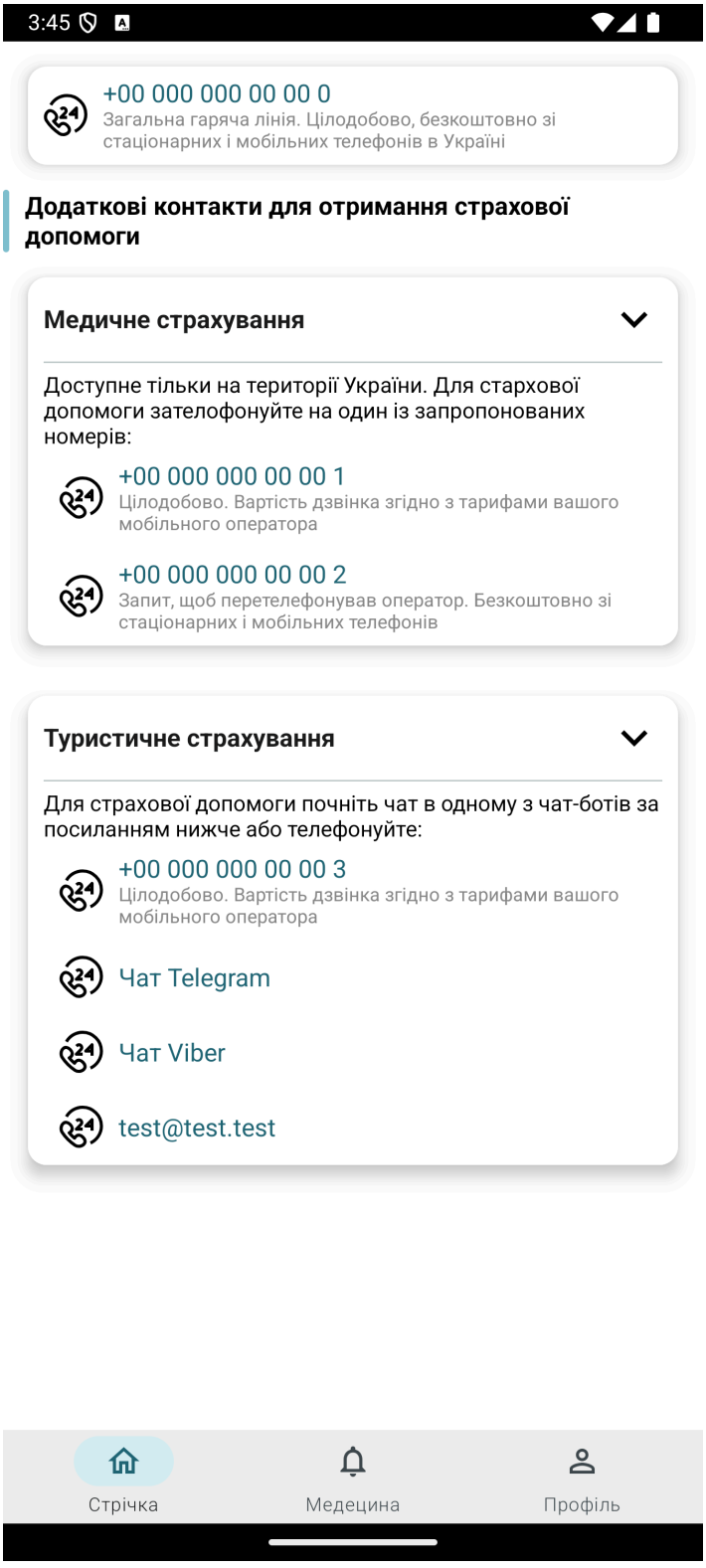


Рисунок 3.1 Контакти для отримання страхової допомоги

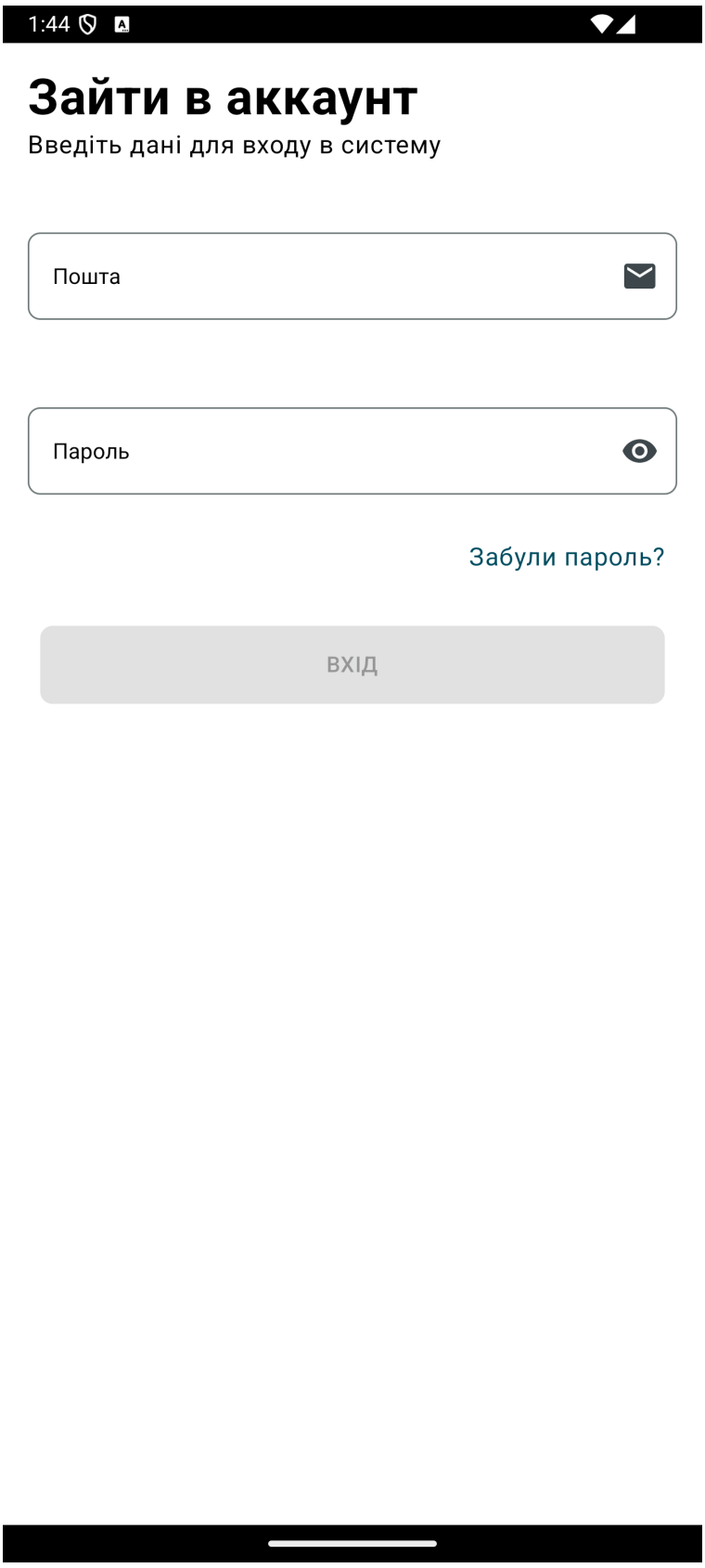


Рисунок 3.2 Успішний вхід у додаток через форму авторизації.

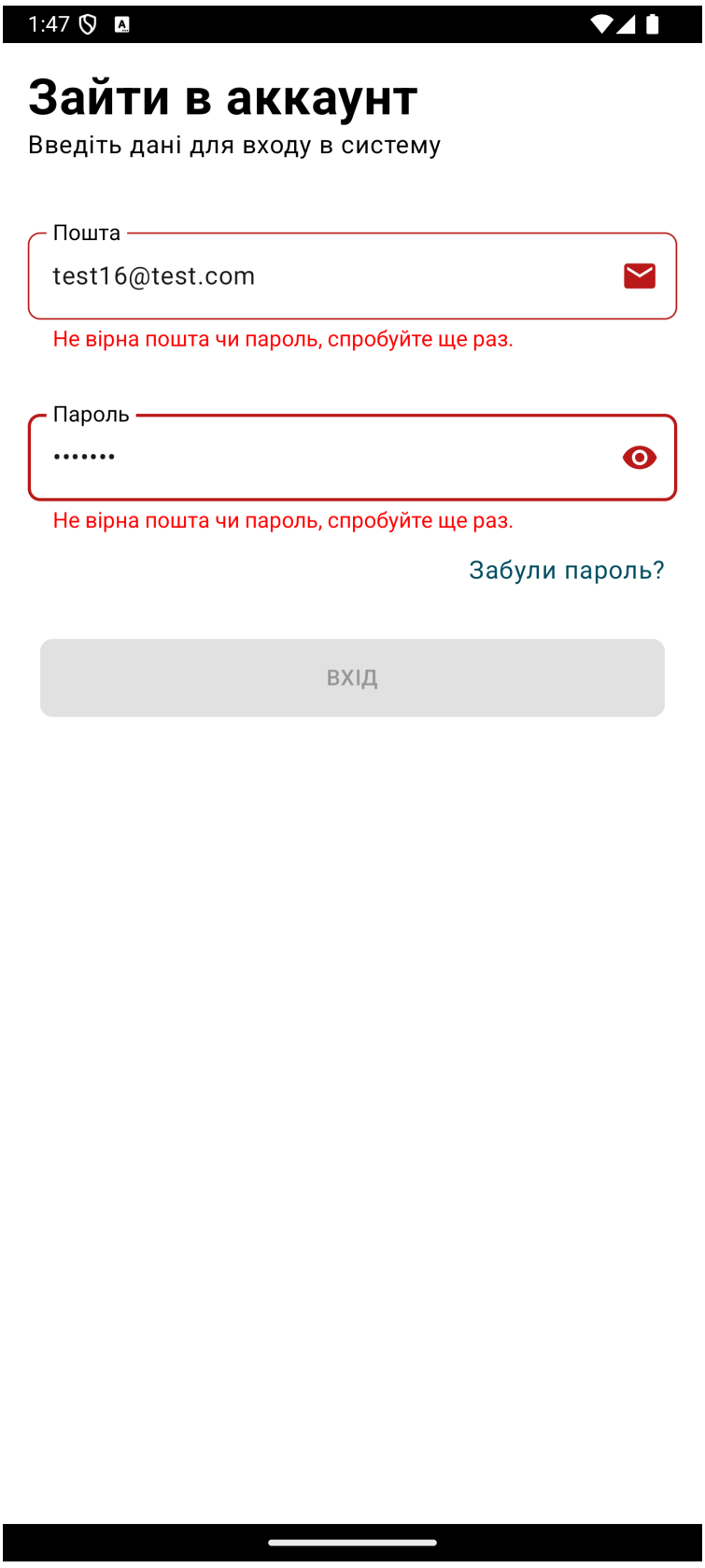


Рисунок 3.3 Повідомлення про помилку у разі невірного вводу даних

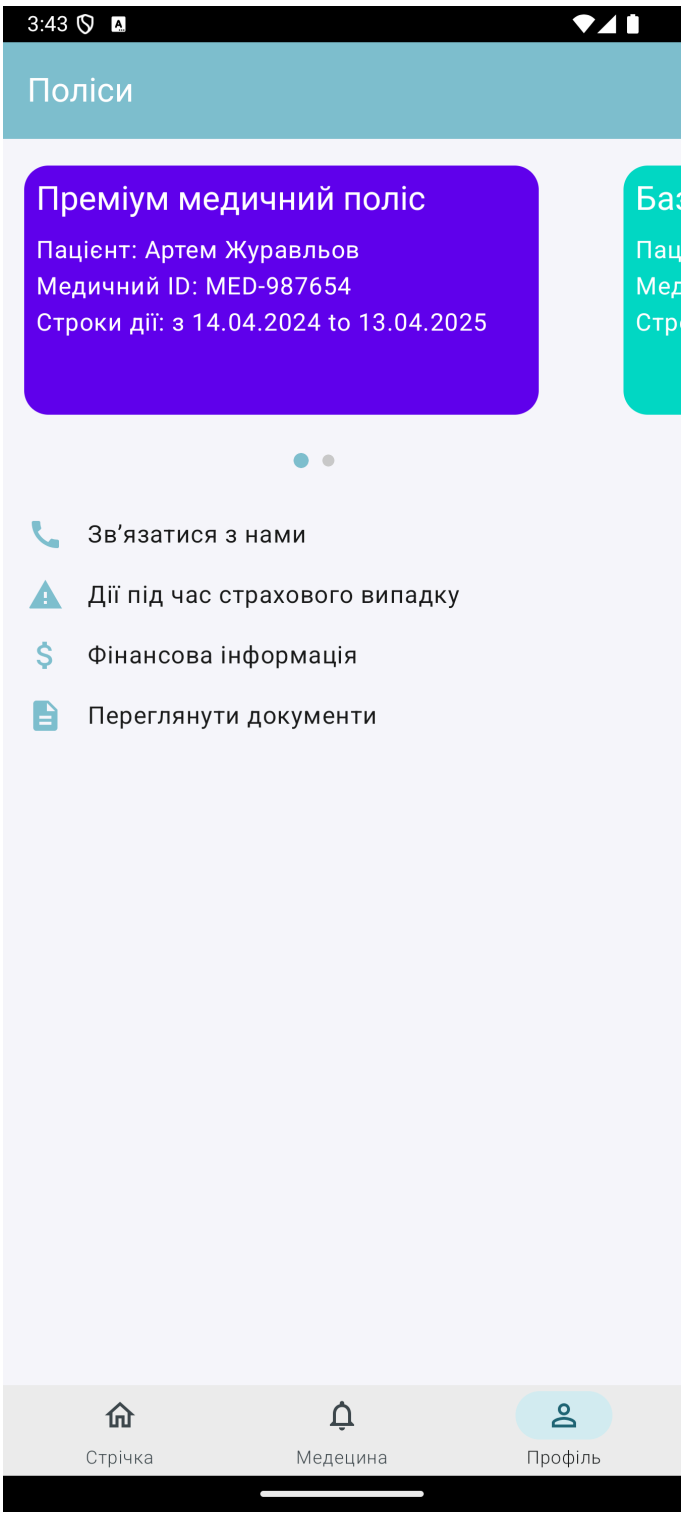


Рисунок 3.4 Відображення преміум медичного полісу з детальною інформацією про пацієнта та терміни дії полісу

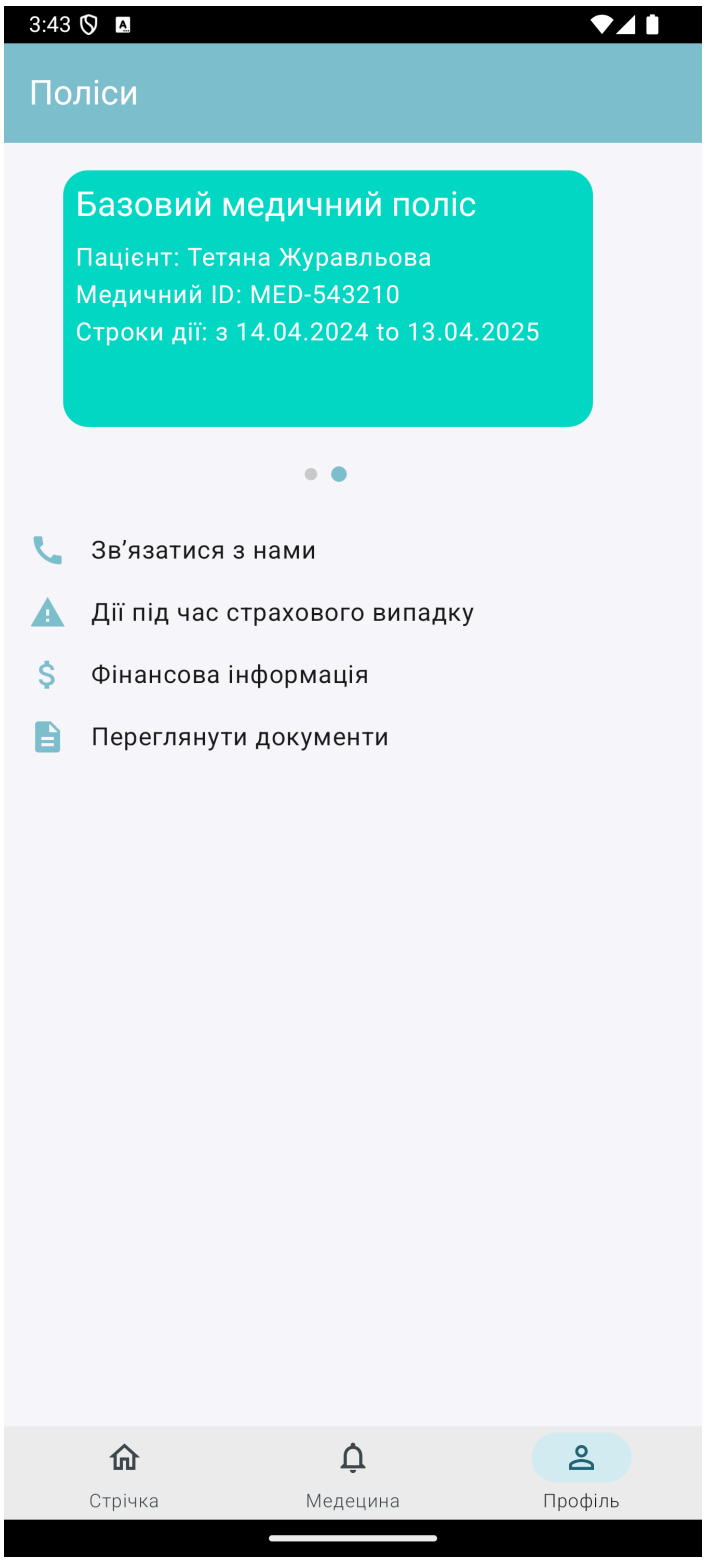


Рисунок 3.5 Відображення базового медичного полісу з інформацією про пацієнта та терміни дії полісу.

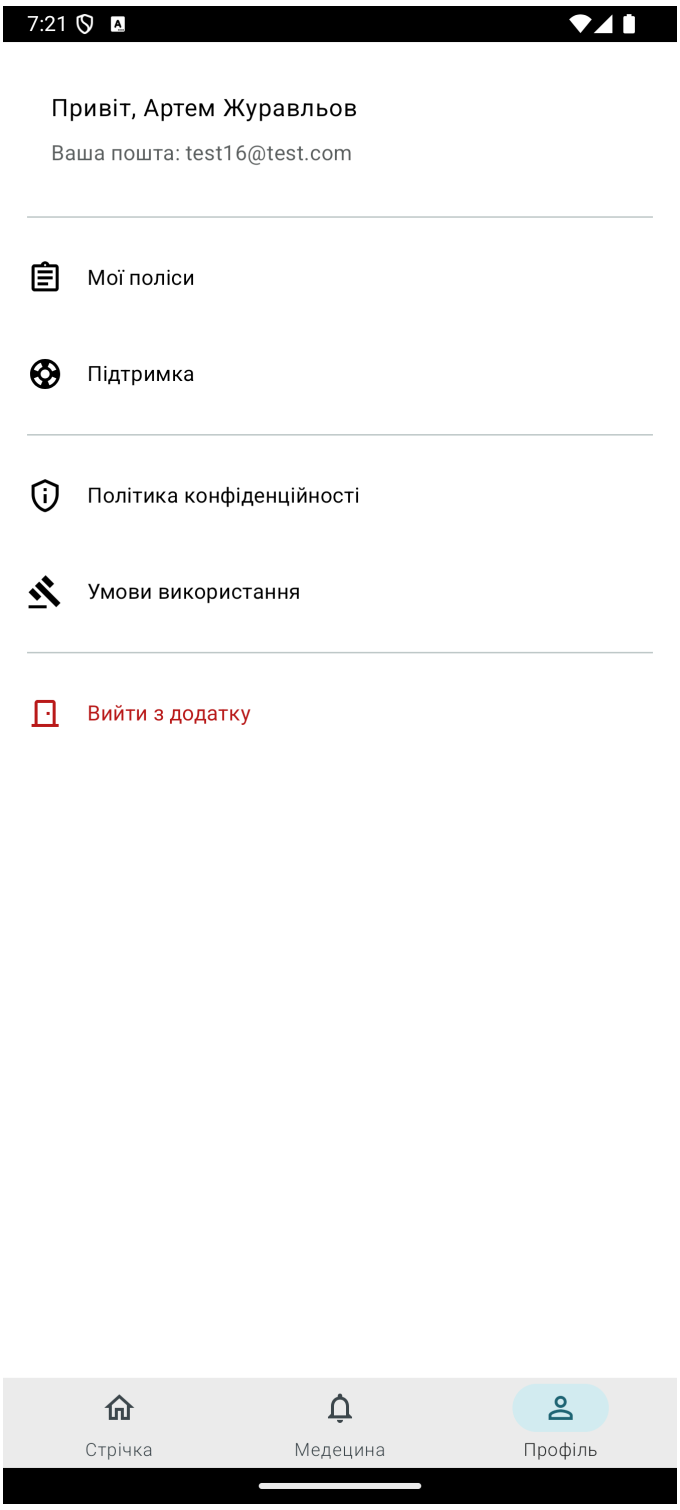


Рисунок 3.6 Особистий кабінет користувача.

Рисунок 3.7 Відображення карти з медичними закладами для навігації користувача.

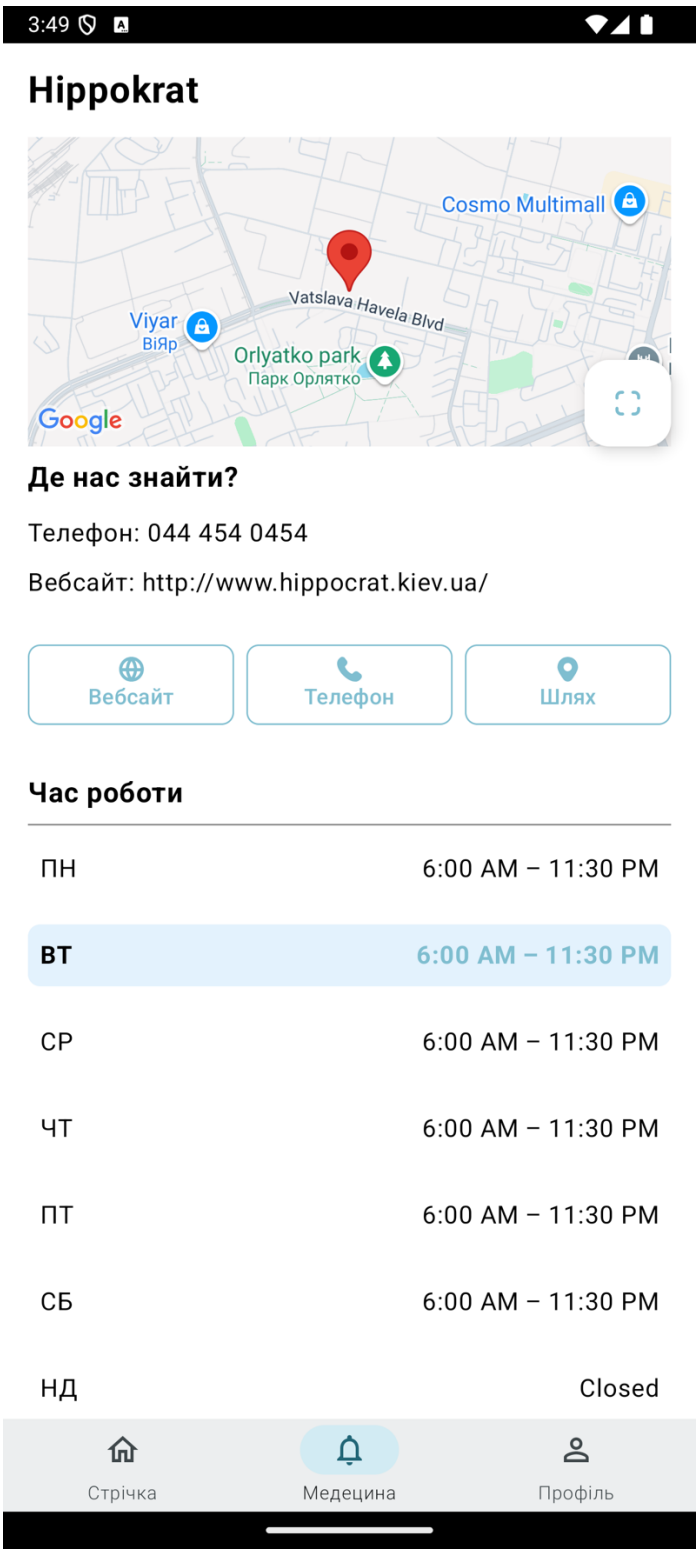


Рисунок 3.8 Детальна інформація про медичний заклад із контактами та часом роботи.

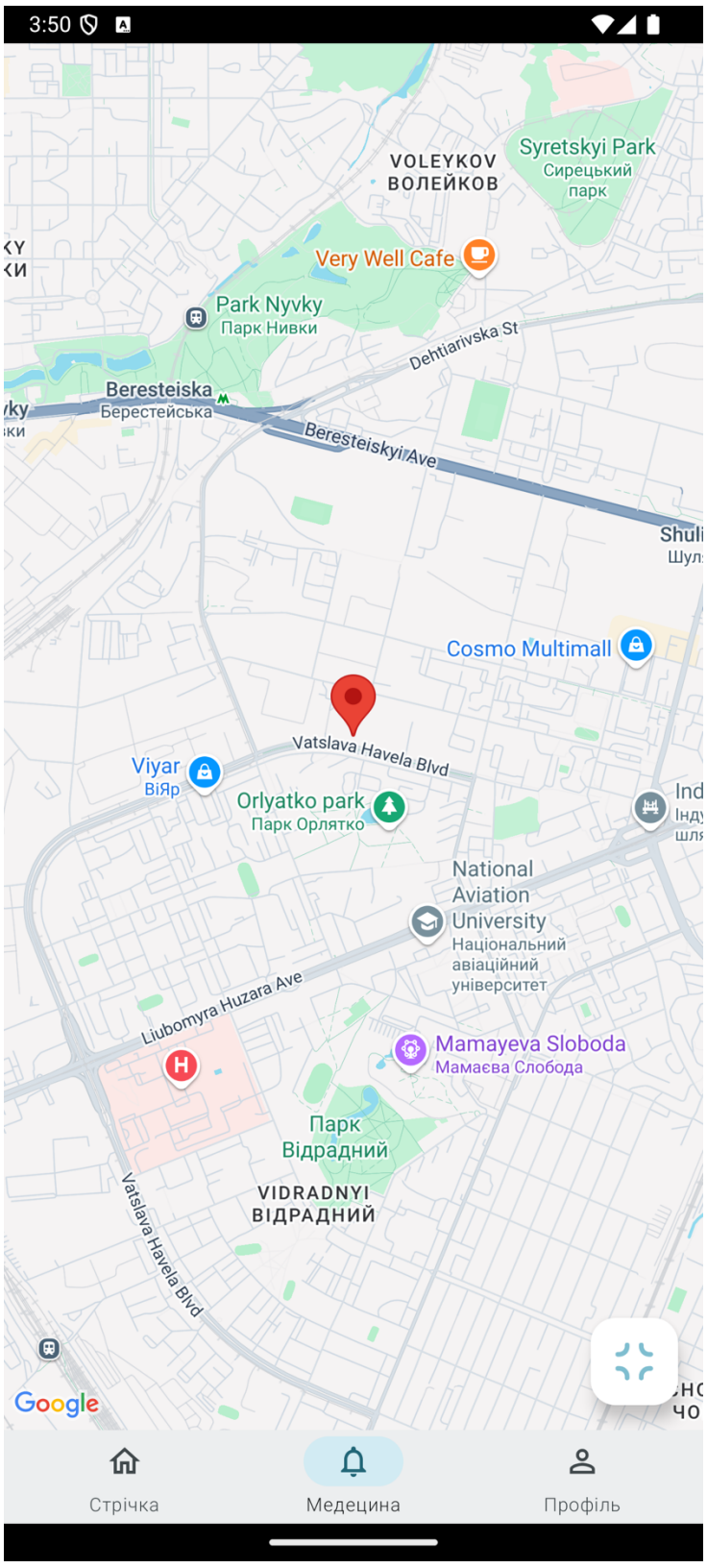


Рисунок 3.9 Список медичних закладів, доступних у додатку для вибору користувачем

ДОДАТОК В

Демонстраційні матеріали

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Гібридний мобільний додаток медичного страхування на основі Kotlin Multiplatform

Журавльов Артем Геннадійович
студент навчальної групи КНДМ-62
спеціальність 122– Комп'ютерні науки
керівник Іщеряков Сергій Михайлович

Київ - 2024

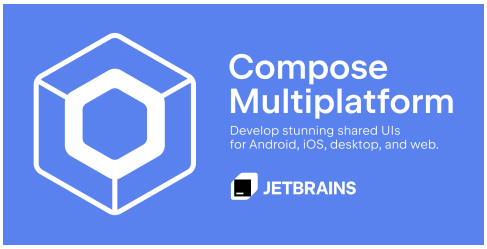
Вступ

- **Мета роботи:**
- Оцінка ефективності Kotlin Multiplatform у розробці гібридних додатків, її вплив на процес розробки, продуктивність, стабільність та повторне використання коду.
- **Об'єкт і предмет дослідження:**
- **Об'єкт:** Процес розробки гібридних мобільних додатків.
- **Предмет:** Застосування Kotlin Multiplatform для медичних додатків, вплив на швидкість, продуктивність і стабільність роботи.
- **Методи дослідження:**
- Аналіз підходів до кросплатформеної розробки.
- Проектування архітектури додатку.
- Інтеграція бізнес-логіки з Kotlin Multiplatform.
- Тестування та оптимізація продуктивності.

Вступ

- **Основні результати:**
- Створено архітектуру додатку із спільною бізнес-логікою та адаптивним інтерфейсом.
- Інтегровано Ktor та Firebase Cloud Messaging для обробки даних і сповіщень.
- Реалізовано прототип, що дозволяє:
  - Переглядати інформацію про медичні заклади.
  - Керувати профілем користувача.
  - Отримувати сповіщення про статус полісів.
- **Особливості:**
- Оптимізація продуктивності.
- Забезпечення безпеки даних.
- Покращення користувацького досвіду.

Технології та інструменти



Kotlin Multiplatform

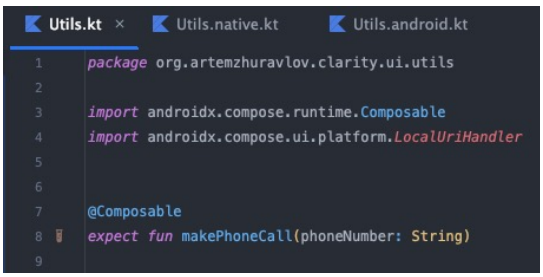
- **Kotlin Multiplatform (KMP):**
- Головна технологія для спільної бізнес-логіки.
- Забезпечує швидкість розробки та повторне використання коду для Android і iOS.
- **Compose Multiplatform (CMP):**
- Інструмент для створення адаптивного інтерфейсу користувача.
- Зберігає нативний досвід для кожної платформи.
- **Інтеграція з бекендом:**
- **Ktor:** Використовується для HTTP-запитів.
- **Firebase Cloud Messaging (FCM):** Система для надсилання push-сповіщень.
- **Бази даних:**
- **Realm:**Легка, швидка база даних для локального збереження даних на Android та iOS.
- **Keychain:** Збереження даних для iOS.

Архітектура системи

- **Структура програми:**
- **Спільний код:** реалізовано бізнес-логіку, роботу з базою даних (Realm) та взаємодію з бекенд-сервісами через Ktor.
- **Платформо-залежний код:** включає специфічні функції для Android та iOS, такі як інтеграція з платформеними API для роботи з інтерфейсом користувача та обробки push-сповіщень (Firebase Cloud Messaging).
- **Архітектурний підхід:**
- **MVVM (Model-View-ViewModel):** розділення бізнес-логіки (Model), логіки представлення (ViewModel) та інтерфейсу користувача (View) для підвищення тестованості та підтримки.
- **Clean Architecture:** забезпечує чітке розділення рівнів програми:
  - **Рівень інтерфейсу користувача (UI):** реалізований за допомогою Compose Multiplatform.
  - **Рівень бізнес-логіки:** спільний код для обробки даних та логіки.
  - **Рівень зберігання даних:** інтеграція з Realm для локального зберігання та бекенд-сервісами для синхронізації.

Інтеграція з платформозалежними функціями

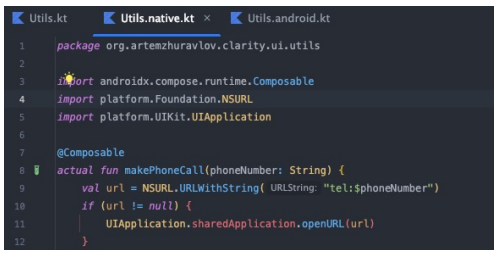
Однією з основних переваг Kotlin Multiplatform є можливість гнучкої інтеграції з платформозалежними функціями, такими як GPS, камери, push-повідомлення та біометрична автентифікація. Завдяки використанню Kotlin Native, розробники можуть взаємодіяти з нативними API кожної платформи, зберігаючи спільний код для бізнес-логіки.

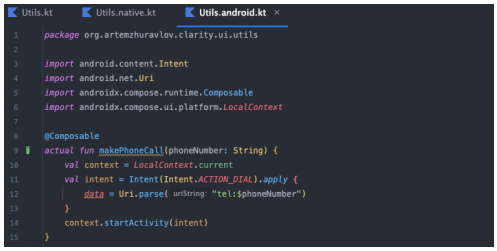


Інтеграція з платформозалежними функціями

Основні аспекти інтеграції:

- **Доступ до нативних бібліотек:** через очікувані та реалізовані функції (expect/actual).
- **Гнучкість та продуктивність:** забезпечення швидкого виконання коду завдяки прямому доступу до нативних функцій.
- **Масштабованість:** можливість поступового додавання нативних функцій, що дозволяє оптимізувати інтеграцію.



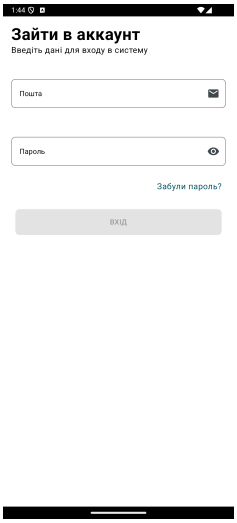


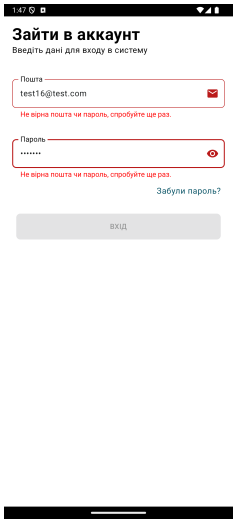
Функціональність розробленої системи

- **Основні функції:**
- Авторизація: безпечний вхід до системи за допомогою електронної пошти.
- Управління профілем: редагування персональних даних користувача.
- Перегляд полісів: доступ до інформації про страхові поліси.
- Карта клінік: інтерактивна карта з пошуком медичних закладів.
- **Інтерфейс системи:**
- Реалізовано сучасний і зручний дизайн для підвищення користувацького досвіду.
- Представлені скріншоти, що демонструють основні функції додатку.

Функціональність розробленої системи

Зображення з додатку, функція авторизації





Інтеграція з бекенд-сервісами

- **Взаємодія з сервером:** Реалізовано передачу даних між клієнтською частиною додатку та сервером за допомогою API, що забезпечує обробку запитів у реальному часі.
- **Використання Ktor:** Застосовано для побудови надійної та масштабованої архітектури обміну даними.
- **Firebase Cloud Messaging (FCM):** Використано для надсилання сповіщень користувачам про оновлення статусу полісів або важливі події.

Ktor

Firebase

Оптимізація та користувацький досвід

- **Оптимізація продуктивності:** Використання Kotlin Multiplatform та Compose Multiplatform дозволило зменшити час розробки й забезпечити стабільну роботу додатку на різних платформах.
- **Безпека даних:** Реалізовано шифрування даних користувачів, безпечну авторизацію, а також зберігання даних у Realm для захисту інформації.
- **UX/UI особливості:** Інтуїтивно зрозумілий інтерфейс, адаптація під Android та iOS, легкий доступ до основних функцій, таких як перегляд полісів і карти клінік.

Результати та переваги

- **Досягнуті результати:** Створено прототип мобільного додатку для медичного страхування з функціями авторизації, управління профілем, перегляду страхових полісів, та інтерактивною картою клінік.
- **Переваги Kotlin Multiplatform:**Спільна кодова база зменшила витрати часу на розробку.
- Збереження нативного досвіду користувача для Android та iOS.
- Легка інтеграція з бекенд-сервісами (Ktor, Firebase).
- **Висновки:**Забезпечено високу продуктивність додатку.
- Підтримується стабільність коду на різних платформах.
- Оптимізовано процес розробки та підтримки додатку.

Висновки

- **Узагальнення результатів:**
- Проведено дослідження можливостей Kotlin Multiplatform для кросплатформеної розробки.
- Створено прототип додатку для медичного страхування, який відповідає сучасним вимогам ринку.
- Реалізовано ключові функції: авторизація, управління профілем, інтеграція з бекенд-сервісами та карта клінік.
- **Практична цінність роботи:**
- Знижено витрати часу та ресурсів завдяки спільному коду.
- Підвищено зручність розробки та підтримки завдяки використанню KMP.
- Забезпечено високу продуктивність та стабільність роботи додатку.
- **Напрями подальших досліджень:**
- Впровадження нових функцій, таких як рекомендації полісів за допомогою AI.
- Оптимізація роботи додатку для більшої кількості платформ (Web, Desktop).
- Поглиблене дослідження нових можливостей Kotlin Multiplatform у поєднанні з іншими сучасними технологіями.