**Київський національний торговельно-економічний університет**
**Кафедра інформаційних технологій**

ВИПУСКНИЙ КВАЛІФІКАЦІЙНИЙ ПРОЕКТ (РОБОТА)
на тему:

# «Розробка месенджера для ІТ компанії»

*(за матеріалами ТОВ "Прайм-Груп" м. Київ)*

Студент 7м групи, факультету обліку, аудиту та інформаційних систем, денної форми навчання спеціальності 122 «Комп'ютерні науки» Лисянюк Дмитро Сергійович

_____
*(підпис студента)*

Науковий керівник
кандидат фізико-математичних наук, доцент Нагірна Алла Миколаївна

_____
*(підпис наукового керівника)*

Гарант освітньої програми
доктор технічних наук, професор Краскевич Валерій Євгенійович

_____
*(підпис гаранта освітньої програми)*

**Київ 2018**

# ЗМІСТ

Анотація

Лисянюк Д.С.

Розробка месенджера для IT компанії

У даній роботі описуєтеся розробка мобільного клієнту дня передачі текстових повідомлень . Система обміну повідомленнями є однією з найбільш доступних і потрібних засобів спілкування в Інтернеті, в корпоративних та локальних мережах.

Ключові слова: РОЗРОБКА МОБІЛЬНИХ ДОДАТКІВ, РЕАКТИВНЕ ПРОГРАМУВАННЯ, ХМАРНІ БАЗИ ДАНИХ, ТЕСТУВАННЯ.

Developing a messenger for an IT company

This work describes the development of a mobile client for the sending text messages. Messaging system is one of the most accessible and necessary means of communication on the Internet, corporate and local networks.

Keywords: MOBILE DEVELOPMENT, REACTIVE PROGRAMMING, CLOUD DATABASES, TESTING.

# ВСТУП

У наш час смартфони використовуються для самих різних завдань, однак головна з них, звичайно, спілкування. Варіантів спілкування зараз є безліч, і одним з них є обмін повідомленнями.

**Актуальність:** З появою мобільного зв'язку, для обміну текстовими повідомленнями користувачі мобільних телефонів могли використовувати лише сервіс коротких повідомлень SMS. З розвитком технологій людям стали доступні мультимедійні повідомлення MMS, а пізніше, коли телефони стали більш просунутими, почав зароджуватися клас додатків для миттєвого обміну повідомленнями (IM, инстант-месенджери). Для цього потрібен лише смартфон з виходом в Інтернет і один з численних IM-клієнтів.

**Метою** дипломної роботи є - показати сучасні методи та підходи розробки мобільних додатків на операційній системі Android та написання реалізації додатку з обміном миттєвими повідомленнями.

**Завдання** дипломної роботи:

1. Показати особливості програмування для ОС Android
2. Розібрати сучасні технології в розробці додатку для ОС Android
3. Розробити серверну частину додатку на основі хмарної бази даних Firebase Cloud Firestore для зберігання повідомлень.
4. Виконати повний цикл розробки мобільного додатку
5. Опублікувати додаток у Google Play

**Предметом** є реалізація синхронізованої бази даних Firebase Cloud Firestore для додатку з обміну повідомленнями , а також розробка мобільного додатка

.

# Розділ 1. Обмін миттєвими повідомленнями в ОС Android

## 1.1 Система обміну миттєвими повідомленнями

Система обміну повідомленнями є однією з найбільш доступних і потрібних засобів спілкування в Інтернеті, в корпоративних та локальних мережах. Служби обміну повідомлень розділяються на служби обміну повідомленнями в режимі оффлайн (поштові системи E-mail) та служби миттєвих повідомлень (Internet Relay Chat і Instant Messaging Service) в режимі онлайн.

Системи обміну повідомленнями мають свої комунікаційні мережі, більшість з яких побудовані по архітектурі «клієнт-сервер».

Обмін повідомленнями в режимі оффлайн здійснюється за рахунок взаємодія двох програм - поштового сервера і поштового клієнта. Для роботи з електронною поштою можна використовувати як поштові клієнти, так і поштові веб - інтерфейси, які розташовуються на поштових веб - серверах. За допомогою веб - інтерфейсу можна працювати з поштою безпосередньо на поштових веб - серверах.

Поштові системи на основі WWW дозволяють обробляти поштові повідомлення в Інтернеті за допомогою звичайного браузера, а не поштової програми. Вони працюють за принципом «2 в 1», поєднуючи в собі функції поштового сервера і поштового клієнта.

Служба IRC (Internet Relay Chat або Чат) є першим засобом для онлайнового спілкування, яка надає великий вибір каналів (тем) для проведення дискусій з однодумцями. Чат - це текстовий діалог в реальному масштабі часу.

Ця служба заснована на мережевій архітектурі клієнт-сервер, тому для онлайнового спілкування в Інтернет необхідно на ПК встановити клієнтську програму (IRC-клієнт). При запуску програми - клієнта, вона встановлює з'єднання з обраним IRC-сервером. Так як IRC-сервери мережі об'єднані між собою, то для спілкування досить підключиться до одного з її серверів. При підключенні до сервера IRC користувач бачить список доступних тем (каналів), в яких він може спілкуватися.

Спочатку служба IRC мала одну мережу IRC, яка згодом розділилася на кілька IRC - мереж. Ці IRC-мережі не пов'язані один з одним і мають свої імена (DALnet, IRCnet, UNDERnet, RusNet, WeNet, IrcNet.ru і т.д.). Усередині кожної IRC-мережі існують свої тематичні області або канали. В Інтернеті можна скачати IRC-клієнти для Unix-подібних ОС, OS / 2, Windows-систем і мобільних телефонів.

Для спілкування в чаті можна використовувати як IRC-клієнти, так і Web-чати. Web-чати призначені для обміну повідомленнями на сервері (веб-сторінці) за допомогою браузера, в цьому випадку встановлювати на ПК клієнтську програму не потрібно. Web-чат - це веб-сторінка, на якій можна в реальному часі спілкуватися з іншими відвідувачами.

Результатом розвитку чату стала служба миттєвих повідомлень (Instant Messaging Service, IMS). IMS - це одна з технологій, що забезпечує комунікації в мережах Інтернет. У службі миттєвих повідомлень крім текстових повідомлень можна передавати, картинки, відео, аудіо, файли.

Ця служба має свої мережі. Мережева архітектура IMS побудована за принципом клієнт-сервер. Клієнтська програма IMS, яка призначена для ведення бесіди і миттєвого обміну повідомленнями в режимі онлайн через служби миттєвих повідомлень, називається месенджером (Instant messengers, IM).

Як правило, мережі обміну мають окремий сервер (деякі мережі є децентралізованими), до якого підключаються месенджери, і свої протоколи

взаємодії. Більшість мереж служби миттєвих повідомлень використовують закриті або пропрієтарні протоколи (власні протоколи, що належать тільки одній мережі) обміну інформацією. В основному в кожній з таких мереж застосовується свій месенджер.

Між різними мережами IMS зазвичай відсутні взаємозв'язку, тому месенджер однієї мережі, наприклад Telegram не може зв'язатися з месенджером мережі Skype. Це означає, що для ведення спілкувань між собою користувачі повинні зареєструватися в одному і тому ж сервісі і встановити їх месенджери.

Але існують і альтернативні месенджери для служб миттєвих повідомлень, які можуть одночасно працювати в декількох мережах. Наприклад, безкоштовний відкритий мультипротокольний модульний клієнт (месенджер) Miranda IM (або Trillian, Pidgin) дозволяє підключатися одночасно до декількох мереж, що позбавляє від необхідності встановлювати окремий месенджер для кожної мережі.

Крім того, в якості альтернативи пропрієтарним протоколами для IM був розроблений відкритий протокол Jabber (джаббер - сімейство протоколів і технологій) або XMPP, який використовується в багатьох месенджерах (Jabber-клієнти: Psi, Miranda IM, Tkabber, JAJC, Pandion та інші). Jabber - система миттєвого обміну повідомленнями і присутності між будь-якими двома абонентами Інтернет на основі відкритого протоколу XMPP, що використовує формат XML.

У мережі не існує єдиного центрального сервера, Jabber є децентралізованою (з децентралізованими серверами), що розширюється і відкритою системою. Всі бажаючі можуть відкрити свій сервер миттєвих повідомлень, реєструвати на ньому користувачів і взаємодіяти з іншими серверами Jabber. Jabber використовується для організації спілкування в Інтернет, локальних і корпоративних мережах.

Сучасні месенджери надають користувачам багато корисних функцій, таких як IP-телефонія, індикація про мережевий статус користувачів і т.д. Для спілкування в службі миттєвих повідомлень можна використовувати як десктопний або мобільний IM-клієнт (месенджер), так і веб-версію клієнта (наприклад, Google Talk Gadget, JWChat, Meebo, MDC і т.д.).

Список основних функцій, які можуть надавати сучасні месенджери служб миттєвих повідомлень:

- чат (текстовий і голосовий);
- VoIP сервіси: дзвінки на комп'ютер, дзвінки на стаціонарні і мобільні телефони;
- можливість відправки SMS;
- передача файлів;
- інструменти для спільної роботи в режимі реального часу;
- можливість спілкуватися в чаті безпосередньо на веб-сторінці;
- нагадування та оповіщення;
- зберігання історії спілкування по кожному контакту;
- індикація про мережевий статус користувачів (в мережі, немає на місці і т.д.), занесених до списку контактів.

Однією з основних особливостей багатьох клієнтів миттєвого обміну повідомленнями є можливість бачити, чи є друг або співробітник в мережі та підключений через вибрану службу. По мірі того, як технологія розвивалася, багато клієнтів IM додали підтримку для обміну не просто текстовими повідомленнями, а для здійснення таких дій, як передача файлів та обмін зображеннями в сеансі чату.

Миттєвий обмін повідомленнями відрізняється від електронної пошти при безпосередній обміні повідомленнями. IM також має тенденцію бути на основі сеансів, маючи початок і кінець. Оскільки IM призначений для імітації особистих розмов, окремі повідомлення часто є короткими. З іншого боку,

електронна пошта зазвичай відображає стиль довготривалої форми, написання листа.

Як правило, користувачі чату повинні знати ім'я користувача чи ідентифікатор, щоб розпочати сеанс чату або додати їх до свого списку контактів або списку друзів. Після того як призначений одержувач був ідентифікований та обраний, відправник відкриває вікно чату, щоб розпочати сеанс.

Для ІМ, щоб працювати, як передбачено, обидва користувачі повинні бути в мережі одночасно, хоча майже всі платформи миттєвих повідомлень дозволяють асинхронну взаємодію між користувачами в режимі онлайн і офлайн. Якщо обмін повідомленнями в автономному режимі не підтримується, спроба ІМ для недоступного користувача призведе до повідомлення про те, що передача не може бути завершена. Крім того, передбачуваний одержувач повинен бути готовий прийняти миттєві повідомлення, оскільки можна налаштувати клієнт ІМ для відхилення деяких користувачів.

Коли ІМ отримується, він сповіщає одержувача вікно, що містить вхідне повідомлення. Або в залежності від налаштувань користувача вікно може вказувати, що ІМ надійшов разом із запитом на прийняття чи відхилення. Багато клієнти чату також чутно повідомляють користувачеві чутливий звук, наприклад, дзвін або чірік. Користувач також може отримувати візуальне сповіщення, натискаючи вікно чату або його піктограму на панелі завдань, коли з'являється повідомлення.

Хоча клієнти ІМ частіше засновані на власних протоколах в минулому, і для цього потрібні обидва користувачів використовувати одне і те ж програмне забезпечення для зв'язку - прийняття відкритих стандартів стало більш поширеним явищем. Це дозволило збільшити кількість мультиплатформенних миттєвих повідомлень, таких як Pidgin і Trillian.

Ще одним важливим зрушенням у ІМ є те, як він доступний і доставлений. Тривалий час розгортання як клієнт для настільних комп'ютерів, який треба

було завантажувати та встановлювати, миттєві повідомлення частіше зустрічаються як функція в рамках іншої веб-служби або хмарної служби - таких як Facebook, Gmail та Skype - або як мобільний додаток. наприклад, WhatsApp Messenger.

Особливості обміну миттєвими повідомленнями

Обмін текстом вже давно є головною функцією обміну миттєвими повідомленнями, але тепер це є однією особливістю багатьох. Можливість вставляти зображення та емоції в повідомлення є стандартною для багатьох клієнтів, як і для передачі файлів. Facebook Messenger навіть дозволяє користувачам надсилати гроші через чат. Численні клієнти зараз підтримують ескалацію від IM до інших способів спілкування, таких як груповий чат, голосові дзвінки або відеоконференції.

Також багато месенджерів підтримують зберігання повідомлень на сервері, завдяки цьому можна отримати історію повідомлень з користувачем, навіть якщо він оффлайн.

## 1.2 Операційна система Android

Android – операційна система для смартфонів, інтернет-планшетів, електронних книг, цифрових програвачів, наручних годинників, ігрових приставок, нетбуків, смартбуків, окулярів Google, телевізорів та інших пристроїв. В майбутньому планується підтримка автомобілів і побутових роботів. Заснована на ядрі Linux і власної реалізації віртуальної машини Java від Google. Спочатку розроблялася компанією Android, Inc., яку потім купила Google. Згодом Google ініціювала створення альянсу Open Handset Alliance (OHA), який зараз займається підтримкою і подальшим розвитком платформи. Android дозволяє створювати Java-додатка, що керують пристроєм через розроблені Google бібліотеки. Android Native Development Kit дозволяє перенести бібліотеки і компоненти додатків, написані на Сі та інших мовах.

Додатки для Android пишуться на мові програмування Java. Інструменти Android SDK (Software Development Kit - комплект розробки програмного забезпечення) компілюють написаний вами код і всі необхідні файли даних і ресурсів - в файл APK - програмний пакет Android, який представляє собою файл архіву з розширенням .apk. У файлі APK знаходиться все, що потрібно для роботи Android-додатку, і він дозволяє встановити додаток на будь-якому пристрої під управлінням системи Android.

### 1.3 Огляд існуючих додатків обміну повідомленнями

На сьогоднішній день найбільш популярними месенджерами вважаються:

**Telegram –** (рис. 1.3.1) є одним з найпопулярніших додатків для обміну повідомленнями з гарантією конфіденційності. Месенджер має, 256-бітове симетричне AES-шифрування, 2048-бітове шифрування RSA і багато іншого. Це робить додаток одним з найбезпечніших месенджерів. Додаток також має крос-платформену підтримку, груповий чат, підтримку GIF-анімації, обмін мультимедійними файлами багатьох форматів, ботів і багато іншого.


Рис. 1.3.1. Месенджер Telegrem

**Slack** - (рис. 1.3.2) один з кращих корпоративних месенджерів. Додаток має стриманий, професійний зовнішній вигляд. Можна створювати канали,

здійснювати голосові виклики та багато іншого. Додаток також підтримує сторонні додатки, такі як Dropbox, Google Диск, Salesforce і інші. Користувачі, також можуть підключатися до декількох серверів Slack.

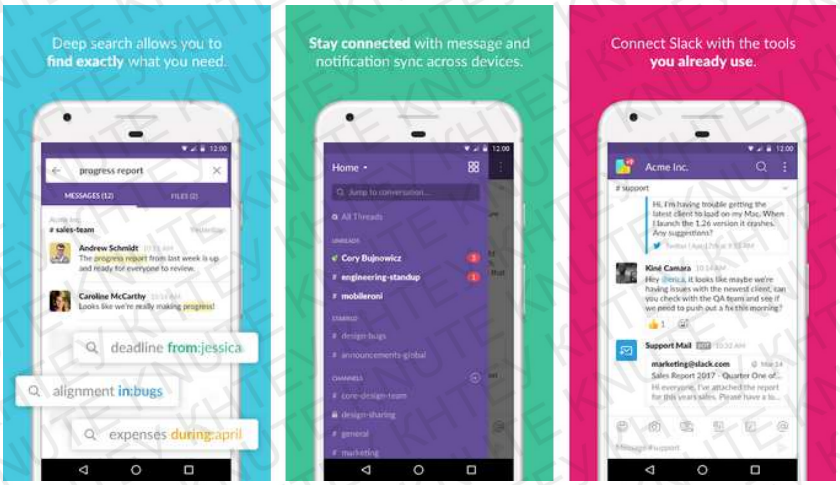


Рис. 1.3.2 Месенджер Slack

**Skype** - (рис. 1.3.3) є одним з найбільш впізнаваних додатків для обміну повідомленнями та здійснення відео дзвінків. Skype може відправляти текстові повідомлення, здійснювати відео і голосові дзвінки між іншими користувачами цієї програми.


Рис. 1.3.3 Месенджер Skype

Також може телефонувати на реальні номери телефонів, за невелику плату. У додатку є кілька додаткових функцій, кому-то можуть корисними, а кому-то марними. Це залежить від ваших уподобань і потреб.

**Viber** - (рис. 1.3.4) Популярний сервіс для безкоштовного обміну повідомленнями, здійснення звичайних і відеодзвінків, передачі файлів, посилань і інформації про місцезнаходження. Viber синхронізується зі списком контактів на смартфоні, визначаючи, хто з них користується сервісом, і додає їх в контакт-лист. Можна створювати, і приєднаються до групових (до 250 співрозмовників) чатах, стежити за публічними акаунтами. Чати можна ховати, а також спілкуватися по зашифрованому каналу зв'язку.

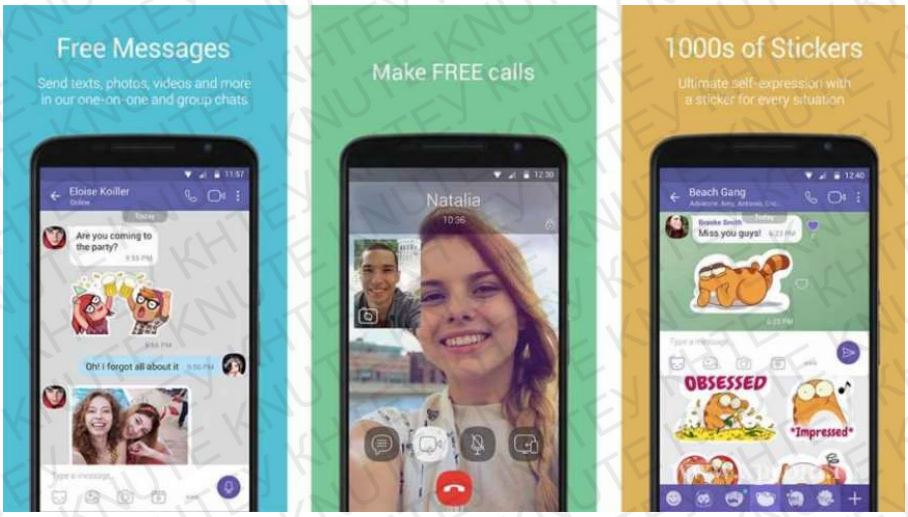


Рис. 1.3.4 Месенджер Viber

**Whatsapp** – (рис. 1.3.5)Додаток для обміну повідомленнями та дзвінків. Відправка файлів, голосових і відеоповідомлень, зашифрована листування, прив'язка до номера телефону без необхідності реєструвати обліковий запис: все, що повинен вміти сучасний месенджер вміє і Whatsapp. А ще це найпопулярніший в світі сервіс для обміну повідомленнями - навіть популярніше Skype або Viber.

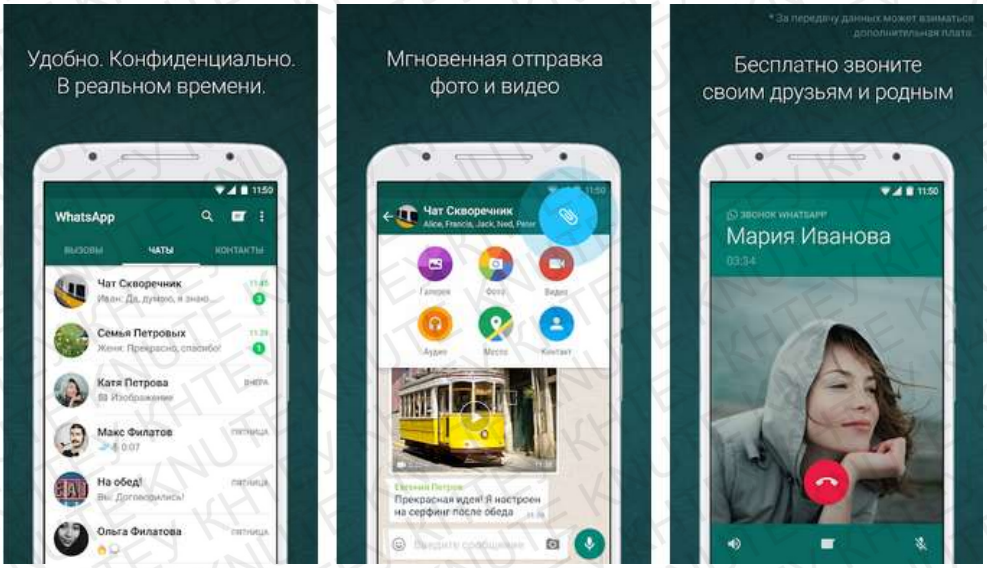


Рис. 1.3.4 Месенджер Whatsapp

**Facebook Messenger** – (рис.1.3.5) є одним з найпопулярніших додатків для обміну повідомленнями. Для цього у Фейсбук є два додатки. Звичайне включає в себе всі функції, такі як групові чати, відеодзвінки, наклейки і багато іншого. Версія Lite - це просто додаток для чату і голосових викликів, без надмірностей.


Рис. 1.3.5 Месенджер Facebook Messenger

Отже, розглянуто технологію Instant Messaging Service для передачі повідомлень в режимі онлайн та інші технології передачі повідомлень.

Розглянуто операційну систему Android та популярні IM-клієнти для OS Android.

# Розділ 2. Основні технології в додатку

## 2.1 Хмарна база даних Firebase Cloud Firestore

Основною частиною додатку обміну повідомленнями є сервер з яким спілкуйтеся додаток, в якому повинна виконуватися вся логіка передачі повідомлень, які передаються від одного користувача іншому, на якому будуть зберігатися всі повідомлення та данні о користувачах, але для даного проекту написання серверної частина зайняло б дуже багато часу, тож було вирішено використати хмарну базу даних Firebase Firestore.

Firebase Cloud Firestore - це гнучка, масштабована база даних для мобільних пристроїв, веб-сайтів та серверів Firebase та Google Cloud Platform. Подібно до бази даних Firebase Realtime, вона зберігає дані в синхронізації між клієнтськими програмами за допомогою прослухувань у реальному часі та пропонує офлайн-підтримку для мобільних пристроїв та Інтернету, щоб ви могли створювати програми, які працюють незалежно від доступу мережі або підключення до Інтернету. Cloud Firestore також пропонує бездоганну інтеграцію з іншими продуктами Firebase та Google Cloud Platform, включаючи Cloud функції. Також це хмарна NoSQL база даних, яка дозволяє користувачам отримувати доступ до iOS, Android та веб-програм за допомогою власних SDK. Cloud Firestore також доступний у власних Node.js, Java, Python і Go SDK, API REST та RPC. [17]

Основні властивості:

1. Гнучкість - модель даних Cloud Firestore підтримує гнучкі ієрархічні структури даних. Збереження даних в документах, організованих у колекції. Документи можуть містити складні вкладені об'єкти та підколекції.

2. Експресивні запити - у Cloud Firestore можна використовувати запити для отримання окремих, конкретних документів або для отримання всіх документів у колекції, які відповідають параметрам запиту. Запити можуть містити багаторазові, приєднані фільтри та комбінувати фільтрацію та сортування. Вони також індексуються за умовчанням, тому ефективність запитів пропорційна розміру набору результатів, а не набору даних. Оновлення в режимі реального часу. База даних Cloud Firestore використовує синхронізацію для оновлення даних на будь-якому підключеному пристрої. Тобто, підключені до бази пристрої будуть отримувати всі зміни або внесені нові дані. Тим не менш, Cloud Firestore також призначений для простого, одноразового отримання запитів.

3. Офлайн-підтримка. Cloud Firestore кешує дані, які активно використовується додатком, тому додаток може записувати, читати, прослуховувати та запитувати дані, навіть якщо пристрій знаходиться поза мережею. Коли пристрій повертається в режимі он-лайн, Cloud Firestore синхронізує будь-які місцеві зміни з Cloud Firestore.

4. Призначена для масштабування. Cloud Firestore пропонує найкращу потужну інфраструктуру Google Cloud Platform: автоматичну багаторазову реплікацію даних, надійну послідовність та підтримку транзакцій. Cloud Firestore розробляли для обробки найважчих робочих навантажень бази даних з найбільших у світі додатків.

Дані Cloud Firestore зберігаються у документах, що містять поля, що відрізняються від значень. Ці документи зберігаються в колекціях, які є контейнерами для документів, які можна використовувати для організації ваших даних та створення запитів. Документи підтримують різні типи даних, від простих рядків і чисел, до складних вкладених об'єктів. Також можна створювати підколекції в документах і створювати ієрархічні структури даних, які масштабуються коли ваша база зростає. Модель даних Cloud Firestore

підтримує будь-яку структуру даних, яка найкраще підходить для додатку (рис. 2.1.1).

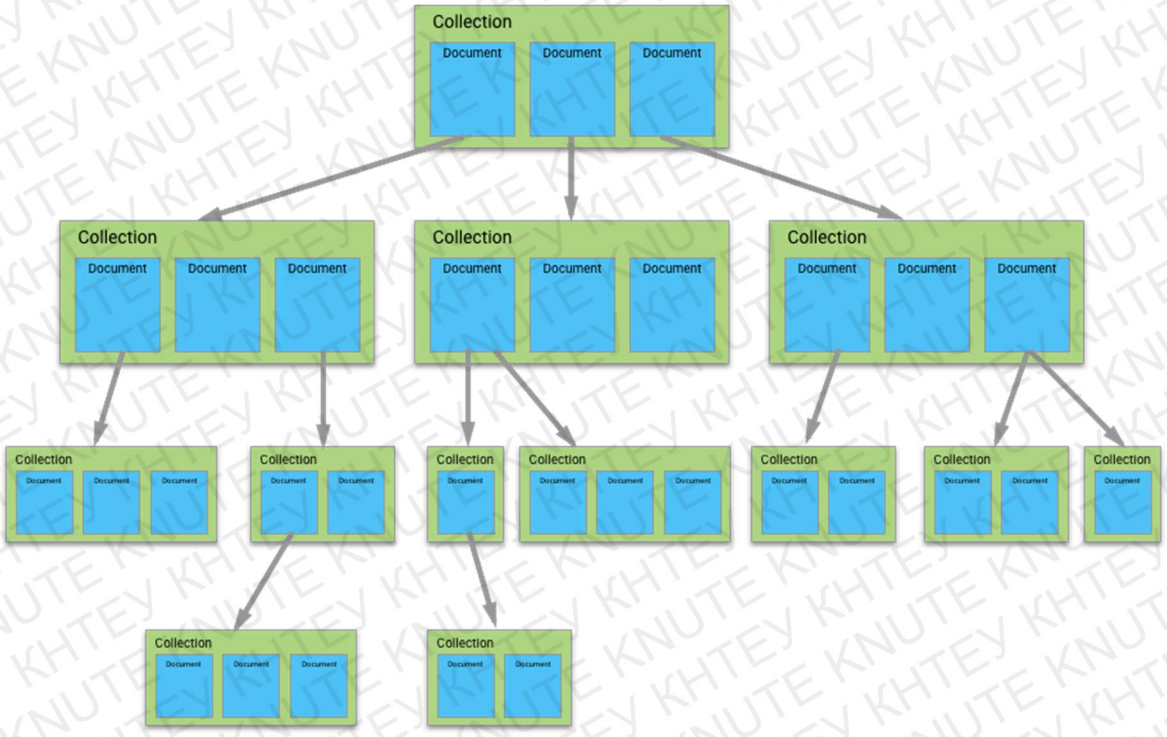


Рис. 2.1.1 Структура даних Cloud Firestore

Крім того, запити в Cloud Firestore є ефективними та гнучкими. Є можливість створювати неглибокі запити для отримання даних на рівні документа без необхідності отримувати всю колекцію або будь-які вкладені підколекції. Можливе додання сортування, фільтрування та обмеження для запитів або курсорів, щоб поширити результати.

З Cloud Firestore Security Rules можна зосередити увагу на створенні чудового досвіду роботи користувачів, не маючи необхідності керувати інфраструктурою або писати на сервері автентифікацію та код авторизації.

Security Rules передбачають контроль доступу та перевірку даних у простому, але виразному форматі. Щоб створити системи доступу на основі

користувачів та ролі, які захищають дані ваших користувачів, слід використовувати Firebase Authentication з правилами безпеки Cloud Firestore.

Усі Security Rules складаються з відповідності тверджень, які ідентифікують документи у базі даних та контролюють доступ до цих документів (рис. 2.1.2).

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /<some_path>/ {
      allow read, write: if <some_condition>;
    }
  }
}
```

Рис. 2.1.2 Security Rules для Cloud Firestore

Кожна запит на базу даних з бібліотеки мобільного веб-клієнта Cloud Firestore оцінюється відповідно до правил безпеки перед читанням або написанням будь-яких даних. Якщо правила заперечують доступ до будь-якого із зазначених шляхів до документа, весь запит не вдається.

Наведено кілька прикладів основних правил.

Дозволити на доступ до читання або запису на всіх документах будь-якому користувачеві, який увійшов до програми (рис. 2.1.3).

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth.uid != null;
    }
  }
}
```

Рис. 2.1.3 Security Rules на доступ до читання або запису на всіх документах.

Заборона на читання або запис до всіх документів для всіх користувачів за будь-яких умов (рис. 2.1.4).

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

Рис. 2.1.4 Security Rules на заборону доступу.

Дозволити доступ до читання або запису для всіх користувачів за будь-яких умов (рис. 2.1.5).

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true;
    }
  }
}
```

Рис. 2.1.5 Security Rules на дозвіл до даних.

Так як Firestore це база даних на основі колекцій та документів то показати структура даних можна в вигляді ієрархії папок та файлів.

Для додатку було розроблено таку модель бази даних:


Рис. 2.1.3 Структура бази даних для додатку.

База даних додатку містить в собі дві основні колекції users та dialogs. В колекції users, зберігаються всі зареєстровані користувачі, які містяться в окремих документах, назва яких сгенерована сервером, назва документа є ідентифікатором кожного користувача, в а самому документі зберігається інформація про ім’я та фото користувача. Колекція dialogs містить в собі всі документи діалогів або групових чатів користувачів, в яких міститься назва чату, масив ідентифікаторів користувачів які входять в діалог, останнє надіслане повідомлення та колекція з повідомленнями. Колекція з повідомленнями конкретного діалогу містить в собі всі документі повідомлень в яких знаходиться саме повідомлення, дата повідомлення та ідентифікатор користувача який надіслав повідомлення в діалог.

Але Firestore це не єдиний продукт з  Firebase який використовуються в додатку. Так як Firestore не може зберігати в собі файли, а точніше фотографії користувачів, то для додатку було використано Firebase Storage. Firebase Storage - це потужна, захищена, проста і економічно вигідна служба зберігання об'єктів,

в яку додаток буде завантажувати фотографії користувачів, а в базі даних буде зберігатися посилання на них.

## 2.2 Мови програмування Java та Kotlin

Для будь-якого розробника програм для мобільних пристроїв по всьому світу перша і найбільш популярна мова програмування для додатка Android - це Java, одна з причин, що мова є офіційною для розробки Android додатків.

Сама Java була розроблена компанією Sun Microsystems ще в 1995 році і використовується для широкого кола програмних додатків. Код Java запускається віртуальною машиною, яка працює на пристроях Android і інтерпретує код.

На жаль, Java також трохи складний, і це не чудова мова для вирішення проблеми, якщо ви новачок. Це найбільша перешкода, з якою стикаються люди, які планують зайнятися розробкою додатків Android. Java - це об'єктно-орієнтована мова програмування з незрозумілими темами, такими як конструктори, винятки з нульовим вказівником, перевірки винятків і багато іншого. Це не надто читабельно, і ви будете використовувати багато коду для простих речей. Тому останнім часом для розробки Android додатків використовується нова мова програмування Kotlin.

Kotlin нещодавно з'явився як "інша" офіційна мова для розробки Android.

Як і Java, Kotlin працює на віртуальній машині Java. Він також повністю сумісний з Java і не створює перешкод або збільшує розмір файлів.

Основна відмінність полягає в тому, що Kotlin вимагає менш "шаблонного коду", що означає, що це більш раціональна та зручна для читання система. Це також призводить до виникнення помилок, таких як винятки із нульової точки, і навіть виправдовує вас припинення кожного рядка точкою з комою. Це чудова мова програмування, якщо ви просто навчаєтесь розробляти додатки для Android вперше. [18]

Kotlin повністю підтримується в Android Studio 3.0 і вище, тому легко створювати нові проекти з файлами Kotlin, додати файли Kotlin до вашого існуючого проекту та конвертувати код мови Java у Kotlin.

Розглянемо основні переваги Kotlin в порівнянні з Java для розробки Android додатків:

1. Підвищує ефективність команди. Будучи досить ясним і компактним, мова дозволяє підвищити ефективність роботи команди завдяки своєму стислому та інтуїтивно зрозумілому синтаксису. Більше роботи можна зробити, оскільки він займає менше часу і менше рядків для написання та розгортання робочого коду.

2. Відповідає існуючому коду Java. Kotlin позиціонує як 100% Java-сумісну мову програмування. Це узгоджується з Java та всіма відповідними інструментами та рамками, що дозволяє поступово переходити на Kotlin. У випадку, якщо ваш продукт не може бути написаний тільки в Kotlin, обидві мови можуть бути зручно використані одночасно.

3. Легко обслуговується. Kotlin підтримує переважна більшість IDE, включаючи Android Studio та інші інструменти SDK. Це допомагає підвищити продуктивність розробників, оскільки вони можуть продовжувати працювати з набором інструментів, яким вони використовуються.

4. Менше помилок при розробці. Kotlin пропонує набагато більш чітку та компактну кодову базу, що робить код у виробництві більш стабільним та послідовним. Помилки виявляються під час компіляції, тому розробники можуть виправляти помилки перед виконанням.

Отже було вирішено для написання додатку використовувати мову програмування Kotlin для більш зручного написання коду. Було зроблено приклад порівняння коду написаного на Java та Kotlin (додаток А та Б).

Як видно з прикладу в Kotlin значно менше коду в порівнянні з Java, завдяки стандартній шаблонній логіці додану в синтаксис Kotlin, що робить код значно легшим до читання та розуміння.

## 2.3 Реактивне програмування з використанням RxJava 2

У реальному додатку все працює асинхронно. Є мережа, в яку додатки відправляють запити і з якої через тривалий час отримують відповіді. Програма не може блокувати основний потік виконання, тож робота з мережею повинна виконуватися у фоновому потоці. Програма також не може блокувати основний потік при роботі з файловою системою, базою даних, записи в сховище або навіть в shared preferences, так що доводиться виконувати ці операції в фонових потоках.

Тому для зручного використання асинхронності в розробці Android додатків на сьогоднішній день популярно використовувати реактивне програмування.

Реактивне програмування - це, в основному, асинхронне програмування на основі подій. Все можна уявити як асинхронний потік даних, який може бути спостережуваним (Observable) (рис. 2.3.1), і події будуть відбуватися коли потік випромінює данні. Можна створити потік даних з чого завгодно; зміни змінної, події кліків, http-запитів, зберігання даних, помилки та ін. Потік називається асинхронним, коли кожен модуль коду працює на власному потоці, виконуючи одночасно кілька блоків коду. [6]

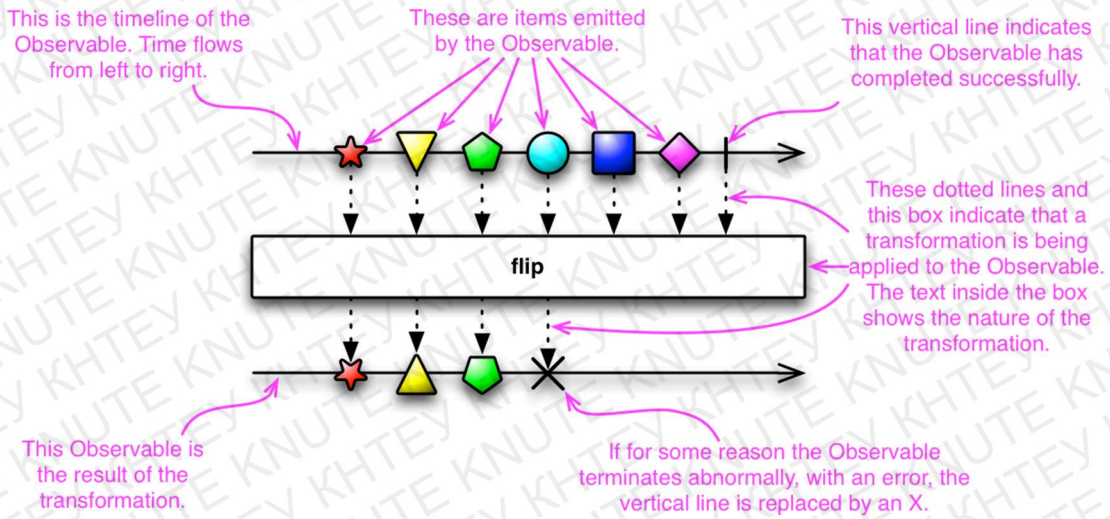


Рис. 2.3.1 Діаграма трансформації випромінюваних даних

Перевага асинхронного підходу полягає в тому, що, як кожне завдання виконується за власним потоком, все завдання може розпочатися одночасно, а час виконання всіх завдань еквівалентний довшому завданню списку. Коли справа доходить до мобільних додатків, оскільки завдання виконуються у фоновому потоці, ви можете досягти безперервної роботи без блокування основного потоку.

Реактивні розширення (ReactiveX або RX) - це бібліотека, яка слідує за принципами реактивного програмування, тобто складають асинхронні та події на основі програм з використанням спостережуваної послідовності. Ці бібліотеки забезпечують набір інтерфейсів та методів, які допомагають розробникам написати чистий і простий код.

Реактивні розширення доступні на декількох мовах C ++ (RxCpp), C # (Rx.NET), Java (RxJava), Kotlin (RxKotlin), Swift (RxSwift), PHP(RxPHP) та ін. Ми спеціально зацікавлені в RxJava і RxAndroid, оскільки Android - це наша сфокусована область.

Модель ReactiveX Observable дозволяє обробляти потоки асинхронних подій за допомогою тих самих простих, складних операцій, які використовуються для збірок елементів даних, таких як масиви. Це звільняє від заплутаних мереж зворотних викликів, і тим самим робить код більш читабельним і менш схильним до помилок.

RxJava - це Java-реалізація реактивного розширення (від Netflix). В основному це бібліотека, яка складається з асинхронних подій за шаблоном Observer. RxJava може створювати асинхронний потік даних у будь-якому потоці, перетворювати дані та віддавати його спостерігачем у будь-якому потоці. Бібліотека пропонує широкий спектр дивовижних операторів, таких як map, combine, merge, filter та багато іншого, які можна застосувати до потоку даних.

RxAndroid - це розширення для платформи Android, з небагатьма додатковими класами на вершині RxJava. Більш конкретно, планувальники вводяться в RxAndroid (AndroidSchedulers.mainThread()), який відіграє важливу роль у підтримці концепції багатопотокового використання в програмах Android. Планувальники в основному визначають потік, на якому виконується певний код, як на фоновому потоці, так і на основному потоці. Крім того, в RxAndroid входить бібліотека RxJava.

Також існує безліч планувальників, Schedulers.io() та AndroidSchedulers.mainThread() які широко використовуються в програмуванні Android.

RxJava - це всього два основних компоненти: Observable і Observer. На додаток до них є й інші речі, такі як планувальники, оператори та підписка.

Спостережуваний (Observable) - це потік даних, який виконує деяку роботу та випромінює дані.

Спостерігач (Observer) є контрчастиною Observable. Він отримує дані, що випромінює Observable.

Підписка (Subscription) - зв'язок між Observable і Observer називається підпискою. Можуть бути декілька спостерігачів, підписаних на єдиний Observable або навпаки.

Оператор або трансформація - змінюють дані, що випромінює Observable, перш ніж спостерігач їх приймає.

Планувальники (Schedulers) вказують потік, на якій Observable слід виділяти дані, і на який Observer повинен отримувати дані, тобто логіку асинхронного виконання програми.

Disposable використовується для розпорядження підпискою, коли спостерігач більше не хоче слухати Observable. У Android - Disposable дуже корисні, щоб уникнути витоків пам'яті та помилок коли нам вже не потрібно отримувати данні.

Скажімо, програма запускає довготривалий мережевий запит та оновлюєте інтерфейс користувача. Поки мережевий запит завершує свою роботу, якщо Activity або Fragment вже знищено, оскільки підписка спостерігача ще жива, вона намагається оновити вже знищену Activity. У цьому випадку він може кинути витік пам'яті. Отже, використовуючи Disposable, підписка може бути виключена, коли Activity знищується.

Розглянемо приклад реактивного програмування з використанням RxJava \ RxAndroid у додатку:

За приклад візьмемо логіку оновлення повідомлень та логіку відправлення нового повідомлення (додаток В):

Під час ініціалізації виконуються підписка на оновлення останніх 40 повідомлень з Firesote в методі useCase.getMessages(dialogId,Message-Observer(),limit). MessageObserver в нашому випадку це спостерігач (Observer) який очікує випромінення оновлень повідомлень та оновлює view згідно нових даних.

В ChatUseCase (додаток А) виконується основна логіка отримання та відправлення повідомлень. В методі buildUseCaseObservableUnit створюється новий Observable який створює нове повідомлення в базі Firestore.

```
class SentMessage(private val dialogPath: String, val message: Message) : Observable<Unit>() {

    override fun subscribeActual(observer: Observer<in Unit>) {
        FirebaseFirestore.getInstance()
                .collection( collectionPath: "$dialogPath/messages")
                .document()
                .set(message)
                .addOnCompleteListener { it: Task<Void!>
                    if (it.exception == null) {
                        observer.onNext(Unit)
                        observer.onComplete()
                    } else {
                        observer.onError(it.exception!!)
                    }
                }
    }
}
```

Рис. 2.3.2 Observable для створення нового повідомлення в базі даних

Далі SentMessage (рис. 2.3.2) виконує оператор OnErrorResumeNext (рис. 2.3.3) який в випадку помилки при створенні нового повідомлення виконає повторне підписування на новій потік даних(на інший Observable).

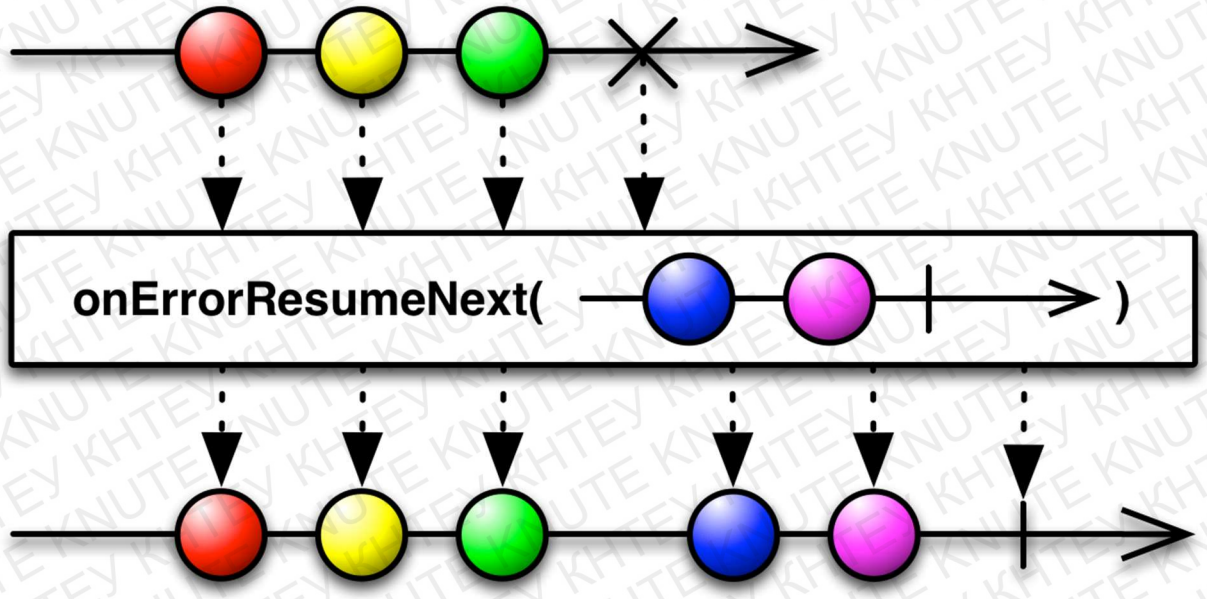


Рис. 2.3.3 Схема роботи оператору OnErrorResumeNext

Помилка на даному етапі може трапитися у випадку коли ще не створено діалог між двома користувачами, тому після помилки потік виконає повторне підписування на CreateUserDialog (рис. 2.3.4) який створює новий діалог між користувачами.

```
class CreateUserDialog(private val dialogPath: String, val message: Message) : Observable<Unit>() {
    override fun subscribeActual(observer: Observer<in Unit>) {
        val db = FirebaseFirestore.getInstance()
        db.runTransaction { transaction : Transaction ->
            val dialogId = dialogPath.replace( oldValue: "/dialogs/", newValue: "")
            val user = FirebaseAuth.getInstance().currentUser!!.uid
            if (!dialogId.contains(user))
                throw FirebaseFirestoreException("only user to user dialog creation supported",
                        FirebaseFirestoreException.Code.ABORTED)
            val user2 = dialogId.replaceFirst(user, newValue: "")
            transaction.set(db.document(dialogPath), Dialog(lastMessage = message,
                    users = hashMapOf(user to Date(), user2 to Date())).toUpdateModel())
            return@runTransaction
        }.addOnSuccessListener {
            observer.onNext(Unit)
            observer.onComplete()
        }.addOnFailureListener { it: Exception
            observer.onError(it)
        }
    }
}
```

Рис. 2.3.4 Observable для створення нового діалогу в базі даних

Далі після створення діалогу в операторі flatMap в ChatUseCase буде повторне створення повідомлення яке так і не було створено.

Наступним кроком в doOnNext ми повідомляймо другий Observable який відповідає за оновлення повідомлень в діалозі, що треба зробити запит знову, тому що він напевно виконався з помилкою оскільки діалогу на той час ще не було створено. В нашому випадку Observable який відповідає за оновлення повідомлень та Observable який відповідає за створення нового повідомлення ніяк не зв'язані між собою, та щоб зв'язати їх між собою було створено новий

PublishSubject (рис. 2.3.5), який буде повідомляти Observable який відповідає за оновлення повідомлень що треба оновити підписку на повідомлення діалогу. Отже Observable який відповідає за оновлення повідомлень виконує оператор retryWhen, в якому, в випадку помилки ми умовно підписуємося на події з PublishSubject , та після події Observable повністю перезапуститься та виконається напевно без помилки.

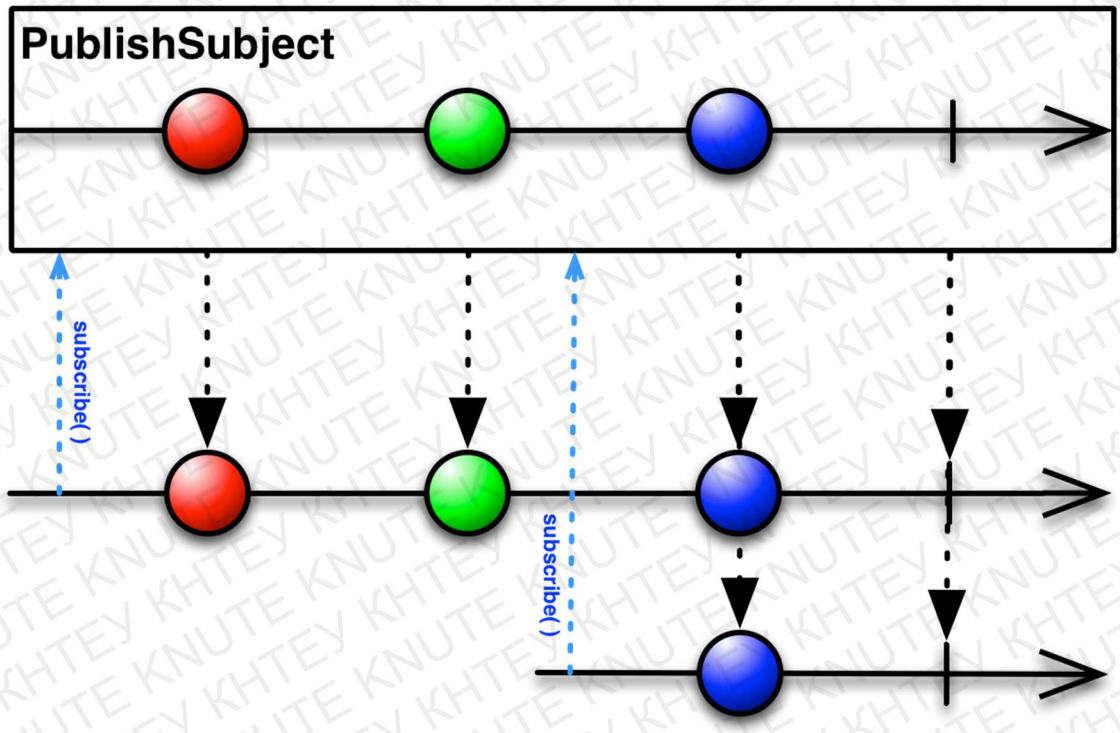


Рис. 2.3.5 Схема роботи PublishSubject

Отже, було розглянуто доволі простий приклад реалізації реактивного програмування на Android, так як, потужність реактивного програмування розкриваєтесь на більш складних задачах з використанням різних типів даних, різних шляхів отримання даних та взаємодію з різними потоками виконання. Але даний приклад значно зменшує кількість коду та логіки в порівнянні з написанням не в реактивному стилі, та значно покращує розуміння та швидкість зміни логіки програми.

Розглянуто основні технології та методи при розробці мобільних додатків на ОС Android. Було зроблено порівняння мов програмування Java та Kotlin.

Розглянуто Firebase Cloud Firestore як гнучку, масштабовану базу даних для мобільних пристроїв, веб-сайтів та серверів Firebase та Google Cloud Platform яка за допомогою прослухувань у реальному часі та офлайн-підтримки буде використана для серверної частини додатку для обміну повідомленнями.

Показано доцільність використання реактивного програмування з використанням RxJava2 для досягнення асинхронності подій в роботі Android додатків.

# Розділ 3. Розробка месенджера

## 3.1 Середовище розробки

Для розробки програми операційної системи Android необхідно використовувати певні середовища програмування. Для розробки під ОС Android від компанії Google є офіційне середовище розробки Android Studio (рис. 3.1.1). Крім офіційної IDE (Integrated Development Environment) є кілька аналогів, не менше потужних і зручних в розробці мобільних додатків.

Android Studio - це відносно нове середовище розробки Андроїд додатків, що базується на платформі Intellij IDEA компанії JetBrains (автори IntellijIdea, PhpStorm, AppCode, ReSharper та язику програмування Kotlin), яка була анонсована на всесвітній конференції Google I \ O 2013.

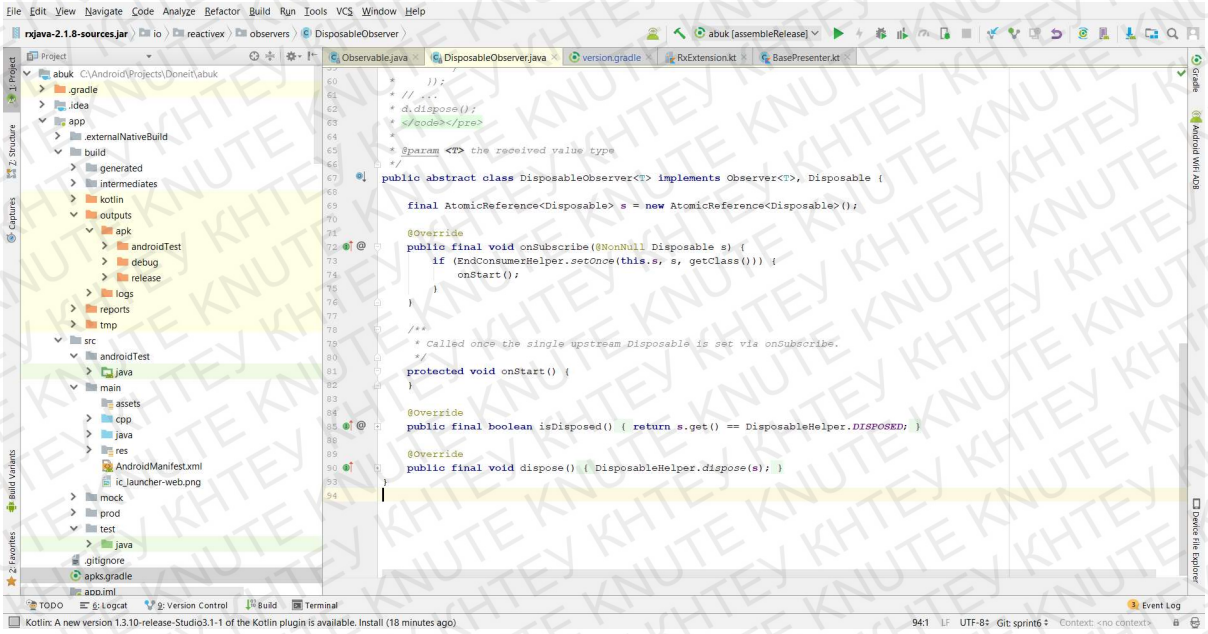

Рис. 3.1.1 Середовище розробки Android Studio.

Особливості та переваги Android Studio:

Нові функції з'являються з кожною новою версією Android Studio. На даний момент доступні наступні функції:

1. Розширений редактор макетів: WYSIWYG, здатність працювати з UI компонентами за допомогою Drag-and-Drop, функція попереднього перегляду макета на декількох конфігураціях екрану.
2. Будову додатків за допомогою Gradle.
3. Різні види збірок і генерація кількох .apk файлів
4. Рефакторинг коду
5. Статичний аналізатор коду (Lint), що дозволяє знаходити проблеми продуктивності, несумісності версій і інше.
6. Вбудований ProGuard і утиліта для підписки додатків.
7. Шаблони основних макетів і компонентів Android.
8. Підтримка розробки додатків для Android Wear і Android TV.
9. Вбудована підтримка Google Cloud Platform, яка включає в себе інтеграцію з сервісами Google Cloud Messaging і Firebase.
10. Вбудовану підтримку мов програмування Kotlin та C++

Іншим варіантом для розробки додатків під ОС Android є Xamarin (рис. 3.1.2). Xamarin - це фреймворк для кроссплатформенної розробки мобіль-них додатків (iOS, Android, Windows Phone) з використанням мови C#.

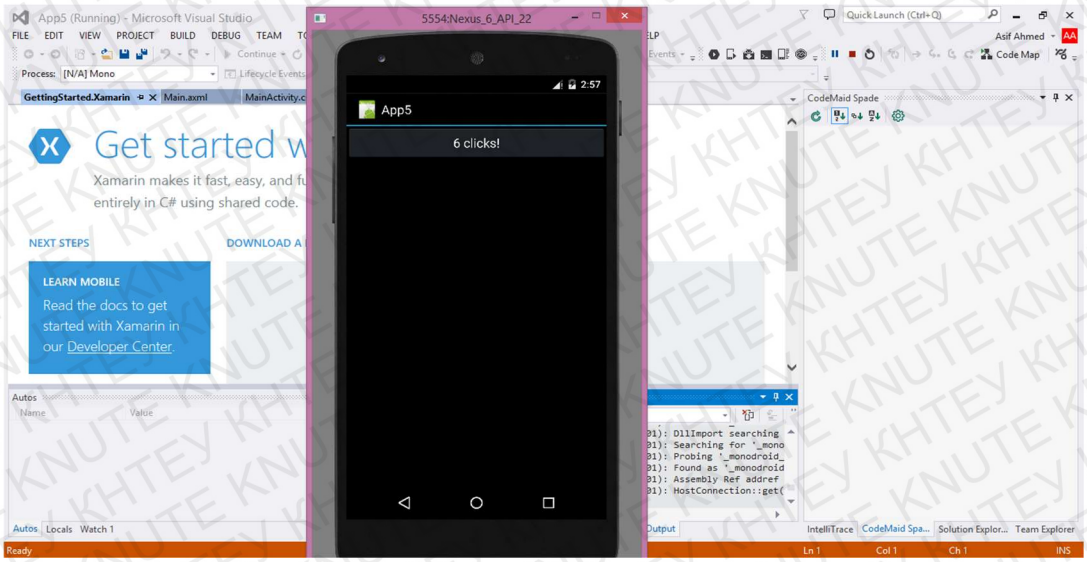


Рис. 3.1.2 Середовище розробки Xamarin.

Фреймворк складається з декількох основних частин:

- Xamarin.IOS - бібліотека класів для C#, що надає розробнику доступ до iOS SDK;
- .Android - бібліотека класів для C#, що надає розробнику доступ до Android SDK;
- Компілятори для iOS і Android;
- IDE Xamarin Studio;
- Плагін для Visual Studio.

Вибір був зроблений на користь офіційної IDE, яку рекомендують програмісти з компанії Google для розробки додатків під ОС Android.

## 3.2 Архітектура додатку

При розробці складних додатків можна зіткнутися з проблемами, які, ймовірно, виникали раніше і вже мають велику кількість рішень. Такі рішення називаються паттернами (шаблонами). Вони спрощують розробку додатків, тому доцільно їх використовувати, якщо така можливість є.

Дизайн проекту повинен бути пріоритетним завданням з самого початку розробки. Одне із завдань, які потрібно вирішити в першу чергу - це архітектура проекту, яка визначає, як різні елементи нашого додатку будуть співвідноситися один з одним.

Існує досить багато підходів для побудови складних систем з хорошою архітектурою. Незважаючи на невеликі відмінності цих підходів, у них багато спільного. Зрозуміло, вони все задають способи розбиття програми на окремі модулі. При цьому в кожній системі як мінімум є модулі, що містять бізнес-логіку програми, і модулі для відображення даних. І кожен підхід в підсумку дозволяє побудувати систему, яка задовольняє наступним принципам:

1. Архітектура повинна бути незалежна від різних фреймворків. Зрозуміло, в сучасному світі ми не можемо обходитися без якихось бібліотек, які дозволяють вирішувати завдання набагато швидше і частіше ефективніше, ніж це зробили б ми в разі самостійної реалізації. Але тут важливо розуміти, що бібліотека повинна вбудовуватися в архітектуру, а не архітектура повинна підлаштовуватися під обрану бібліотеку. При використанні бібліотеки, яка змушує переробляти архітектуру програми, завжди будуть певні обмеженнями цієї бібліотеки і не можна буде перебудовувати архітектуру за потрібним чином. Потрібно використову-вати бібліотеки тільки в якості допоміжних інструментів.

2. Система повинна бути протестована. При цьому потрібно мати можливість тестувати як модулі системи окремо, так і тестувати взаємодію цих модулів між собою і інтеграцію їх в системі. Крім того необхідно тестувати систему без UI, реального сервера і роботи з базою даних, тобто архітектура повинна бути незалежна від оточення.

3. З попередніх пунктів плавно випливають і такі принципи, які говорять про те, що додаток має бути незалежно від усього: від інтерфейсу користувача, від роботи з базою даних, від роботи сервера і від інших елементів оточення. Незалежність архітектури від оточення дуже важлива, так як це

дозволяє змінювати різні компоненти оточення без зміни самої архітектури. Що мається на увазі під зміною компонентів архітектури? Це може бути зміна у виборі бази даних (або ж взагалі відмова від неї) або ж зміна в UI-частини програми (наприклад, потрібно змінити зовнішній вигляд екрану). Якщо для внесення таких змін вимагається зміна архітектури, то, можливо, варто переглянути архітектуру.

Отже, у додатку було обрано архітектуру MVP.

MVP патерн розробки для ОС Android, що пропонує розбивати додаток на наступні частини:

1. Model - є обгорткою для отриманих даних. При цьому особливої різниці звідки дані бути не повинно - дані мережевих запитів або дані взаємодії користувача з призначеним для користувача інтерфейсом (кліки, свайпи і т.д). Гарне місце для впровадження «рукописних» кешей. Доброю практикою є для кожної відповіді сервера створювати унікальну модель, для зменшення перетинів і наступних проблем при змін арі.

2. Presenter - встановлює зв'язок між обробкою даних, одержуваних з Model і викликом методів у View, реалізуючи тим самим реакцію компонентів для користувача інтерфейсу на дані. Методи Presenter викликаються з методів життєвого циклу activity / fragment і часто «симетричні» їм.

3. View - відображає отримані дані з Presentor. У правильній реалізації об'єкт View не має уявлення про отримані дані із зовні, він повинен тільки відображати що вимагає від нього Presenter. View може бути будь-яка Activity або Fragment в додатку під ОС Android.

Model / View / Presenter (рис. 3.2.1) повинні представляти із себе інтерфейси для більшої гнучкості модифікації коду.

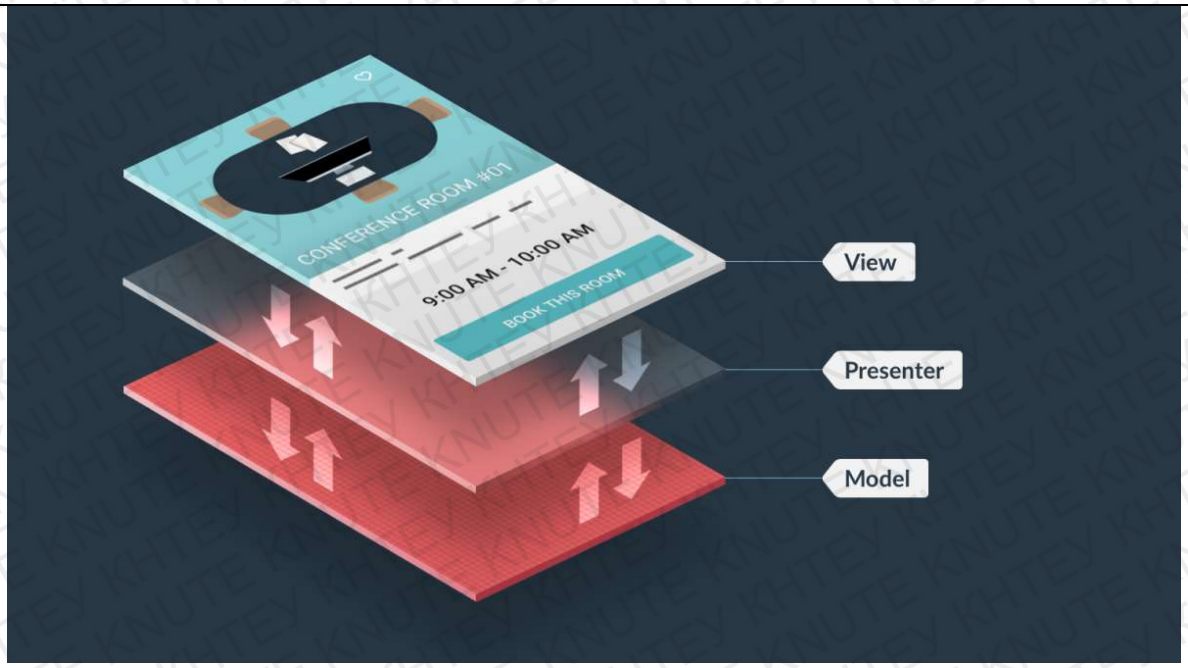


Рис. 3.2.1 Схема взаємодії компонентів в MVP

Дана архітектура має ряд плюсів в порівнянні зі стандартною схемою написання додатка одним з яких є хороше покриття тестами, що важливо для реалізації програми, а також гнучкості. Додаток не знає що відбувається до того як відобразити елемент на екрані, вся логіка закрита від View. Два пункти враховані по створенню гарної архітектури. Так як MVP поділяють логіку та відображення елементів, а також якщо програма має клієнт серверну структуру розділяє і запити до сервера. За допомогою MVP можна також досягти і досить простого і зрозумілого коду, це третій пункт хорошою архітектури додатку. Крім цього MVP робить можна сказати свої власні блоки для кожного вікна, який бачить користувач, це означає що система, додаток, спокійно розширюється без всяких конфліктів.

## 3.3 Опис функціональності

Додаток має 6 функціональних екранів :

1. Авторизація користувача.
2. Редагування профілю користувача або підтвердження реєстрації (якщо профіль тільки що зареєстровано).
3. Список активних діалогів.
4. Список користувачів.
5. Налаштування додатку
6. Відкритий діалог

**Авторизація користувача**. Після встановлення додатку кожен користувач має пройти авторизацію для користування додатком. Коли користувач авторизований він має змогу створювати нові діалоги з іншими користувачами та посилати їм повідомлення. Якщо користувач вже зареєстрований та авторизований в додатку то після входження в додаток користувача буде одразу направлено на екран активних діалогів.

Авторизація в додатку виконується за допомогою Firebase Authentication та має 2 активних способи авторизації:

1. Авторизація за допомогою електронної пошти та паролю – авторизує користувача через email та пароль. Після реєстрації треба підтвердити електронну адресу на яку прийде код авторизації.
2. Авторизація за допомогою облікового запису Google (рис. 3.3.1) – мабуть найшвидший та найпростіший спосіб авторизації. При авторизації через Google додаток робить запит до поточного облікового запису пристрою, а після підтвердження доступу авторизує користувача в додатку.

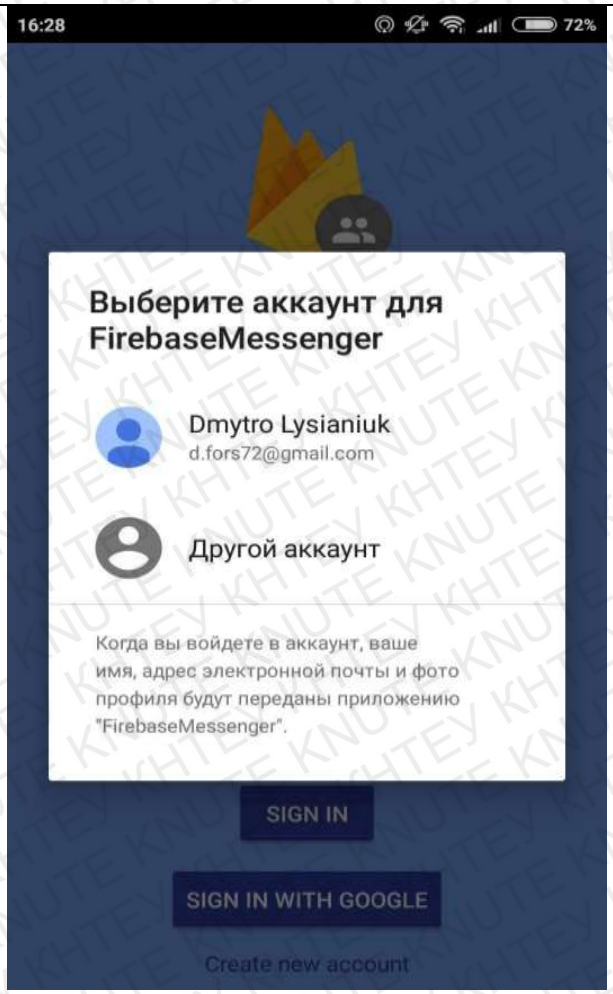


Рис. 3.3.1 Екран авторизації за допомогою облікового запису Google

Також Firebase Authentication має ще такі способи авторизації які не були реалізовані в додатку а саме через номер мобільного телефону, Play Ігри, Facebook, Twitter та GitHub.

Після успішної авторизації користувача буде направлено на екран активних діалогів. Якщо користувач тільки зареєструвався то користувача буде направлено на екран підтвердження реєстрації.

**Редагування профілю користувача або підтвердження реєстрації.** Після реєстрації користувач повинен заповнити данні свого профілю(ім'я та фото). Завантаження фото відбувається за допомогою Firebase Storage та після того як користувач вибрав фото з пристрою, фото буде завантажене на сервер. Після

завантаження додаток отримає url-посилання яке разом з введеним ім'ям буде збережене в базі даних на сервері.

Оскільки ми використовуємо NoSql базу даних на сервері та без використання серверної логіки (тобто запит на створення, зміну, читання, видалення даних відбувається напряму з додатку), з'являється проблема з доступом до даних, тобто кожен користувач буде мати доступ до всіх даних, та зможе читати, змінювати та видаляти данні інших користувачів. Для вирішення цієї проблеми було використано Security Rules (правила безпеки).

Правила безпеки Cloud Firestore дозволяють вам контролювати доступ до документів та колекцій у вашій базі даних. Синтаксис гнучких правил дозволяє створювати правила, які відповідають будь-яким параметрам, від усіх записів до всієї бази даних до операцій за конкретним документом.

Отже для безпечної авторизації та захисту зберігання даних було розроблено такі Security Rules (рис. 3.3.2):

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read: if signedIn();
      allow update,create: if request.auth.uid == userId;
      allow delete: if false;
      match /private/privateInfo {
        allow update,create,read: if request.auth.uid == userId;
        allow delete: if false;
      }
    }
  }
  function signedIn() {
    return request.auth.uid != null;
  }
}
```

Рис. 3.3.2 Security Rules для доступу до користувачів

Данні профілів користувачів зберігаються в колекції users з такими доступами:

На читання даних – для всіх авторизованих користувачів.

На створення – тільки користувач який має той самий ідентифікатор користувача. Тобто користувач може створити документ назва якого співпадає з його ідентифікатором.

На редагування – так само як і на створення але коли вже створено документ в базі. Тобто користувач може редагувати документ назва якого співпадає з його ідентифікатором.

На видалення – жоден користувач не може видаляти данні в даній колекції.

В документі користувача ще є внутрішня колекція private з єдиним документом privateInfo, доступ на читання, створення та редагування є тільки у поточного користувача.

Також даний екран використовується не тільки для підтвердження реєстрації але й для редагування профілю в який можна потрапити з налаштувань додатку.

Після створення документу користувача та успішної підтвердження реєстрації користувача буде направлено на екран активних діалогів.

**Список активних діалогів.** На екрані активних діалогів відображаю-ться всі діалоги в яких даний користувач є учасником. Це можуть бути діалог з конкретним користувачем або груповий діалог де може знаходитись 2 або більше користувачів. Для захисту від редагування користувачами які не є учасниками діалогів були розроблені таки правила безпеки (рис. 3.3.3):

```
service cloud.firestore {

  match /databases/{database}/documents {

    match /dialogs/{dialogId}{

      allow read: if isMemberOfDialog(resource) ||
          !exists(/databases/$(database)/documents/dialogs/$(dialogId));

      allow update: if isMemberOfDialog(resource);

      allow create: if signedIn();

      allow delete: if false;

    }
  }

  function signedIn() {
      return request.auth.uid != null;
  }

  function isMemberOfDialog(dialogDoc) {
      return dialogDoc.data.users[request.auth.uid] != null;
  }
}
```

Рис. 3.3.3 Security Rules для доступу до діалогів

Данні діалогів користувачів зберігаються в колекції dialogs з такими доступами:

На читання даних – тільки користувачі які є учасниками даного діалогу.

На створення – для всіх авторизованих користувачів.

На редагування – так само як і на читання але коли вже створено документ в базі.

На видалення – жоден користувач не може видаляти данні в даній колекції.

Для створення діалогу з конкретним користувачем створюється документ з назвою яка відповідає конкатенації ідентифікаторів обох користувачів в алфавітному порядку, це буде гарантією що ідентифікатор діалогу буде завжди однаковий між двома користувачами і так можна перевірити чи існує діалог між користувачами. Для створення діалогу з групою користувачів створюється документ з унікальною назвою яка згенерована сервером. Ще обов'язковим пунктом створення діалогу є те що треба додати усі ідентифікатори

користувачів в масив документу який буде відповідати полю users щоб кожен доданий користувач мав доступ до створеного діалогу згідно правил доступу діалогів.

**Відкритий діалог.** На екрані відкритого діалогу відображаються всі повідомлення користувачів даного діалогу в хронологічній послідовності. Кожне повідомлення містить в собі – фото та імя відправника, текстове повідомлення та дату створення повідомлення. Для захисту від редагування користувачами які не є учасниками діалогів були розроблені таки правила доступів (рис. 3.3.4):

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /dialogs/{dialogId} /messages/{messageId}{
          allow read,create: if isMemberOfDialog(getDialogDoc(database,dialogId));
          allow update,delete: if false;
      }
  }
  function isMemberOfDialog(dialogDoc) {
      return dialogDoc.data.users[request.auth.uid] != null;
  }
  function getDialogDoc(database,dialogId) {
      return get(/databases/$(database)/documents/dialogs/$(dialogId));
  }
}
```

Рис. 3.3.4 Security Rules для доступу до діалогів

Данні повідомлень діалогу зберігаються в колекції dialogs/{dialogId} /messages з такими доступами:

На читання даних та створення – тільки користувачі які є учасниками даного діалогу.

На редагування та видалення – жоден користувач не може видаляти данні в даній колекції.

Оскільки Cloud Firestore це синхронізована база даних то при створені та збереженні нове повідомлення буде відображено одразу у всіх учасників діалогу без примусового оновлення діалогу.

**Список користувачів.** На екрані списку користувачів відображаються всі зареєстровані користувачі. При переході на даний екран відображаються перші 20 користувачів, при прокручуванні екрану вниз виконуються пагінація та відображаються наступні 20 користувачів

**Налаштування додатку.** На екрані налаштування додатку можна змінити персональні данні авторизованого користувача та виконати вихід з облікового запису користувача.

### 3.4 Тестування додатку

Тестування додатка є невід'ємною частиною процесу розробки додатка. Послідовно виконуючи тести в додатку, можна підтвердити правильність, функціональну поведінку та зручність використання додатка, перш ніж публікувати його.

Тестування також надає такі переваги:

1. Швидкий відгук про помилки.
2. Раннє виявлення несправностей у циклі розробки.
3. Рефакторинг безпечного коду, що дозволяє оптимізувати код, не турбуючись про регресії.
4. Стабільна швидкість розвитку, що допомагає мінімізувати час розробки.

Користувачі взаємодіють із додатком на різних рівнях, натискаючи кнопку "Надіслати", щоб надіслати нове повідомлення, треба впевнитися що з новими змінами в додатку логіка буде відповідати потребам та не створюватиме перешкод в роботі або помилок. Відповідно, потрібно протестувати різні

випадки використання та взаємодії, коли повторно розробляєтеся додаток, наприклад коли відсутній зв'язок з інтернет.

Тестування додатку можна умовно уявити в виді піраміди (рис. 3.2.1). Тестова піраміда (рис. 3.4.1) містить три шари: тести інтерфейсу на вершині, тести на інтеграцію в середині, а також Unit тестування внизу. Коли ви працюєте вниз по піраміді, тести зменшуються в точності, але вони також зменшують час виконання та зусилля для підтримки та налагодження. Тому потрібно написати більше одиничних тестів, ніж інтеграційних тестів, і більше інтеграційних тестів, ніж тестів інтерфейсу.

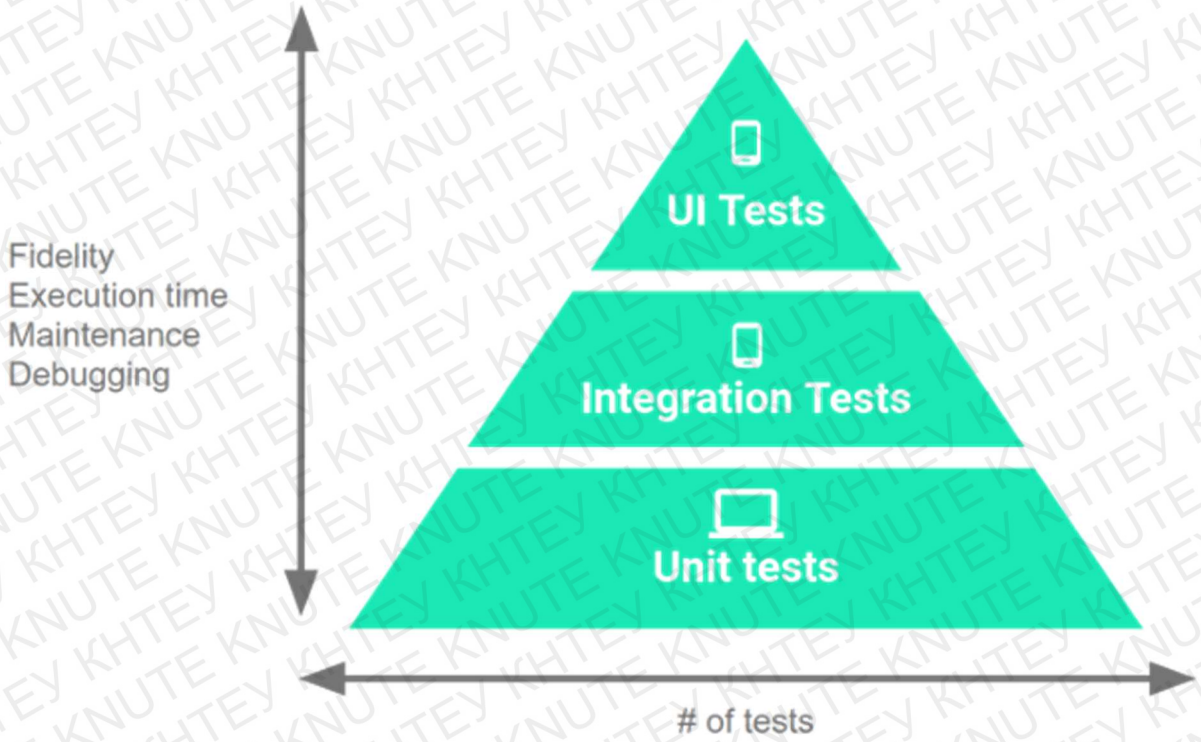


Рис. 3.4.1 Тестова піраміда, яка показує три категорії тестів, які слід включити до набору тестів додатка

Тестова піраміда, показує, як ваш додаток має включати три категорії тестів - малі, середні та великі:

Малі тести - Unit тест, як правило, використовує функціональні можливості найменшої можливої одиниці коду (який може бути методом, класом або компонентом). Потрібно створити тести на одиницю, коли потрібно

перевірити логіку певного коду у додатку. Як правило, одиниця коду перевіряється в ізоляції; тест впливає і контролює зміни тільки цього блоку. Можна використовувати постачальників залежностей, таких як Robolectric, щоб ізолювати пристрій від його залежностей. Unit тести виконуються на локальному комп'ютері тому виконуються швидко так як не потрібно встановлювати їх на пристрій.

Середні тести - це інтеграційні тести, які проходять між невеликими тестами та великими тестами. Вони інтегрують кілька компонентів, і працюють на емуляторах або реальних пристроях. Тестування вже проходить в реальних умовах роботи, тобто вже можна протестувати правильність роботи бази даних або запит на сервер на реальному пристрої.

Великі тести - це інтеграція та тести інтерфейсу, які виконуються, заповнюючи робочий процес інтерфейсу користувача. Вони гарантують, що основні кінцеві користувачі працюють так, як очікувалося, від емуляторів або реальних пристроїв. Тестування інтерфейсу дозволяє вам гарантувати, що ваше додаток відповідає його функціональним вимогам і забезпечує високий рівень якості. Одним з підходів до тестування інтерфейсу користувача є просто тестування людиною, яка виконує набір дій користувача в цільовому додатку та перевіряє, чи правильно він працює. Однак цей ручний підхід може вимагати тривалого часу, втомливого і підданого помилці. Більш ефективний підхід - це написати тести інтерфейсу користувача, щоб дії користувача виконувалися автоматично.

Незважаючи на те, що невеликі тести дозволяють швидко і цілеспрямовані вирішувати помилки, вони також мають низьку точність та автономність, що ускладнює упевненість у тому, що поточний тест дозволяє вашому додатку працювати.

Через різну характеристику кожної категорії тестів, слід включати тести з кожного шару тестової піраміди. Незважаючи на те, що частка тестів для кожної категорії може відрізнятися залежно від випадків використання вашого

додатка, Google пропонуємо поділ серед цих категорій: 70% малих, 20% середніх та 10% великих. Коли ви додаєте та змінюєте функціональність свого додатка, переконайтеся, що ці функції ведуть себе так, як задумано, створюючи та виконуючи тестування з ними. Хоча можна оцінити одиниці на пристрої або емуляторі, зазвичай швидше і простіше тестувати підрозділи у вашому середовищі розробки, додаючи приховані або викривлені методи, необхідні для взаємодії з системою Android.

Якщо середовище тестування додатка потребує тестування пристроїв для більш широкого взаємодії з платформою Android, можна використовувати Robolectric. Цей інструмент виконує справжній код Android у місцевому JVM. Щоб допомогти конструювати структурні об'єкти, які ви використовуєте у своїх тестах, тест AndroidX містить набір класів Android Builder, які можна використовувати в середовищі Robolectric або на пристрої.

Robolectric тести практично відповідають повній точності тестування на пристрої Android, але виконуються швидше, ніж тестування на пристрої.

Окрім підтримки Android API Test API, Robolectric пропонує декілька інших API-інтерфейсів, які AndroidX Test не підтримує.

**Взаємодія з середовищем Android.** Якщо в додатку мінімальна залежність від Android SDK, і потрібно протестувати конкретні взаємодії між компонентом та його залежностями у додатку, можна скористатися схемою “заглушки”, щоб виключити зовнішні залежностей в коді. Таким чином, можна легко перевірити, що компоненти взаємодіють правильно. Підставляючи залежність Android від макета-об'єктів, можна ізолювати тестування блоку від системи Android, перевіряючи, чи називаються правильні методи цих залежностей. За допомогою Mockito ви можете налаштувати фейкові об'єкти, так щоб вони повертали певне значення під час виклику.

Також можна запускати інструментальні тестові пристрої на фізичному пристрої або емуляторі, що не вимагає ніяких заглушок або неповоротності структури. Оскільки така форма тестування передбачає значно повільніший час

виконання, ніж Unit тести, найкраще покладатися на цей метод лише тоді, коли важливо оцінити поведінку додатка щодо фактичного пристрою.

Після того, як перевірили кожну одиницю додатку в середовищі розробки, слід перевірити, чи правильно компоненти поводяться, коли запускаються на емуляторі чи пристрої. Середні тести дозволяють завершити цю частину процесу розробки. Ці тести особливо важливі для створення та запуску, якщо деякі компоненти додатку залежать від фізичного обладнання.

Середні тести оцінюють, як додаток координує декілька одиниць, проте вони не перевіряють додаток повністю. Приклади середніх тестів включають службові тести, інтеграційні тести та герметичні інтерфейсні тести, які імітують поведінку зовнішніх залежностей.

Як правило, краще тестувати програму на емульованому пристрої або на основі хмарної служби, як Firebase Test Lab, а не на фізичному пристрої, тому що можна перевірити декілька комбінацій розмірів екрана та конфігурації апаратного забезпечення легше та швидше.

Хоча важливо тестувати кожен шар і функціональність у додатку окремо, це також важливо для тестування спільних робочих процесів та використання випадків, які включають повний стек, від інтерфейсу користувача через бізнес-логіку до рівня даних.

Якщо додаток досить малий, то може знадобитися лише один набір великих тестів для оцінки функціональності додатка в цілому. В іншому випадку потрібно розділити великі тестові набори.

Для кожного великого тесту на основі потоку, який пишеться, слід також написати середні тести, які перевіряють функціональність кожного компонента інтерфейсу, що входить до робочого процесу. Таким чином, тестовий набір може продовжувати ідентифікувати потенційні проблеми на кожному етапі критичного шляху споживача, навіть якщо відповідний великий тест не працює протягом одного з перших кількох кроків.

Для прикладу в додатку було виконано 4 Unit тести. Спочатку в момент запуску тесту MockitoJUnitRunner виконує метод з налаштуваннями (рис. 3.4.2) який вказано з анотацією @Before. В методі налаштування створюються фейкові об’єкти за допомогою Mochito які надалі будуть використовуватися в тесті.

```
@RunWith(MockitoJUnitRunner::class)
class ExampleUnitTest {

    private lateinit var chatUseCase: ChatUseCase

    private lateinit var chatView: ChatView

    @Before
    fun start() {
        chatUseCase = Mockito.mock(ChatUseCase::class.java)
        chatView = Mockito.mock(ChatView::class.java)
        RxAndroidPlugins.setInitMainThreadSchedulerHandler { t : Callable<Scheduler!> ->
            Schedulers.trampoline()
        }
    }
...
}
```

Рис. 3.4.2 Налаштування тесту

Далі, так як ми використовуймо реактивне програмування та всі запити виконуються в фоновому потоці, а тести повинні завершитися в потоці виконання, то для цього було перевизначено Schedulers для фонового потоку в методі RxAndroidPlugins.setInitMainThreadSchedulerHandler який буде перехоплювати всі спроби виконання операцій в фоновому потоці та продовжувати виконання в поточному потоці.

Кожен тест вказується анотацією @Test та виконується послідовно. Після завершення всіх тестів в класі – результат тестів виводиться в консолі (рис. 3.4.3).

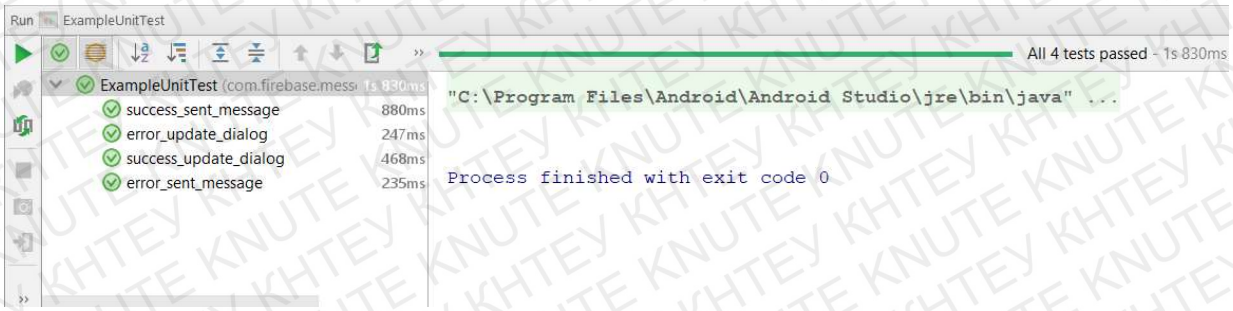


Рис. 3.4.3 Вдалий результат тестів в консолі

В першому тесті error_sent_message виконується перевірка на спроможність додатка обробити помилку при спробі надіслати нове повідомлення та показати повідомлення помилки користувачу (рис. 3.4.4).

```
@Test
fun error_sent_message() {
    Mockito.`when`(chatUseCase.buildUseCaseObservableUnit(any()))
            .thenReturn(Observable.error(Exception("test error")))
    Mockito.`when`(chatUseCase.sentMessage(any(), any())).thenCallRealMethod()
    val chaPresenter = ChatPresenter(chatUseCase)
    chaPresenter.attachView(chatView)
    chaPresenter.sentMessage( dialogId: "testDialog", Message( message: "testMessage"))
    Mockito.verify(chatView).showError(any())
}
```

Рис. 3.4.4 Unit тест на спроможність обробити помилку при спробі надіслати нове повідомлення.

За допомогою метода Mockito.`when` створюється поведінка викликів методів для фейкового об’єкту chatUseCase. В даному випадку при спробі

надіслати повідомлення буде генеруватися Observable який буде викидувати помилку.

Далі в тесті імітується ініціалізація презентеру діалогу та спроба надіслати тестове повідомлення, так як спроба надіслати повідомлення закінчиться помилкою метод Mockito.verify(chatView).showError(any()) перевіряє що повідомлення про помилку було показано користувачу.

В другому тесті success_sent_message (рис. 3.4.5) імітується ситуація коли повідомлення буде надіслано вдало, та тест виконає перевірку на те що оновлені данні були показані користувачу.

```
@Test
fun success_sent_message() {
    Mockito.`when`(chatUseCase.buildUseCaseObservableUnit(any()))
            .thenReturn(Observable.just(Unit))
    Mockito.`when`(chatUseCase.sentMessage(any(), any())).thenCallRealMethod()
    val chaPresenter = ChatPresenter(chatUseCase)
    chaPresenter.attachView(chatView)
    chaPresenter.sentMessage( dialogId: "testDialog", Message( message: "testMessage"))
    Mockito.verify(chatView).renderDialog(any())
}
```

Рис. 3.4.5 Unit тест на те що при надісланні повідомлення дані будуть показані користувачу.

В тесті error_update_dialog (рис. 3.4.6) виконується перевірка на спроможність додатка обробити помилку при спробі оновити повідомлення та показати повідомлення помилки користувачу.

```
@Test
fun error_update_dialog() {
    Mockito.`when`(chatUseCase.buildUseCaseObservable(any()))
            .thenReturn(Observable.error(Exception("test error")))
    Mockito.`when`(chatUseCase.getMessages(any(), any())).thenCallRealMethod()
    val chaPresenter = ChatPresenter(chatUseCase)
    chaPresenter.attachView(chatView)
    chaPresenter.initialize( dialogId: "testDialog", limit: 40)
    Mockito.verify(chatView).showError(any())
}
```

Рис. 3.4.6 Unit тест на те що при надісланні повідомлення дані будуть показані користувачу.

В тесті success_update_dialog (рис. 3.4.7) імітується ситуація коли повідомлення були оновленні вдало, та тест виконає перевірку на те що оновлені данні були показані користувачу.

```
@Test
fun success_update_dialog() {
    Mockito.`when`(chatUseCase.buildUseCaseObservable(any()))
            .thenReturn(Observable.just(Mockito.mock(QuerySnapshot::class.java)))
    Mockito.`when`(chatUseCase.getMessages(any(), any())).thenCallRealMethod()
    val chaPresenter = ChatPresenter(chatUseCase)
    chaPresenter.attachView(chatView)
    chaPresenter.initialize( dialogId: "testDialog", limit: 40)
    Mockito.verify(chatView).renderDialog(any())
}
```

Рис. 3.4.7 Unit тест на те що дані будуть показані користувачу.

Отже в результаті тестування всі тести було виконано з відмінним результатом.

## 3.5 Публікація додатку в Google Play

Після розробки та тестування додаток потрібно опублікувати. Для розповсюдження додатку було обрано Google Play.

Google Play - це потужна платформа видавців, яка допомагає рекламувати, продавати та розповсюджувати свої додатки для Android користувачам усього світу. Коли ви випускаєте свої програми через Google Play, ви маєте доступ до набору інструментів розробника, які дозволяють аналізувати ваші продажі, визначати ринкові тенденції та керувати тим, хто розповсюджує ваші програми. Ви також маєте доступ до кількох функцій, що підвищують прибутки, таких як виставлення рахунків у додатку та ліцензування програм. Багатий набір інструментів і функцій разом із численними функціями спільноти кінцевих користувачів робить Google Play платформу для продажу та придбання додатків для Android.

Для перевірки додатку перед публікацію в першу чергу треба підібрати новій applicationId який буде ідентифікатором додатку в Google Play. Для перевірки на доступність ідентифікатору можна просто перейти по посиланню https://play.google.com/store/apps/details?id={applicationId}, де applicationId це новий ідентифікатор який буде використовуватися. Якщо сторінки з таким ідентифікатором немає то його можна використовувати для додатку.

Маніфест додатку (рис. 3.5.1) також має містити в собі атрибути назви(label) та іконки(icon) додатку.

Версії також є важливим компонентом стратегії оновлення та підтримки свого додатка.

Користувачам потрібно мати конкретну інформацію про версію додатка, встановлену на своїх пристроях, щоб мати можливість оновити додаток з версією вище.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.firebase.messenger">
    <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:name=".MessengerApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="FirebaseMessenger"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
...
    </application>
</manifest>
```

Рис. 3.5.1 Маніфест додатку

Сервісам через які публікуються додатки, також може знадобитися версія додатку, щоб вони могли відображати версію для користувачів та щоб визначити сумісність та встановити відносини для оновлення.

Система Android підтримує сумісність версій системи, як це виражається в налаштуваннях minSdkVersion у файлах збірки. Цей параметр дозволяє додатку вказати мінімальний системний API, який він сумісний. Тобто версії Android нижче ніж minSdkVersion не будуть підтримуватись додатком.

При розробці версія додатку, applicationId, minSdkVersion та targetSdkVersion вказується файлі build.gradle (рис. 3.5.3).

```
android {
    ...
    compileSdkVersion 27
    defaultConfig {
        applicationId "com.firebase.messenger"
        minSdkVersion 17
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        multiDexEnabled true
        vectorDrawables.useSupportLibrary = true
    }
    ...
}
```

Рис. 3.5.3 Версії додатку в build.gradle

Коли все вже готове, додаток для Android має бути підписаний сертифікатом, завдяки якому можна ідентифікувати автора програми. Коли додаток тестується, встановлюючись через Android Studio на пристрій додаток підписується автоматично. Але для створення реліз-версії треба зробити додатково ряд дій.

При створенні сертифіката слід пам'ятати, що при оновленні програми система буде порівнювати сертифікати старої і нової версії. І оновлення відбуватиметься, якщо сертифікати обох версій співпадають. Але якщо нова версія буде підписана новим сертифікатом, то додаток буде розцінюватися як абсолютно новий, ніяк не пов'язаний зі старою версією і представляє абсолютно іншу програму. У цьому випадку щоб його встановити, нова версія повинна буде мати інший назву пакета, ніж стара.

Для створення сертифікату в Android Studio виберемо в меню пункт Build - Generate Signed APK. Після цього нам відкриється вікно (рис.3.5.4):

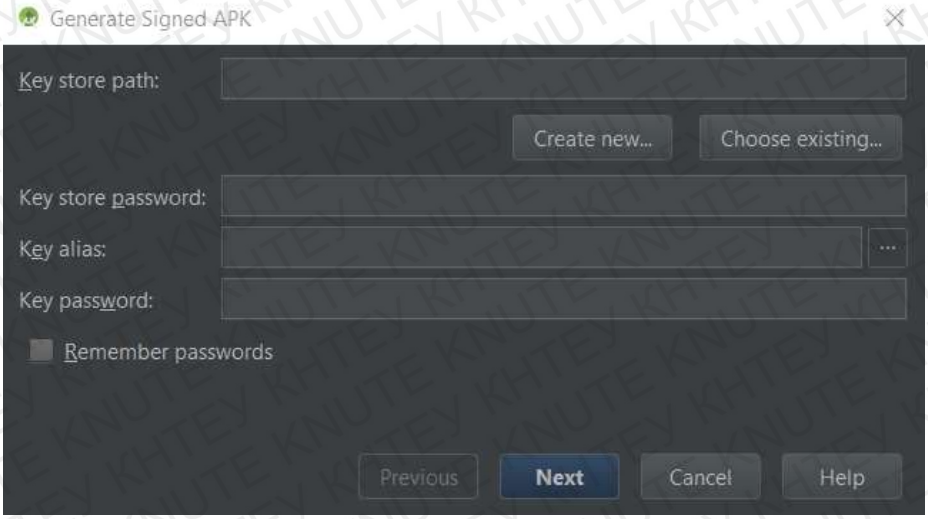


Рис. 3.5.4 Вікно підпису APK

Натиснемо на кнопку Create new. Після цього нам відкриється вікно створення ключа (рис. 3.5.5):

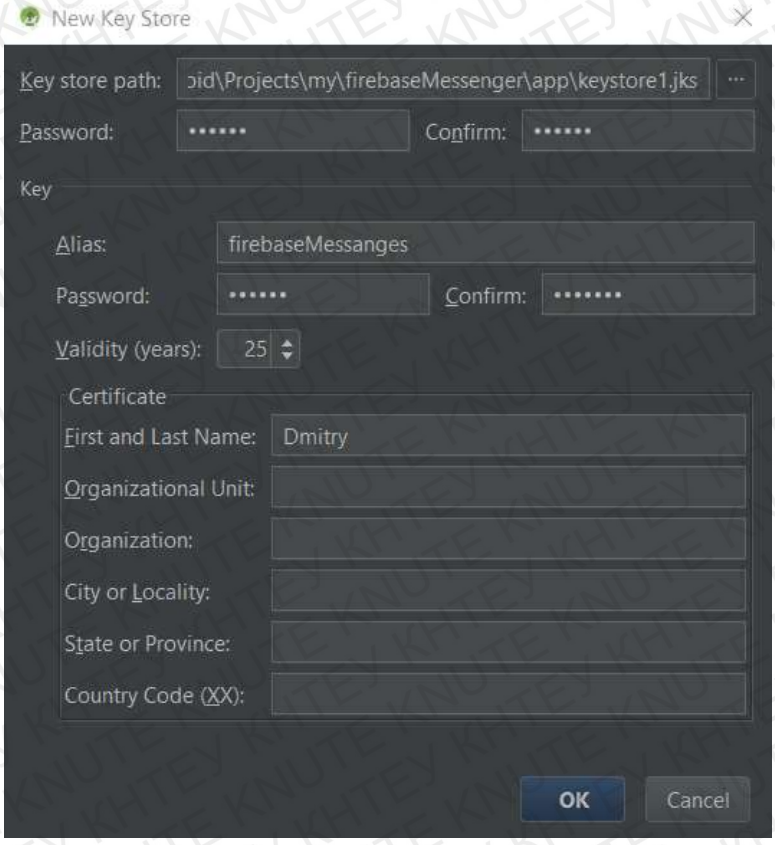


Рис. 3.5.5 Створення сертифікату для додатку

Введемо в поле Key store path шлях до файлу сертифіката, який буде створений. Якщо зазначеної папки не існує, то її треба створити або визначити існуючу папку.

В поле Password / Confirm вказуємо пароль.

В поле Alias вказуємо псевдонім. Можна поставити довільну назву.

В поле First and Last Name вписуємо ім'я і прізвище. І далі заповнюємо підрозділ, організацію, місто, країну і код країни. В кінці натискаємо OK.

Після створення сертифіката можна налаштувати в build.gradle щоб при кожному білді релізної версії додатку, додаток підписувався нашим створеним сертифікатом.

```
android {
    signingConfigs {
        release {
            keyAlias '123456'
            keyPassword '123456'
            storeFile file('keystore.jks')
            storePassword '123456'
        }
    }
    ...
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
            signingConfig signingConfigs.release
        }
    }
}
```

Рис. 3.5.6 Налаштування build.gradle для підпису додатку релізним сертифікатом

Після білду підписаного нашим сертифікатом додатку, його можна починати публікувати на Google Play. Для реєстрації облікового запису розробника в Google Play потрібно заплатити реєстраційний збір 25$ (для

прикладу в магазині додатків App Store для iOS це 99$ на рік) та після створення нового облікового запису розробника можна публікувати додатки в необмеженій кількості.

Для створення нового додатку потрібно в Google Play Console натиснути на «Створити додаток» (рис. 3.5.7).:

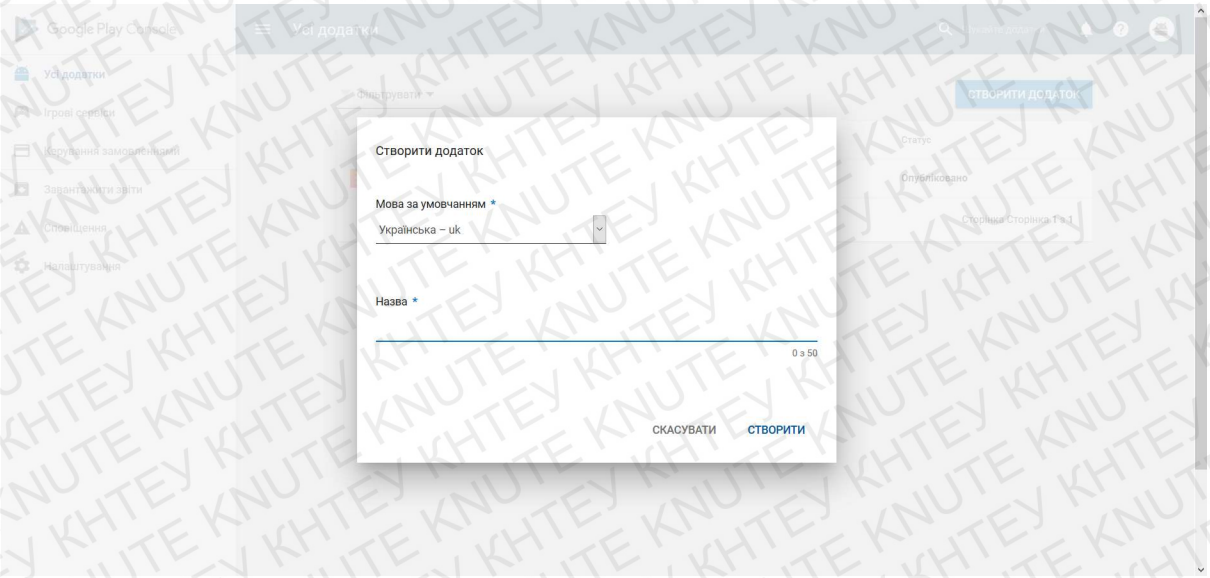


Рис. 3.5.7 Створення додатку в Google Play

Після створення нового додатку можна завантажити APK файл (рис. 3.5.8).

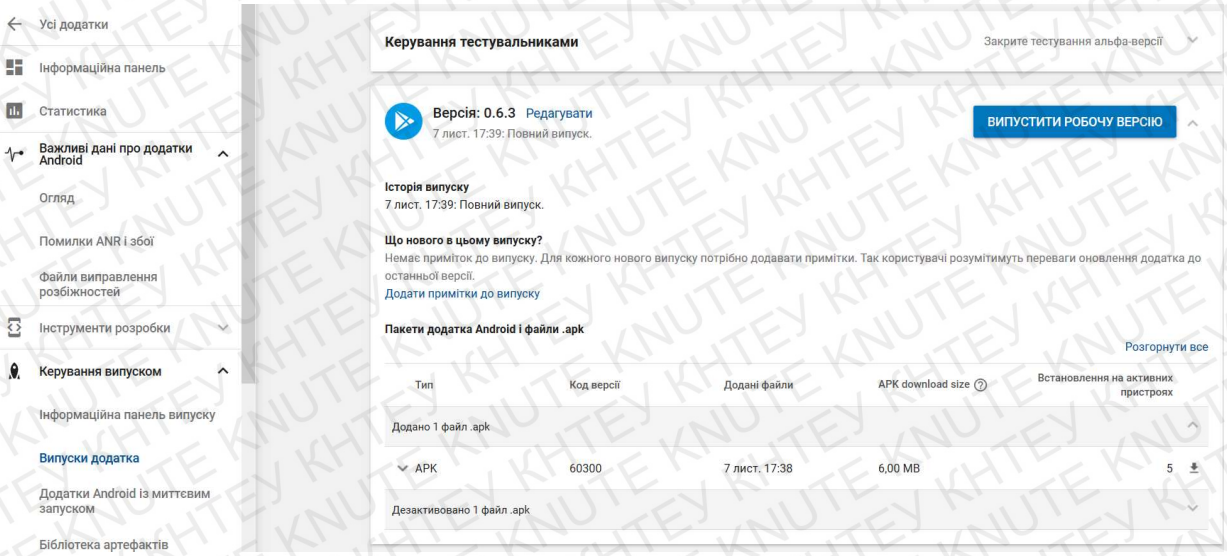


Рис. 3.5.8 Завантаження APK в Google Play

Існує три види версій додатку – Робоча версія, Відкрита версія(або бета версія) та закрита версія(або альфа версія).

Закрита версія дозволяє протестувати версію програми в групі тестувальників. Після того як додаток буде протестовано в невеликій групі співробітників або довірених користувачів, можна випускати версію для відкритого тестування. На сторінці Версії додатка альфа-версія буде доступна як вихідна закрита версія. При необхідності можна створити додаткові закриті версії і присвоїти їм назви. При тестуванні опубліковано-го раніше додатки оновлення його закритої версії отримають лише тестувальники.

Також в закритій версії можна протестувати покупки в додатку яку будуть оплачуватися з тестового віртуального рахунку.

Відкрита версія дозволяє перевірити тестову версію додатка в великій групі і опублікувати її в Google Play. Будь-який користувач може приєднатися до відкритого тестування і надіслати приватні відгуки розробнику.

Робоча версія це повноцінна версія додатку. Будь який користувач може встановити та повноцінно використовувати додаток по призначенню, також може залишати відгук та оцінити встановлений додаток.

Але тільки завантаження файлу APK не дасть можливості опублікувати додаток, також потрібно підготувати основну інформацію яка буде відображатися на сторінці опублікованого додатку.

Для публікації потрібно заповнити назву, короткий опис та повний опис додатку (рис. 3.5.9).

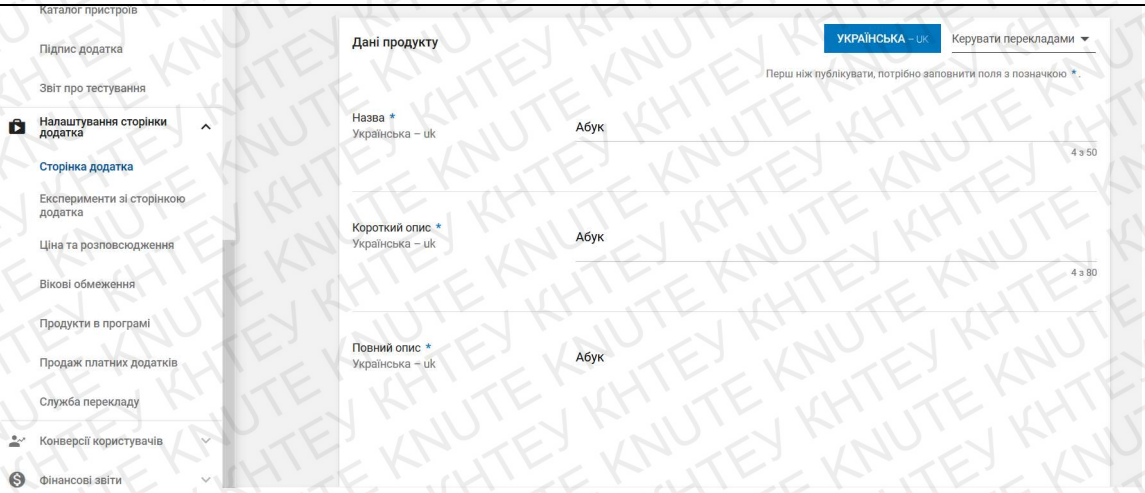


Рис. 3.5.9 Опис додатку в Google Play

Також потрібна іконка додатку, не менше 2х знімків екрану та зображення для опису (рис. 3.5.10).

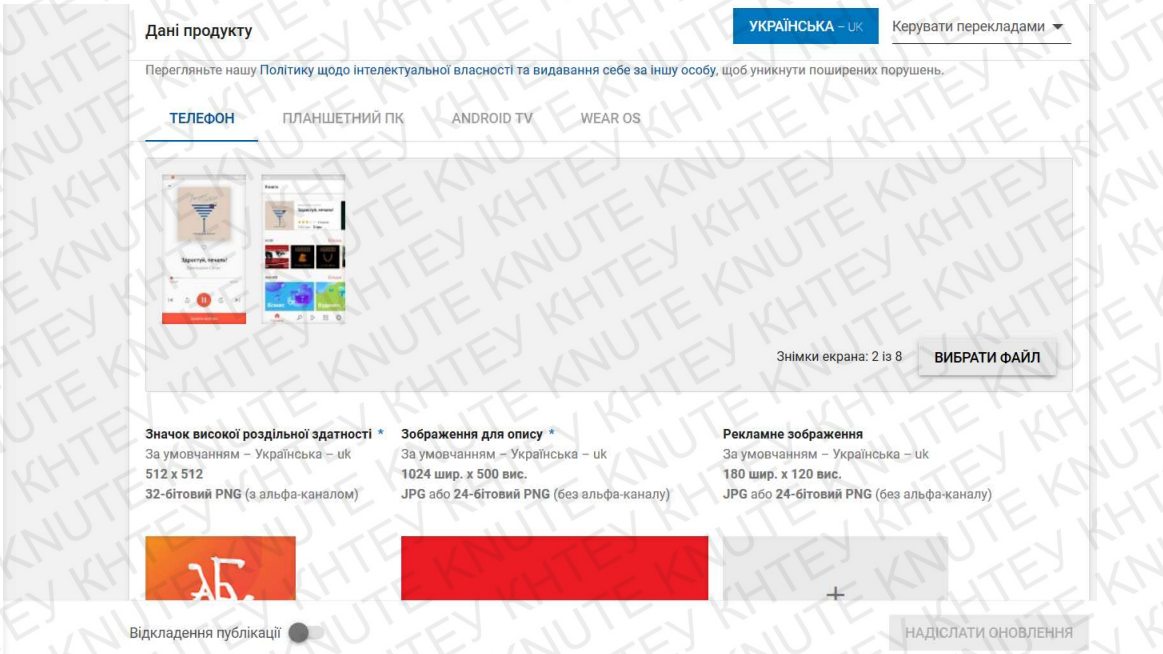


Рис. 3.5.10 Матеріали для публікації в Google Play

Далі потрібно заповнити данні про тип програми, категорію та вікові обмеження якій відповідає додаток (рис. 3.5.11).


Рис. 3.5.11 Тип додатку та вікові обмеження

Політика конфіденційності є також важливою частиною публікації. До даних користувача відносяться відомості, надані їм, а також отримані в ході його роботи, в тому числі інформація про пристрій. Потрібно обов'язково повідомити, як і для чого будуть збиратися і використовуватися дані, а також відкривати до них доступ. Використовувати дані іншим чином або без згоди користувача заборонено. Якщо додаток обробляє особисті або конфіденційні дані, він повинен відповідати вимогам, перерахованим в розділі "Особиста і конфіденційна інформація". Ці умови Google Play доповнюють вимоги чинного законодавства про конфіденційність і захист даних.

Після заповнення всіх необхідних даних додаток можна опублікувати. Перед публікацією додатки модеруються автоматично чи інколи навіть в ручну на предмет вірусів або шкідливого програмного забезпечення яке не відповідає умовам Google Play.

## 3.6 Можливі варіанти розвитку додатку

Після розробки та публікації додатку користувачі зможуть користуватися ним, але популярність додатку навряд буде великою, так як зараз на ринку існує велика кількість популярних додатків обміну повідомленнями, які мають значно ширший функціонал та значно кращий дизайн, обслуговування та ін. Так як метою дипломної роботи було створення простої реалізації додатку та використання популярних технологій при розробці під Android, а розробка більш функціонального додатку, який був би спроможний конкурувати на ринку, займала би значно більше часу(від 4-5місяців). Отже можливими варіантами розвитку додатку можуть бути:

1. Прив'язка облікового запису до номеру телефону. На даний момент використовується два методи авторизації – це через обліковий запис Google та через email з паролем. Також можна додати авторизацію через номер мобільного телефону, на номер якого буде приходити код підтвердження реєстрації. Цей процес також можна автоматизувати якщо перед реєстрацією за номером телефону запросити у користувача дозвіл на читання повідомлень, після того як код підтвердження буде отримано через SMS додаток перехопить повідомлення та закінчить процес реєстрації автоматично. Без автоматичного підтвердження користувач повинен буде пісня отримання коду підтвердження перейти до додатку з повідомленнями, запам'ятати код або скопіювати його та перейти назад в додаток та вписати його в спеціальне поле з підтвердженням, також повідомлення з кодом треба буде видалити щоб воно не засмічувало повідомлення.

2. Контакти користувача – де можна зберігати всі контакти користувачів з якими ви будете спілкуватися. Це зручний спосіб щоб знайти потрібного користувача якщо діалогу з ним ще немає.

3. Push-сповіщення – які будуть з'являтися в сповіщеннях телефону навіть коли додаток вимкнуто. В Firebase Push-сповіщення можна реалізувати за

допомогою продукту Firebase Cloud Messaging який може надсилати сповіщення, які відображаються для вашого користувача. Або надсилайте повідомлення даних які будуть оброблятися в додатку. Якщо додаток увімкнуто то сповіщення буде направлено в сервіс Firebase конкретного додатку, але якщо додаток знаходиться в фоновому режимі то сповіщення буде направлено в систему пристрою(System tray) де сповіщення буде створено системою (рис. 3.6.1).

| App state | Notification | Data | Both |
|---|---|---|---|
| Foreground | onMessageReceived | onMessageReceived | onMessageReceived |
| Background | System tray | onMessageReceived | Notification: system tray<br>Data: in extras of the intent. |

Рис. 3.6.1 Отримання сповіщень з Firebase Cloud Messaging

Але так як ми використовуємо Firebase Firestore то ця база даних ніяк не пов’язана з Firebase Cloud Messaging, отже при створені нового повідомлення сповіщення не буде відправлятися на пристрій, але щоб зв’язати ці продукти можна використати ще один продукт Cloud Functions. Cloud Functions це хмарні функції для Firebase які дозволяють автоматично запускати бекенд-код у відповідь на події, викликані функціями Firebase та запитами HTTPS. Код зберігається в хмарі Google і працює в керованому середовищі. Не потрібно керувати та масштабувати власні сервери.

В нашому випадку після створення нового повідомлення в базу даних, Firebase Firestore буде сповіщавати хмарну функцію про те що додано нове повідомлення, далі функція виконає код котрий відправить це повідомлення на пристрій (рис. 3.6.2).

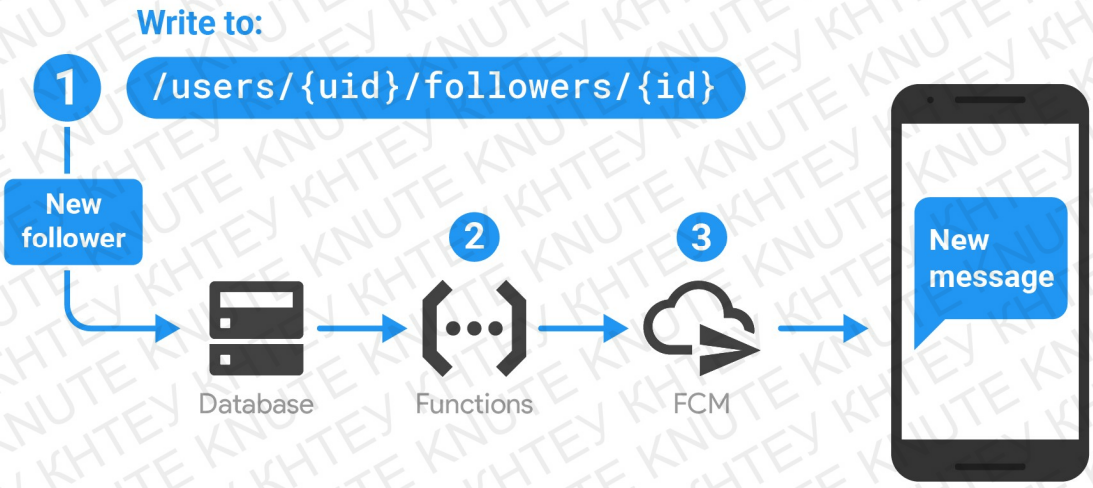


Рис. 3.6.2 Схема створення нового повідомлення за допомогою Cloud Functions.

Відправлення сповіщення можна реалізувати за допомогою запиту (рис. 3.6.3):

```
https://fcm.googleapis.com/fcm/send
Content-Type:application/json
Authorization:key=AIzaSyZ-1u...0GBYzPu7Udno5aA
{
  "to": "/topics/foo-bar",
  "data": {
    "message": "This is a Firebase Cloud Messaging Topic Message!",
    }
}
```

Рис. 3.6.3 HTTPS запит на створення сповіщення

Для Cloud Functions буде потрібно середовище Node.js для написання функцій, і CLI Firebase (який також вимагає Node.js і npm) для розгортання функцій під час виконання Cloud-функцій.

4. Можливість відправляти фотографії, аудіозаписи, відеозаписи та інші файли. Перед відправленням повідомлення файли будуть завантажуватися на Firebase Storage, та разом з текстовим повідомленням буде записуватися інформація про вложення та url-посилання на сам файл. Після отримання такого повідомлення користувач зможе переглянути файл або завантажити його.

5. Можливість надсилати відео або аудіо повідомлення. Основною відмінністю від відправлення аудіо або відео файлу тим що відео або аудіо повідомлення буде записано в момент відправлення за допомогою мікрофону або камери, та їх можна буде одразу прослухати та переглянути. Для запису та відправленню аудіо повідомлення додаток повинен запросити дозвіл на використання мікрофону, оскільки дозвіл на використання мікрофону вважається «небезпечним» то користувач власноруч повинен підтвердити дозвіл (рис. 3.6.4).

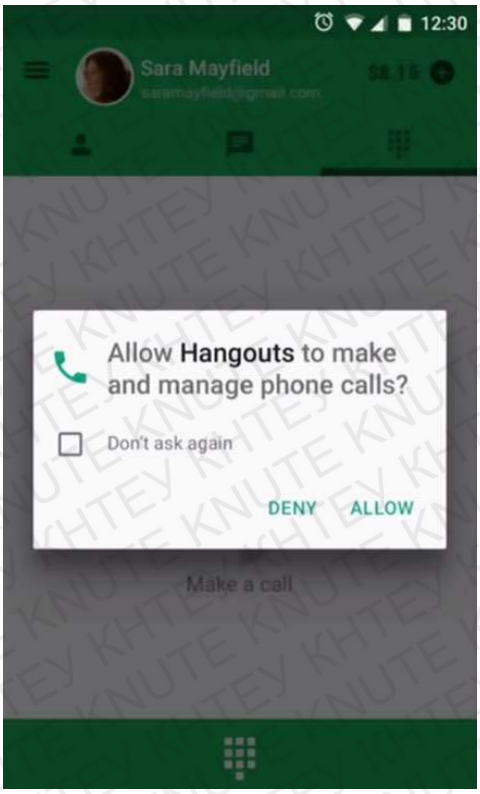


Рис. 3.6.4 Запит на дозвіл використання дзвінків додатком

Також на ряду небезпечних дозволів є дозволи на доступ до календару, камери, контактів, місцезнаходження, телефонних дзвінків, смс та внутрішнього сховища пристрою.

Для відео повідомлення також буде потрібний доступ до камери та мікрофону. Одразу після створення, повідомлення буде завантажене на Firebase Storage та відправиться в базу даних окремим типом повідомлення.

7. Дзвінки між користувачами. Такий функціонал вже значно складніший, та потребує потокового зв'язку між користувачами, який буде зв'язувати користувачів за допомогою окремого серверу.

Отже, було розроблено мобільний додаток на базі операційної системи Android, що дозволяє в режимі реального часу здійснювати передачу текстових повідомлень між користувачами додатку.

Розроблено реалізацію серверної частини додатку за допомогою Firebase Firestore для зберігання та синхронізації повідомлень між користувачами, створено правила доступу для безпечного зберігання даних між користувачами. Також в додаток інтегровано інші продукти Firebase такі як Firebase Authentication, Firebase Cloud Storage. В ході розробки серверної частини було виявлено що Firebase Firestore не підходить для дуже складних структур даних, так як має ряд недоліків в порівнянні з SQL базою даних.

Також протестовано додаток Unit тестами за допомогою яких в подальшій розробці додатку буде легше виявити помилки або невідповідність з завданням додатку, та опубліковано додаток в магазин Google Play.

# ВИСНОВОК

В даній випускній кваліфікаційній роботі було розроблено мобільний додаток на базі операційної системи Android, що дозволяє в режимі реального часу здійснювати передачу текстових повідомлень між користувачами додатку.

Розглянуто основні технології та методи при розробці мобільних додатків на ОС Android. Було зроблено порівняння мов програмування Java та Kotlin, реалізовано архітектуру MVP.

Показано доцільність використання реактивного програмування з використанням RxJava2 для досягнення асинхронності подій в роботі Android додатків.

Розроблено реалізацію серверної частини додатку за допомогою Firebase Firestore для зберігання та синхронізації повідомлень між користувачами, створено правила доступу для безпечного зберігання даних між користувачами. Також в додаток інтегровано інші продукти Firebase такі як Firebase Authentication, Firebase Cloud Storage. В ході розробки серверної частини було виявлено що Firebase Firestore не підходить для дуже складних структур даних, так як має ряд недоліків в порівнянні з SQL базою даних.

Також протестовано додаток Unit тестами за допомогою яких в подальшій розробці додатку буде легше виявити помилки або невідповідність з завданням додатку, та опубліковано додаток в магазин Google Play.

Всі цілі та завдання випускної кваліфікаційної роботи були виконані. По завершенню кваліфікаційної роботи було представлено додаток з можливістю авторизації за допомогою електронної пошти та паролю або обліковим записом Google, а також можливістю надіслання та перегляду повідомлень іншим зареєстрованим користувачам.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Билл Ф. Android. Программирование для профессионалов/ Ф. Билл. – Питер. – 2016 – 640с.
2. Гриффитс Д. Head First. Программирование для Android / Гриффитс Д., Гриффитс Д. – Питер – 2015 – 704с.
3. Жемеров Д. Kotlin in Action/ Жемеров Д., Исакова С. – ДМК Пресс. – 2017 – 402с.
4. Коматинени С. Android 4 для профессионалов. Создание приложений для планшетных компьютеров и смартфонов. / Коматинени С., Маклин Д. – М.:Вильямс. – 2014 – 880с.
5. Коматинэни С. Google Android: программирование для мобильных устройств. / Коматинэни С., Маклин Д., Хэшими С. – СПб.: Питер – 2011 – 736с.
6. Нуркевич Т. Reactive Programming with RxJava Creating Asynchronous, Event-Based Applications. / Нуркевич Т., Кристенсен Б. – O'Reilly Media. – 2016 – 372с.
7. Роджерс Р. Android. Разработка приложений. / Роджерс Р., Ломбардо Д. – ЭКОМ Паблишерз – 2010 – 400с.
8. Смит К. XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies 1st Edition / Смит К. Сэйнт-Андре П. – O'Reilly Media – 2009 – 310с.
9. Фелкер Д. Android Application Development For Dummies. / Фелкер Д. – Диалектика-Вильямс – 2011 – 336с.
10. Шилдт Г. Java 8 / Шилдт Г. – Диалектика-Вильямс. – 2018 – 720с.

11. Ashok K. Mastering Firebase for Android Development / Ashok K. – Packt Publishing – 2018 – 394c.
12. Yahiaoui H. Firebase Cookbook / Yahiaoui H. – Packt Publishing – 2017 – 367c
13. Android Application Publishing [Електронний ресурс]. – Режим доступу: https://developer.android.com/studio/publish/preparing
14. Android Reactive Programming [Електронний ресурс]. – Режим доступу: https://www.androidhive.info/RxJava/android-getting-started-with-reactive-programming/
15. Bauman National Library [Електронний ресурс]. – Режим доступу https://ru.bmstu.wiki/Android_(%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B)
16. Documentation for app developers [Електронний ресурс]. – Режим доступу: https://developer.android.com/docs/
17. Firebase Documentation [Електронний ресурс]. – Режим доступу: https://firebase.google.com/docs/
18. Kotlin Programming Language [Електронний ресурс]. – Режим доступу: https://kotlinlang.org/
19. Kotlin Programming Language for Developing Android Apps [Електронний ресурс]. – Режим доступу: https://www.netguru.co/blog/pros-and-cons-of-the-kotlin-programming-language-for-developing-android-apps-
20. ReactiveX [Електронний ресурс]. – Режим доступу: http://reactivex.io/documentation/observable.html

# ДОДАТКИ

*Додаток А*

```
open class ChatUseCase @Inject constructor() :
UseCase<QuerySnapshot, ChatUseCase.ChatParams>() {

 private val updateSubject: Lazy<PublishSubject<Unit>> = lazy {
PublishSubject.create<Unit>() }

 override fun buildUseCaseObservableUnit(params: ChatParams):
Observable<Unit> {
  val dialogId = params.dialogId
  val message = params.message!!
 return SentMessage(dialogId, message)
         .onErrorResumeNext(CreateUserDialog(dialogId, message)
                 .flatMap { SentMessage(dialogId, message) }
                 .doOnNext {
                   if (updateSubject.isInitialized())
                       updateSubject.value.onNext(Unit) })
                 .flatMap {
                     UpdateDialog(dialogId, message)
                             .onErrorReturn { Unit }
                 }
    }
 override fun buildUseCaseObservable(params: ChatParams):
Observable<QuerySnapshot> {
    return FSObservable(FirebaseFirestore.getInstance()
                 .collection("${params.dialogId}/messages")
                 .orderBy("time", Query.Direction.DESCENDING)
                 .limit(params.limit.toLong())).cache()
                 .retryWhen { t ->
                     t.flatMap { _ -> updateSubject.value }
  }
   fun getMessages(dialogId: String, observer:
```

```
DefaultObserver<QuerySnapshot>, limit: Int = 40) {
     execute(ChatParams(1, dialogId, limit = limit), observer)
    }
    fun sentMessage(dialogId: String, message: Message) {
        executeUnit(ChatParams(2, dialogId, message))
    }
    data class ChatParams(val type: Int,
                          val dialogId: String,
                          val message: Message? = null,
                          val limit: Int = 0)
}
```

*Додаток Б*

```
public class ChatUseCaseJava extends UseCase<QuerySnapshot,
ChatParams> {
    private PublishSubject<Void> updateSubject;
    private PublishSubject<Void> getLazyUpdateSubject() {
        if (updateSubject == null) {
            updateSubject = PublishSubject.create();
        }
        return updateSubject;
    }
    @Inject
    public ChatUseCaseJava() {
    }
    @NotNull
    @Override
public Observable<QuerySnapshot> bUseCaseOb(ChatParams params){
  return new FsSObservable(FirebaseFirestore.getInstance()
                .collection("${params.dialogId}/messages")
                .orderBy("time", Query.Direction.DESCENDING)
                .limit((long) params.getLimit()))
                .retryWhen(new Function<Observable<Throwable>,
ObservableSource<?>>() {
```

```
        @Override
        public ObservableSource<?> apply(Observable<Throwable>
tObservable) throws Exception {
     return throwableObservable.flatMap(new Function<Throwable,
ObservableSource<?>>() {
          @Override
          public ObservableSource<?> apply(Throwable throwable)
throws Exception {
          return getLazyUpdateSubject();
            }
       });
     }
  });
 }
  @NotNull
  @Override
  public Observable<Unit> buildUseCaseObservableUnit(ChatParams
params) {
   final String dialogId = params.getDialogId();
   final Message message = params.getMessage();
   return new SentMessage(dialogId, message)
     .onErrorResumeNext(new CreateUserDialog(dialogId, message)
       .flatMap(new Function<Unit, ObservableSource<?>>() {
          @Override
          public ObservableSource<?> apply(Unit unit) {
            return new SentMessage(dialogId, message);
             }
          }).doOnNext(new Consumer<Unit>() {
            @Override
            public void accept(Unit unit) throws Exception {
              if (updateSubject != null) {
               getLazyUpdateSubject().onNext(Unit.INSTANCE);
                                  }
             }
```

```
            }))
.flatMap(new Function<Unit, ObservableSource<?>>() {
 @Override
 public ObservableSource<?> apply(Unit unit) throws Exception {
  return new UpdateDialog(dialogId, message)
                .onErrorReturn(new Function<Throwable, Unit>() {
                    @Override
                    public Unit apply(Throwable throwable) {
                                return Unit.INSTANCE;
                    }
                });
            }
        });
    }
 public void getMessages(String dialogId, DO<QS> observer) {
        getMess(dialogId, observer, 40);
    }
void getMess(String dialogId, DO<QyS> observer, int limit) {
    execute(new ChatUseCaseJava.ChatParams(1, dialogId, null,
limit), observer);
    }
public void getMess (String dialogId, Message message) {
    executeUnit(new ChatUseCaseJava.ChatParams(2, dialogId,
message, 0));
    }
    static class ChatParams {
        private int type;
        private String dialogId;
        private Message message;
        private int limit;
        public ChatParams(int type, String dialogId, Message
message, int limit) {
            this.type = type;
            this.dialogId = dialogId;
```

```
            this.message = message;
            this.limit = limit;
        }
        public int getType() {
            return type;
        }
        public void setType(int type) {
            this.type = type;
        }
        public String getDialogId() {
            return dialogId;
        }
        public void setDialogId(String dialogId) {
            this.dialogId = dialogId;
        }
        public Message getMessage() {
            return message;
        }
        public void setMessage(Message message) {
            this.message = message;
        }
        public int getLimit() {
            return limit;
        }
        public void setLimit(int limit) {
            this.limit = limit;
        }
    }
}
```

```
@ChatScope
class ChatPresenter @Inject constructor(private val useCase:
ChatUseCase)
    :BasePresenter<ChatView,List<DocumentSnapshot>>() {
    var isLoading = false
    fun initialize(dialogId:String,limit:Int = 40){
        if (isLoading)return
        isLoading = true
        useCase.clean()
        useCase.getMessages(dialogId,MessageObserver(),limit)
    }
    fun sentMessage(dialogId:String,message: Message){
        useCase.sentMessage(dialogId, message)
    }
    override fun updateView() {
        model?.let { view()?.initQuery(it) }
    }
    public override fun destroy() {
        useCase.dispose()
        Injector.clearChatComponent()
    }
inner class MessageObserver: DefaultObserver<QuerySnapshot>() {
        override fun onNext(t: QuerySnapshot) {
            model = t.documents
            if (!isLoading){
                view()?.renderQuery(t)
            }else{
                view()?.initQuery(t.documents)
                isLoading = false
            }
        }
        override fun onError(e: Throwable) {
            view()?.showError(e.message?:"error")}}}
```