Мнистерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)

Факультет **Инфокоммуникационных технологий**

Образовательная программа **Интеллектуальные системы в гуманитарной сфере**

Направление подготовки **45.03.04 Интеллектуальные системы в гуманитарной сфере**

Отчет
по курсовой работе
по предмету “Мобильная разработка (Android и IOs)”

Обучающаяся: Мильберг Кристина Алексеевна
Группа: К3444

Санкт-Петербург , 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данный цикл из четырёх лабораторных работ посвящён поэтапной разработке Android-приложения «MessengerApp» на языке Kotlin с использованием архитектурного паттерна MVVM и современных компонентов Android Jetpack.

Все лабораторные выполнялись на базе одного и того же проекта: на этапе ЛР1 создавался базовый каркас приложения, а в последующих работах приложение постепенно дополнялось новой функциональностью.

Поэтому скриншоты в отчёте демонстрируют текущее (финальное) состояние приложения после выполнения всех четырех лабораторных работ, при этом часть интерфейсных элементов могла выглядеть проще на ранних этапа

Лабораторная 1

Тема: Создание базового каркаса Android-приложения с навигацией

1 Задача:

Разработать простое Android-приложение "Мессенджер" с использованием архитектуры на основе Activity и Fragment, а также встроенной навигации

Стэк:

Java/Kotlin

2 Функциональные требования:

1. Приложение должно содержать одну главную активность, которая выступает в качестве точки входа

2. Внутри активности должна быть реализована навигация с помощью Bottom Navigation

3. Каждая вкладка в панели должна открывать отдельный экран:

- экран новостной ленты (пока только заглушка с текстом);
- экран профиля пользователя с отображением основных данных;
- экран настроек (минимум — базовый элемент интерфейса, например переключатель темы)

4. Навигация между экранами должна выполняться через стандартные инструменты Android Navigation (NavController и Navigation Graph)

5. В проекте необходимо реализовать отслеживание жизненного цикла: при создании и завершении работы компонентов выводить сообщения в лог

Нефункциональные требования:

- Код должен быть структурирован и поддерживать дальнейшее расширение функционала
- Приложение должно запускаться без ошибок (лол, очевидно)
- Интерфейс должен быть понятным

3 Ход работы

3.1 Структура проекта

Было создано Android-приложение на Kotlin со следующей структурой:

- MainActivity — основная Activity, содержащая NavHostFragment и BottomNavigationView.
- FeedFragment — экран «Лента».
- ProfileFragment — экран «Профиль».
- SettingsFragment — экран «Настройки».
- activity_main.xml — разметка с контейнером навигации.
- nav_graph.xml — граф навигации между фрагментами.

В activity_main.xml размещены:

- FragmentContainerView / NavHostFragment с android:name="androidx.navigation.fragment.NavHostFragment";
- BottomNavigationView с пунктами меню «Лента», «Профиль», «Настройки».

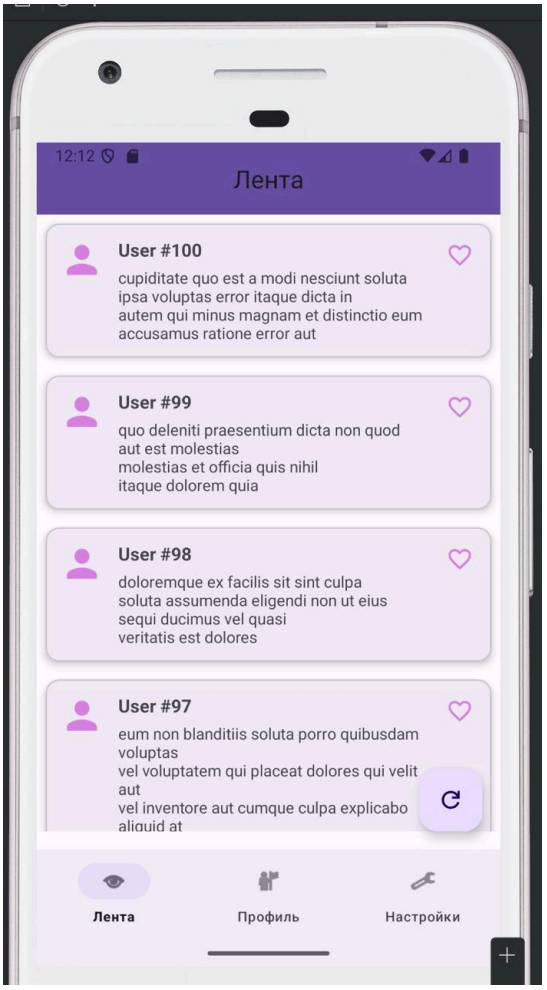

Рис. 1 – Главный экран приложения с нижней навигацией (интерфейс в финальном виде, навигация реализована на этапе ЛР1)

3.2 Настройка навигации

В MainActivity производится инициализация навигации:

- поиск NavHostFragment по ID nav_host_fragment,
- получение navController,

- привязка BottomNavigationView к navController через setupWithNavController.

Это позволяет переключаться между тремя основными фрагментами без создания дополнительных Activity.

3.3 Логирование жизненного цикла

Для демонстрации жизненного цикла компонентов были добавлены вызовы Log.d в методах:

- onCreate, onDestroy в MainActivity;
- onCreate, onCreateView, onDestroy в каждом фрагменте.

Это позволяет в Logcat наблюдать порядок вызова методов при запуске приложения, навигации между экранами и закрытии активити.

Вывод по ЛР1

В ходе первой лабораторной работы был создан каркас приложения «MessengerApp» с одной Activity и тремя фрагментами, организована навигация с помощью BottomNavigationView и Navigation Component, а также реализовано логирование жизненного цикла Activity и Fragment. Это заложило основу для дальнейшего развития приложения в последующих лабораторных работах.

Лабораторная 2

Тема: Использование MVVM, ViewModel и LiveData для хранения состояния

1 Задача:

Реализовать управление состоянием приложения и архитектуру MVVM для хранения и обновления пользовательских данных в экранах

2 Функционал:

1. Добавить в проект ViewModel для хранения состояния профиля и настроек
2. Экран “Профиль” должен отображать имя и статус пользователя, редактируемые через EditText и сохраняемые во ViewModel
3. Экран “Настройки” должен содержать переключатель темы (светлая/тёмная) - состояние хранится во ViewModel
4. При повороте экрана данные не должны теряться
5. В логах фиксировать этапы жизненного цикла ViewModel

Нефункциональные требования:

- Применить принципы разделения ответственности между UI и ViewModel;
- Обеспечить реактивное обновление UI через LiveData

3 Ход работы

3.1 Реализация MainViewModel

Создан класс MainViewModel, унаследованный от ViewModel. В нём хранится состояние:

- имя пользователя;
- статус пользователя;
- флаг включения тёмной темы.

Данные оформлены через MutableLiveData (внутри ViewModel) и LiveData (для внешнего доступа):

```
class MainViewModel : ViewModel() {

    private val _userName = MutableLiveData("Имя пользователя")
    val userName: LiveData<String> get() = _userName

    private val _userStatus = MutableLiveData("Онлайн")
```

```
        val userStatus: LiveData<String> get() = _userStatus

        private val _isDarkThemeEnabled = MutableLiveData(false)
        val isDarkThemeEnabled: LiveData<Boolean> get() = _isDarkThemeEnabled

        init {
            Log.d("MainViewModel", "init: ViewModel создана")
        }

        fun updateName(newName: String) { _userName.value = newName }
        fun updateStatus(newStatus: String) { _userStatus.value = newStatus }
         fun setDarkThemeEnabled(enabled: Boolean) { _isDarkThemeEnabled.value
= enabled }

        override fun onCleared() {
            super.onCleared()
            Log.d("MainViewModel", "onCleared: ViewModel уничтожена")
        }
    }
```

3.2 Подключение ViewModel во фрагментах

Во фрагментах, которые используют общее состояние (например, ProfileFragment и SettingsFragment), ViewModel получена через:

private val viewModel: MainViewModel by activityViewModels()

Это гарантирует, что одна и та же MainViewModel доступна нескольким фрагментам в рамках одной Activity.

3.3 Экран «Профиль»

В ProfileFragment реализованы:

- поля ввода (EditText) для имени и статуса;
- подписка на LiveData из ViewModel;
- отправка изменений обратно в ViewModel при вводе текста.

Таким образом, введённые данные:

- отображаются при каждом открытии фрагмента,
- не теряются при повороте экрана или переключении вкладок.

Рис. 2 – Экран «Профиль» (данные берутся из MainViewModel)

3.4 Экран «Настройки» и тема приложения

Переключатель темы (MaterialSwitch) в SettingsFragment связан с MainViewModel.isDarkThemeEnabled.

Фрагмент:

- наблюдает за isDarkThemeEnabled и устанавливает нужный режим через AppCompatDelegate.setDefaultNightMode(...);
- при изменении состояния переключателя обновляет значение во ViewModel.

Таким образом, режим (светлый/тёмный) привязан к состоянию ViewModel и сохраняется между пересозданиями активити.

Рис. 3 – Экран «Настройки» с переключателем темы

Вывод по ЛР2

В рамках второй лабораторной работы была внедрена архитектура MVVM: данные профиля и настройки темы вынесены в MainViewModel, фрагменты работают с состоянием через LiveData. Это позволило сохранить данные при повороте экрана и переключении вкладок, а также упростило дальнейшее развитие приложения за счёт разделения логики и представления.

Лабораторная 3

Тема: Загрузка данных из сети и локальное кэширование с помощью Room

1 Задача:

Добавить загрузку списка сообщений из API и реализовать сохранение данных в локальную базу Room для офлайн-доступа

2 Функционал:

1. Создать MessageRepository, который получает данные из публичного API
2. Реализовать Retrofit-клиент с корутинами
3. Отображать список сообщений на вкладке “Лента” с помощью RecyclerView
4. Сохранять полученные данные в локальную базу Room
5. При отсутствии сети - загружать сообщения из базы
6. Добавить кнопку “Обновить” для ручного обновления данных

Нефункциональные требования:

Сетевые операции только через корутины

Работа с базой через DAO и Entity

Репозиторий - единая точка взаимодействия ViewModel с данными

3 Ход работы

3.1 Публичное API

Для примера выбран учебный публичный API: https://jsonplaceholder.typicode.com/posts

Он возвращает список постов в формате JSON, каждый из которых рассматривается как «сообщение» в ленте.

Создан DTO:

```
data class MessageDto(
    val id: Int,
    val title: String,
    val body: String,
)
```

и интерфейс Retrofit:

```
interface MessageApi {
    @GET("posts")
    suspend fun getMessages(): List<MessageDto>
}
```

Клиент Retrofit (ApiClient) настроен с базовым URL и конвертером GSON.

3.2 Room: сущность, DAO, база

Сущность сообщения в базе:

```
@Entity(tableName = "messages")
data class MessageEntity(
    @PrimaryKey val id: Int,
    val title: String,
    val body: String,
)
```

DAO:

```
@Dao
interface MessageDao {
    @Query("SELECT * FROM messages ORDER BY id DESC")
    fun getAllMessages(): Flow<List<MessageEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertMessages(messages: List<MessageEntity>)

    @Query("DELETE FROM messages")
    suspend fun clearAll()
}
```

База данных AppDatabase реализована как Singleton через Room.databaseBuilder.

3.3 Репозиторий MessageRepository

Репозиторий инкапсулирует работу с сетью и Room:

- messagesFlow: Flow<List<MessageEntity>> — поток данных из локальной БД;
- refreshMessages() — загрузка данных из сети и сохранение в БД.

```
class MessageRepository(
    private val api: MessageApi,
    private val dao: MessageDao,
) {
    val messagesFlow: Flow<List<MessageEntity>> = dao.getAllMessages()

    suspend fun refreshMessages() {
        val remoteMessages = api.getMessages()
        val entities = remoteMessages.map { dto ->
            MessageEntity(
                id = dto.id,
                title = dto.title,
                body = dto.body,
            )
```

```
            }
            dao.clearAll()
            dao.insertMessages(entities)
        }
    }
```

3.4 FeedViewModel и привязка к UI

FeedViewModel получает в конструкторе Application, создаёт базу и репозиторий, а также определяет:

- messages: LiveData<List<MessageEntity>> — результаты messagesFlow.asLiveData();
- isLoading: LiveData<Boolean> — состояние загрузки;
- error: LiveData<String?> — текст ошибки при неудачной попытке обновления.

Загрузка выполняется через viewModelScope.launch, что не блокирует главный поток.

3.5 Вывод списка в RecyclerView

На экране «Лента» (FeedFragment) используется RecyclerView с адаптером MessageAdapter. Адаптер получает список сообщений и отображает их в разметке item_message.xml.

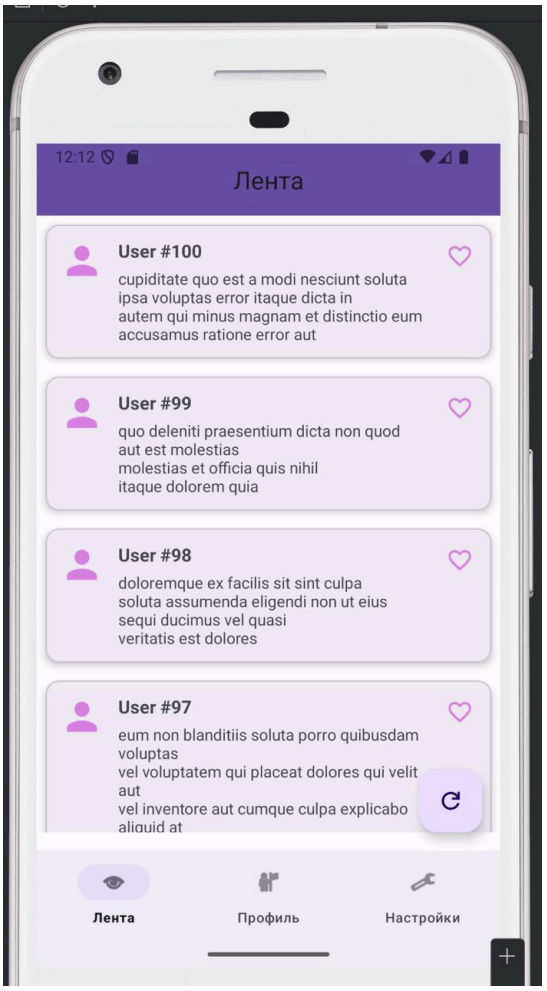

Рис. 4 – Лента сообщений, загруженных из сети и сохранённых в Room (функциональность реализована в ЛР3)

При отсутствии сети данные продолжают отображаться из локальной базы, что обеспечивает базовый офлайн-режим.

Вывод по ЛР3

В третьей лабораторной работе реализована связка: публичный REST-API → Retrofit + корутины → Repository → Room → ViewModel → RecyclerView. Приложение научилось загружать данные из сети, кэшировать их в локальной базе и продолжать отображать в офлайн-режиме, что приближает его к реальным клиент-серверным приложениям.

Лабораторная 4

Тема: Улучшение интерфейса, лайки и фоновая синхронизация

1 Задача:

Улучшить пользовательский интерфейс, добавить уведомления и фоновую синхронизацию сообщений

2 Функционал:

1. Реализовать кастомный RecyclerView.Adapter для списка сообщений аватарка, имя, текст

2. Применить компоненты Material Design:

CardView , FloatingActionButton , AppBarLayout

3. Добавить возможность “лайкать” сообщение (иконка изменяется при клике)

4. Настроить WorkManager , который раз в какое-то время обновляет список сообщений из сети

5. При успешной синхронизации показывать Notification с сообщением “Новые данные получены”

6. При запуске приложения запрашивать разрешения (уведомления, контакты и т.п.)

Уведомления должны работать в фоне

Приложение должно корректно реагировать на системные события (онлайн/офлайн)

Код оформлен в соответствии с принципами MVVM

3 Ход работы

3.1 Кастомный адаптер и интерфейс сообщений

Разметка item_message.xml переработана с учётом Material Design:

- корневой элемент — MaterialCardView;
- внутри располагаются:

  - ImageView с аватаркой;
  - TextView с условным «именем» пользователя (например, User #id);
  - TextView с текстом сообщения;
  - ImageButton с иконкой лайка (активной или неактивной).

В адаптере MessageAdapter реализовано:

- хранение текущего списка сообщений;
- получение множества лайкнутых ID из ViewModel;

- установка разных иконок (ic_favorite_24, ic_favorite_border_24) в зависимости от того, лайкнуто ли сообщение.

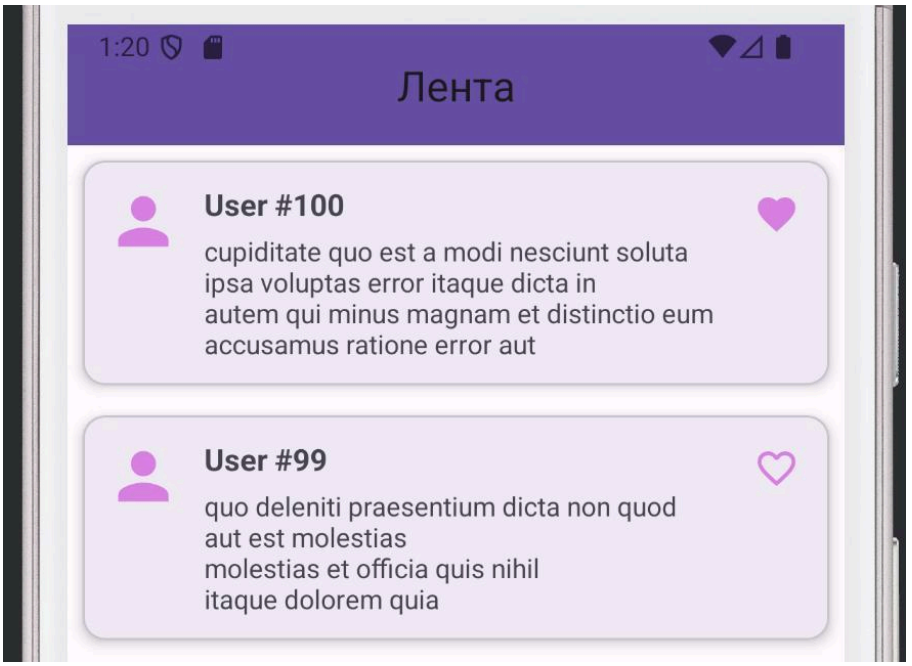

Рис. 5 – Кастомные карточки сообщений с аватаркой и лайком

3.2 Компоненты Material Design

Разметка fragment_feed.xml переведена на использование:

- CoordinatorLayout как корневого контейнера;
- AppBarLayout и MaterialToolbar в качестве верхней панели с заголовком;
- RecyclerView в качестве основного содержимого;
- FloatingActionButton (FAB) для обновления списка сообщений.

FAB привязан к методу refreshMessages() во ViewModel и визуально является кнопкой «Обновить».

3.3 Лайки во ViewModel (MVVM)

В FeedViewModel добавлено поле:

```
private val _likedIds = MutableLiveData<Set<Int>>(emptySet())
val likedIds: LiveData<Set<Int>> get() = _likedIds
```

и метод:

```
fun toggleLike(id: Int) {
    val current = _likedIds.value ?: emptySet()
     _likedIds.value = if (current.contains(id)) current - id else current
+ id
}
```

Таким образом:

- логика лайков сосредоточена во ViewModel;
- адаптер является «тупым» отображением состояния;

- при клике на кнопку лайка вызывается toggleLike(id), и UI обновляется через LiveData.

3.4 WorkManager и фоновая синхронизация

Создан SyncWorker, наследующий CoroutineWorker. В методе doWork():

1. Получается доступ к базе и DAO.
2. Создаётся MessageRepository.
3. Вызывается repository.refreshMessages() для синхронизации.
4. При успехе показывается уведомление.

С помощью WorkManager настроены:

- периодический запрос (PeriodicWorkRequest) с интервалом 15 минут и условием наличия сети (NetworkType.CONNECTED);
- однократный запрос (OneTimeWorkRequest) при запуске приложения, чтобы сразу выполнить синхронизацию и показать уведомление.

В MainActivity.onCreate() вызывается метод:

```
SyncWorker.schedule(this)
```

что запускает планирование фоновой работы.

3.5 Уведомления

В SyncWorker создаётся канал уведомлений (для Android 8+), а затем формируется уведомление с текстом: «Новые данные получены»

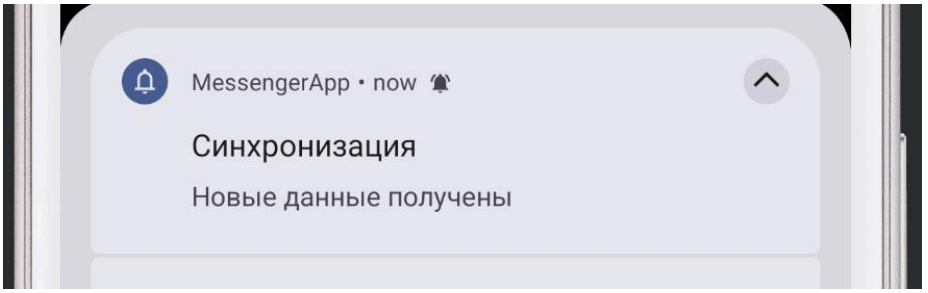

Рис. 6 – Уведомление о новой синхронизации данных

Уведомления продолжают приходить и при работе приложения в фоне, что отвечает условию задания.

3.6 Runtime-разрешения

В MainActivity реализован запрос разрешений:

- POST_NOTIFICATIONS (для Android 13+),
- READ_CONTACTS (как пример дополнительного разрешения).

Проверка выполняется через ContextCompat.checkSelfPermission, запрос — через ActivityCompat.requestPermissions, результаты логируются в onRequestPermissionsResult.

При первом запуске пользователь видит диалоги с запросом разрешений, что соответствует современным требованиям безопасности Android.

Вывод по ЛР4

В рамках четвёртой лабораторной работы приложение было существенно улучшено с точки зрения пользовательского интерфейса и UX: внедрены компоненты Material Design, добавлены лайки к сообщениям, реализована фоновая синхронизация данных с использованием WorkManager и система уведомлений. Приложение корректно реагирует на отсутствие сети и продолжает показывать данные из локальной базы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения четырёх лабораторных работ было поэтапно разработано Android-приложение «MessengerApp», которое иллюстрирует полный цикл реализации клиента:

- от базового каркаса с навигацией (ЛР1),
- через внедрение архитектуры MVVM и ViewModel / LiveData для работы с состоянием (ЛР2);
- до интеграции с удалённым API, локального кэширования в базе данных Room и офлайн-режима (ЛР3);
- и завершая улучшением интерфейса, лайками, фоновой синхронизацией и уведомлениями (ЛР4).

Применение архитектурного паттерна MVVM, использование современных библиотек Android Jetpack (Navigation, ViewModel, LiveData, Room, WorkManager) и компонентов Material Design позволило получить структурированное, расширяемое и удобное для пользователя приложение, которое демонстрирует типичные подходы к разработке мобильных приложений под Android.