Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Кузбасский государственный технический университет
имени Т. Ф. Горбачева»

Институт профессионального образования

Кафедра информатики и информационных систем

Составитель К. В. Ерошевич

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ МОБИЛЬНЫХ ПЛАТФОРМ

Методические материалы

Рекомендовано цикловой методической комиссией
Информационных систем и программирования
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2026

Рецензент:
К. А. Кулиничев, преподаватель первой квалификационной категории кафедры информатики и информационных систем

Ерошевич Кирилл Владимирович
Разработка приложений для мобильных платформ : методические материалы для обучающихся специальности СПО 09.02.11 Разработка и управление программным обеспечением / Кузбасский государственный технический университет имени Т. Ф. Горбачева, Кафедра информатики и информационных систем ; составитель К. В. Ерошевич. – Кемерово, 2026. – 1 файл (582 Кб). – Текст : электронный.

Методические материалы по дисциплине «Разработка приложений для мобильных платформ» для обучающихся специальности СПО 09.02.11 «Разработка и управление программным обеспечением» содержат указания для проведения лабораторных работ.

Лабораторная работа №1
«Создание проекта и настройка build.gradle»

Теоретическая часть

Build.gradle – это система автоматической сборки (основана на Groovy или Kotlin DSL). В проекте Android есть два основных файла:

1. Project-level (build.gradle.kts) – задает репозитории и версии плагинов для всех модулей.
2. Module-level (app/build.gradle.kts) – содержит конфигурацию конкретного модуля:
    - compileSdk (версия API, с которой компилируется код);
    - minSdk (минимальная версия Android для установки);
    - targetSdk (целевая версия Android);
    - dependencies (список библиотек, подключаемых к проекту, например, implementation 'androidx.appcompat:appcompat:1.6.1').

AndroidManifest.xml (Манифест) – обязательный файл, описывающий информацию о приложении: имя пакета, компоненты (Activity, Service), разрешения (permissions) и запускаемый экран (LAUNCHER intent-filter).

Задание

1. Создать новый проект в Android Studio с шаблоном Empty Views Activity.
2. Указать имя пакета (Package name) по схеме: ru.yourname.lab1.
3. Выбрать язык разработки Kotlin и минимальный SDK API 24.
4. Изучить структуру проекта.
5. В файле app/build.gradle.kts изменить версию targetSdk на самую актуальную (например, 34).
6. Добавить в секцию dependencies библиотеку для логирования implementation("com.jakewharton.timber:timber:5.0.1").
7. Синхронизировать проект (Sync Now).
8. Запустить приложение на эмуляторе.

Контрольные вопросы

1. Для чего нужен файл AndroidManifest.xml?
2. В чем разница между minSdk и targetSdk?
3. Что произойдет, если не вызвать синхронизацию Gradle после добавления зависимости?

Лабораторная работа №2
«Работа с интентами и запуск внешних Activity»

Теоретическая часть

Intent – это объект сообщения, который запрашивает действие от другого компонента приложения (или системы).

Явные интенты (Explicit) – указывают конкретный класс для запуска. Используются внутри приложения для переключения между своими экранами.

Неявные интенты (Implicit) – описывают действие (ACTION_VIEW, ACTION_DIAL), которое нужно выполнить. Система ищет приложения, способные это сделать.

Примеры неявных интентов:

1. Открыть веб-страницу: Intent(Intent.ACTION_VIEW, Uri.parse("https://...")).
2. Позвонить по номеру: Intent(Intent.ACTION_DIAL, Uri.parse("tel:12345")).
3. Отправить письмо: Intent(Intent.ACTION_SENDTO, Uri.parse("mailto:...")).

Задание

1. На основе проекта из Лабораторной работы №1 создать макет с тремя кнопками.
2. Написать обработчики нажатий:
    - Первая кнопка: Явный Intent. Создать вторую пустую Activity (SecondActivity) и реализовать переход на нее.
    - Вторая кнопка: Неявный Intent. Открыть браузер с переходом на сайт университета.
    - Третья кнопка: Неявный Intent. Открыть карты с координатами (геометка) главного корпуса.
3. Добавить проверку (в коде), существует ли приложение для обработки неявного интента, чтобы избежать краша.

Контрольные вопросы

1. Чем явный Intent отличается от неявного?
2. Как передать данные (например, строку) из MainActivity в SecondActivity?
3. Что такое IntentFilter?

Лабораторная работа №3
«Настройка Retrofit и получение данных с сервера»

Теоретическая часть

Retrofit – это типобезопасный HTTP-клиент для Android и Java, созданный компанией Square. Позволяет легко превращать HTTP API в Java/Kotlin интерфейс.

Зависимости:

1. com.squareup.retrofit2:retrofit;
2. com.squareup.retrofit2:converter-gson.

Разрешения:

Для работы с интернетом необходимо добавить в манифест строку
<uses-permission android:name="android.permission.INTERNET" />.

Основные компоненты:

1. Data Class – модель данных, которую мы ожидаем получить от сервера.
2. API Interface – описание методов HTTP (GET, POST) с помощью аннотаций.
3. Retrofit.Builder – класс для создания экземпляра Retrofit с указанием базового URL и конвертера (Gson).
4. Callback – обработка ответа в фоновом потоке (или использование Coroutines).

Задание

1. Создать новый проект или продолжить прошлый.
2. Добавить в build.gradle зависимости для Retrofit и Gson.
3. Добавить разрешение на доступ в Интернет в AndroidManifest.xml.
4. Создать data class Post с полями: id (Int), title (String), body (String).
5. Создать интерфейс ApiService с методом getPosts().
6. Написать класс Network Manager, который создает и предоставляет экземпляр Retrofit (базовый URL: https://jsonplaceholder.typicode.com).

7. В MainActivity выполнить GET-запрос к /posts (получить список постов) и вывести в TextView количество полученных записей (например, "Загружено 100 постов").

Контрольные вопросы

1. Для чего нужен конвертер (Converter) в Retrofit?
2. Почему нельзя выполнять сетевые запросы в главном потоке (UI Thread)?
3. Что такое "блокирующий" и "неблокирующий" вызов?

Лабораторная работа №4
«Кэширование данных в SharedPreferences и Room»

Теоретическая часть

В Android существует несколько *способов хранения данных*:

1. **SharedPreferences** – хранение простых пар "ключ-значение" (ключ – String, значение – примитивный тип). Данные хранятся в XML файле. Идеально подходит для настроек приложения, состояния чекбоксов, последней сессии пользователя.
    - Получение: getSharedPreferences("fileName", Context.MODE_PRIVATE).
    - Запись: edit().putString("key", "value").apply().
2. **Room Database** – библиотека-надстройка над SQLite, обеспечивающая типобезопасность и проверку SQL-запросов на этапе компиляции. Состоит из трех основных компонентов:
    - Entity: Класс данных (таблица в базе данных).
    - DAO (Data Access Object): Интерфейс, содержащий методы для работы с данными (@Insert, @Query, @Update, @Delete).
    - Database: Абстрактный класс, который объединяет DAO и Entity, а также предоставляет экземпляр базы данных (реализуется через Singleton).

Задание

1. Взять за основу проект из Лабораторной работы №3 (работа с сетью).
2. Часть А (SharedPreferences):
    - При первом запуске приложения (при успешном получении данных с сервера) сохранить во SharedPreferences текущую временную метку (timestamp) и общее количество загруженных постов.
    - При последующих запусках выводить в Toast сообщение: "Данные обновлены N минут назад. Всего постов: M".
3. Часть Б (Room):

- Добавить в build.gradle плагин id("kotlin-kapt") и зависимости room-runtime, room-ktx, room-compiler (kapt).
- Создать Entity – PostEntity (поля: id, title, body, userId).
- Создать интерфейс PostDao с методами insertAll и getAllPosts.
- Создать класс AppDatabase (абстрактный, наследник RoomDatabase), объединяющий сущность и DAO.
- Реализовать сохранение списка постов, полученных из сети (из ЛР №3), в базу данных Room.

Контрольные вопросы

1. В чем разница между apply() и commit() в SharedPreferences?
2. Почему Room не позволяет делать запросы к базе данных в главном потоке (до версии 2.3.0)?
3. Для чего нужна аннотация @PrimaryKey(autoGenerate = true) в Room?
4. Что произойдет с данными в базе данных (БД) при обновлении версии приложения, если не написать миграцию?

Лабораторная работа №5
«Режимы работы в offline»

Теоретическая часть

Современные приложения должны корректно работать как при наличии интернета, так и без него.

Основные подходы:

Single Source of Truth (Единый источник правды):

- Данные всегда читаются из локальной базы данных (Room).
- При наличии сети данные загружаются с сервера, обновляют БД, а UI автоматически перерисовывается (часто с использованием LiveData/Flow).

Проверка состояния сети:

- Использование ConnectivityManager для определения наличия активного интернет-соединения перед выполнением сетевого запроса.

Стратегии загрузки:

1. **Network-first**: Сначала выполняется попытка загрузить данные с сервера, если возникает ошибка – берем из локального кэша.
2. **Cache-first**: Сначала показываем данные из БД, а затем (в фоновом режиме) загружаем свежие данные с сервера.

Задание

1. Модифицировать приложение из ЛР №4.
2. Создать утилитный класс NetworkUtils, который с помощью ConnectivityManager проверяет, есть ли подключение к интернету.
3. Реализовать гибридную загрузку:
    - При запуске приложения, независимо от наличия сети, немедленно показать данные из базы данных Room.
    - Если есть интернет, выполнить фоновый запрос к Retrofit, обновить базу данных Room новыми данными.
    - Интерфейс должен автоматически обновиться (использовать LiveData или Flow в DAO).

4. Добавить индикатор (например, иконку или текст в Toolbar), который показывает текущий статус подключения (Online/Offline).

Контрольные вопросы

1. Что такое "Single Source of Truth" в контексте мобильной разработки?
2. Как уведомить пользователя о том, что данные могли быть загружены из кэша и являются неактуальными?
3. В чем преимущество использования LiveData или Flow в связке с Room?
4. Какие разрешения нужны для проверки состояния сети?

Лабораторная работа №6
«Работа с системными логами и отладкой Logcat»

Теоретическая часть

Logcat – это инструмент командной строки Android, который выгружает системные сообщения, включая стек-трейсы ошибок и пользовательские сообщения, добавленные с помощью класса Log.

Уровни логирования:

1. Log.v() – VERBOSE (подробные сообщения).
2. Log.d() – DEBUG (отладочная информация).
3. Log.i() – INFO (информационные сообщения).
4. Log.w() – WARN (предупреждения).
5. Log.e() – ERROR (ошибки).
6. Log.wtf() – What a Terrible Failure (фатальные ошибки).

Фильтрация: В окне Logcat можно фильтровать сообщения по:

1. Уровню логирования.
2. Имени пакета приложения.
3. Тексту сообщения.
4. PID (Process ID).

Брейкпоинты (Breakpoints) – позволяют приостановить выполнение программы на определенной строке для анализа значений переменных и стека вызовов.

Задание

1. Создать новый проект или использовать любой существующий.
2. Работа с фильтрами:
    - Добавить в код MainActivity в метод onCreate пять строк с разными уровнями логирования (Log.d(), Log.e(), Log.i()). Каждое сообщение должно содержать уникальный тег, например "MyAppLifecycle".
    - Запустить приложение и в окне Logcat настроить фильтр, чтобы отображать сообщения только с тегом "MyAppLifecycle".

3. Поиск ошибок:
    - Специально добавить в код ошибку (например, обратиться к несуществующему элементу списка по индексу arrayList[100] при размере списка 5).
    - Запустить приложение, поймать краш и проанализировать стек-трейс в Logcat, чтобы найти строку, вызвавшую ошибку.
4. Отладка (Debug):
    - Установить брейкпоинт на строчке с логированием в onCreate.
    - Запустить приложение в режиме Debug (зеленый жучок).
    - При остановке программы попрактиковаться в просмотре переменных (вкладка Variables) и выполнении построчно (Step Over / Step Into).

Контрольные вопросы

1. Для чего используется уровень логирования Log.e()?
2. Можно ли увидеть логи своего приложения на боевом (релизном) устройстве без подключения к студии?
3. В чем разница между Step Over (F8) и Step Into (F7) при отладке?
4. Как в Logcat найти логи только с критическими ошибками, игнорируя информационные сообщения?

Лабораторная работа №7
«Создание асинхронных запросов с Coroutines»

Теоретическая часть

Coroutines (Корутины) – это инструмент Kotlin для асинхронного программирования, позволяющий писать неблокирующий код в последовательном стиле. В отличие от callback'ов, корутины облегчают чтение и поддержку кода.

Основные понятия:

1. CoroutineScope – определяет жизненный цикл корутины. Если скоуп уничтожается, все запущенные в нем корутины отменяются.
2. Dispatchers – указывают, на каком пуле потоков будет выполняться код:
    - Dispatchers.Main – для UI операций.
    - Dispatchers.IO – для операций ввода-вывода (сеть, база данных).
    - Dispatchers.Default – для тяжелых вычислений (CPU-bound).
3. Suspend Functions – функции, которые можно приостановить и возобновить позже. Они могут выполняться только внутри корутины или другой suspend-функции.
4. Структура: viewModelScope.launch { } – запуск корутины из ViewModel с автоматической отменой всех активных задач при очистке ViewModel.

Задание

1. Взять проект из ЛР №5 (с Room и сетью) и выполнить его рефакторинг, заменив использование callback'ов (enqueue) на корутины.
2. В build.gradle добавить зависимости для корутин: implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3").

3. Модифицировать ApiService – добавить suspend перед методами (Retrofit поддерживает suspend-функции начиная с версии 2.6.0).
4. В ViewModel (или MainActivity, если нет архитектуры) запустить корутину с viewModelScope.launch(Dispatchers.IO) { }.
5. Внутри корутины выполнить последовательно два запроса:
    - Загрузить список постов из сети.
    - Сохранить их в базу данных Room.
6. Для обновления UI (например, TextView) после сохранения, переключить контекст обратно на Dispatchers.Main с помощью withContext(Dispatchers.Main).

Контрольные вопросы

1. Чем корутины отличаются от обычных потоков (Threads)?
2. Что произойдет, если внутри корутины, запущенной в Dispatchers.Main, вызвать suspend-функцию, которая делает сетевой запрос (например, api.getPosts())?
3. Для чего нужен оператор withContext?
4. Что такое viewModelScope и как он управляет жизненным циклом корутин?

Лабораторная работа №8
«Подключение базы данных Room»

Теоретическая часть

Room предоставляет мощные возможности для работы со связями между таблицами и асинхронными запросами.

Типы отношений:

1. @ForeignKey – определение внешних ключей между таблицами для обеспечения целостности данных (каскадное удаление и т. д.).
2. @Relation – аннотация для создания POJO-классов, содержащих вложенные объекты (например, "Пользователь" со списком его "Постов").

Типы возвращаемых значений в DAO:

1. LiveData<T> – автоматическое обновление UI при изменении данных.
2. Flow<T> – реактивный поток из библиотеки Kotlin Coroutines.
3. suspend fun getAll(): List<T> – одноразовый запрос в корутине.

Задание

1. Модифицировать проект из ЛР №7.
2. Создать вторую сущность (Entity) User (поля: userId, name, email).
3. В классе Post (Entity) поле userId сделать внешним ключом, ссылающимся на User.
4. Создать UserDao с методами insert и getUserById.
5. Создать класс UserWithPosts, используя аннотацию @Relation, чтобы получить пользователя и все его посты в одном объекте.
6. В DAO написать транзакционный метод @Transaction для получения UserWithPosts по ID пользователя.
7. Заполнить базу тестовыми данными: создать двух пользователей и по 5 постов для каждого из них.

Контрольные вопросы

1. Для чего нужна аннотация @ForeignKey? Что такое onDelete = CASCADE?
2. Чем LiveData в запросе Room отличается от простого List?
3. Почему для запросов, возвращающих отношения с коллекциями, рекомендуется использовать @Transaction?
4. Как бы вы организовали миграцию базы данных с версии 1 на версию 2 при добавлении новой таблицы?

Лабораторная работа №9
«Реализация репозитория с двумя источниками (сеть + кэш)»

Теоретическая часть

Паттерн Репозиторий (Repository Pattern) – это прослойка между источниками данных (сеть, кэш, БД) и остальной частью приложения (ViewModel/UI). Репозиторий инкапсулирует логику принятия решения, откуда брать данные.

Преимущества:

1. Единая точка доступа к данным. ViewModel не знает, откуда пришли данные – из сети или БД.
2. Упрощение тестирования. Можно подменить реальные источники на фейковые (mock).
3. Чистая архитектура. Код становится масштабируемым.

Типичная реализация метода getPosts() в репозитории:

```
fun getPosts(): Flow<List<Post>> = flow {
    // 1. Сначала отправляем (эмитим) данные из БД (кеш)
    val cachedData = postDao.getAllPosts()
    emit(cachedData)

    // 2. Пытаемся загрузить свежие данные из сети
    try {
        val freshData = apiService.getPosts()
        // 3. Сохраняем новые данные в БД (кеш обновится)
        postDao.insertAll(freshData)
        // 4. Новые данные автоматически придут (проэмитятся) через Flow из Room,
        // поэтому emit повторно не требуется.
    } catch (e: Exception) {
        // Ошибка сети – логируем, но не прерываем поток (пользователь уже видит кеш)
    }
}.flowOn(Dispatchers.IO) // Выполняем все на IO диспатчере (выполняем операции ввода-вывода в фоновом потоке)
```

Задание

1. Создать в проекте отдельный пакет repository.
2. Создать интерфейс IPostRepository с методами getPostsStream(): Flow<List<Post>> и suspend fun refreshPosts().
3. Создать класс PostRepositoryImpl, реализующий этот интерфейс.
4. В конструктор репозитория передать зависимости: PostDao и ApiService.
5. Реализовать логику в getPostsStream, которая:
    - Немедленно возвращает текущие данные из БД (как Flow).
    - Запускает фоновую "свежую" загрузку данных из сети и обновление БД.
6. Реализовать логику в refreshPosts, которая принудительно загружает данные из сети (даже если кэш есть) и обновляет БД (для реализации свайпа для обновления – SwipeRefreshLayout).
7. В MainViewModel использовать репозиторий, а не напрямую DAO или API.

Контрольные вопросы

1. Почему репозиторий возвращает Flow, а не List?
2. Как изменится логика репозитория, если мы захотим реализовать политику "сеть важнее кэша"?
3. Где должна происходить обработка ошибок сети: в ViewModel или в Репозитории?
4. Как подменить реальный репозиторий на фейковый для тестирования ViewModel?

Лабораторная работа №10
«Отложенные задачи с WorkManager»

Теоретическая часть

WorkManager – это библиотека Android Jetpack для выполнения отложенных фоновых работ, которые должны быть гарантированно выполнены даже при выходе приложения из памяти или перезагрузке устройства. Она учитывает ограничения и жизненный цикл приложения, выбирая подходящий способ выполнения (JobScheduler, AlarmManager, etc.).

Основные компоненты:

1. Worker – класс, где выполняется сама задача. Переопределяется метод doWork().
2. WorkRequest – запрос на выполнение задачи. Определяет, какой Worker запустить, и задает ограничения (Constraints):
   - setConstraints() – например, только при зарядке или при наличии интернета.
   - setInitialDelay() – задержка перед выполнением.
   - setBackoffCriteria() – политика повторных попыток при ошибке.
3. WorkManager – системный сервис для постановки задач в очередь.

Типы WorkRequest:

1. OneTimeWorkRequest – для разовых задач.
2. PeriodicWorkRequest – для периодических задач (минимальный интервал – 15 минут).

Задание

1. Создать новый проект или использовать существующий.
2. Добавить зависимость WorkManager в build.gradle: implementation("androidx.work:work-runtime-ktx:2.9.0")
3. Создать класс DataSyncWorker, наследующий от Worker() или CoroutineWorker():

   - Внутри doWork() реализовать имитацию синхронизации данных (например, запись в лог и сохранение временной метки в SharedPreferences).
   - Вернуть Result.success() при успехе или Result.retry() при ошибке.
4. Создать OneTimeWorkRequest для этого Worker'a с ограничениями:
   - Требуется подключение к интернету.
   - Устройство заряжается.
5. Поставить задачу в очередь по нажатию кнопки в приложении.
6. Создать PeriodicWorkRequest, который запускается каждые 15 минут и синхронизирует данные в фоне.
7. Добавить возможность отслеживания статуса задачи через WorkManager.getWorkInfoByIdLiveData() и отображать статус (Running, Success, Failed) в UI.

Контрольные вопросы

1. Чем WorkManager отличается от обычного потока (Thread)?
2. Что произойдет с задачей WorkManager, если устройство перезагрузится до ее выполнения?
3. Какие ограничения (Constraints) можно задать для WorkRequest?
4. В чем разница между OneTimeWorkRequest и PeriodicWorkRequest?
5. Как обработать повторные попытки при сбое в Worker'e?

Лабораторная работа №11
«Загрузка фото с камеры и отправка на сервер»

Теоретическая часть

Работа с камерой и файлами в Android требует управления разрешениями и URI (Content URI / File URI).

Ключевые моменты:

1. Разрешения (Permissions):
   - CAMERA – для доступа к камере.
   - WRITE_EXTERNAL_STORAGE / READ_EXTERNAL_STORAGE (для Android < 10) или MANAGE_EXTERNAL_STORAGE (для Android 11+ Scoped Storage – обычно не требуется для сохранения фото приложения).
2. Intent для камеры: MediaStore.ACTION_IMAGE_CAPTURE – системное намерение для запуска камеры.
3. FileProvider: Специальный компонент для передачи URI файлов между приложениями (требуется для Android 7+), так как file:// URI запрещены.
4. Загрузка на сервер: Multipart-запросы с использованием Retrofit. Файл оборачивается в RequestBody и помечается аннотацией @Part MultipartBody.Part.

Задание

1. Создать новый проект с одной Activity и кнопкой "Сделать фото и отправить".
2. Добавить необходимые разрешения в манифест:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera"
android:required="true" />
```

3. Настроить FileProvider:
   - Создать путь в res/xml/file_paths.xml.
   - Прописать <provider> в AndroidManifest.xml.
4. Реализовать логику:
   - При нажатии на кнопку запросить разрешение на камеру (если еще не дано).

- Создать временный файл для сохранения фото.
- Запустить камеру через Intent, передав URI для сохранения через FileProvider.

5. В onActivityResult() (или с помощью Activity Result API) получить подтверждение, что фото сделано.
6. Настроить Retrofit с поддержкой multipart:

```
@Multipart
@POST("upload")
Suspend fun uploadImage(@Part file: MultipartBody.Part):
Response<UploadResponse>
```

7. Сжать изображение (опционально) и отправить на тестовый сервер (например, https://httpbin.org/post или imgur API).
8. Отобразить результат загрузки (успех/ошибка) на экране.

Контрольные вопросы

1. Почему для передачи URI в камеру на Android 7+ требуется FileProvider?
2. Что такое Scoped Storage и как оно влияет на сохранение файлов?
3. Как отправить изображение на сервер вместе с текстовым полем (например, описанием)?
4. В каком формате сервер обычно принимает файлы?
5. Как обработать ситуацию, если пользователь запретил разрешение на камеру?

Лабораторная работа №12
«Построение экрана с подгрузкой данных (Paging)»

Теоретическая часть

Paging Library (Paging 3) – это библиотека Android Jetpack, которая помогает загружать большие данные постранично, экономя трафик и память устройства. Она автоматически управляет загрузкой следующих страниц при прокрутке.

Основные компоненты Paging 3:

1. PagingSource – определяет источник данных (сеть, база данных). Нужно реализовать методы load() (загрузка одной страницы) и getRefreshKey() (ключ для обновления).
2. PagingConfig – конфигурация пагинации (размер страницы, предварительная загрузка, включение placeholder'ов).
3. Pager – строит поток Flow<PagingData> на основе PagingSource и конфигурации.
4. PagingDataAdapter – специальный адаптер для RecyclerView, который умеет реагировать на загрузку новых страниц и обновления данных.
5. RemoteMediator – (продвинутый уровень) для реализации кэширования – загрузка из сети и сохранение в БД Room с пагинацией.

Задание

1. Создать новый проект с RecyclerView для отображения списка постов (как в предыдущих работах).
2. Добавить зависимости Paging 3 в build.gradle: implementation("androidx.paging:paging-runtime-ktx:3.2.0").
3. Создать класс PostsPagingSource, наследующий от PagingSource<Int, Post>:
    - В методе load() выполнять запрос к API (например, api.getPosts(page, limit)), обрабатывать приходящие данные и возвращать LoadResult.Page.
    - Ключом страницы использовать номер страницы (Int).

4. В ViewModel создать Flow<PagingData<Post>> с помощью Pager(PagingConfig(pageSize = 20)) { PostsPagingSource(api) }.flow.
5. Создать адаптер PostsPagingAdapter, наследующий от PagingDataAdapter<Post, PostViewHolder>.
6. В Activity/Fragment подписаться на Flow<PagingData> из ViewModel и передавать данные в адаптер через submitData().
7. Добавить индикаторы загрузки и состояния ошибки (используя adapter.loadStateFlow).
8. Дополнительно: Реализовать поддержку БД, используя RemoteMediator, чтобы кэшировать загруженные страницы в Room.

Контрольные вопросы

1. В чем преимущество Paging Library перед ручной реализацией пагинации?
2. Что такое PagingConfig и какие параметры можно в нем задать?
3. Как обработать ошибку загрузки страницы в Paging 3?
4. Чем отличается PagingSource от RemoteMediator?
5. Как добавить заголовок (header) или подвал (footer) в список с PagingDataAdapter?

Лабораторная работа №13
«Сборка .apk/.aab для разных маркетов»

Теоретическая часть

Форматы распространения приложений:

1. **APK** (Android Package Kit) – традиционный формат установочных файлов. Содержит весь код и ресурсы приложения.
2. **AAB** (Android App Bundle) – современный формат для публикации в Google Play. Содержит скомпилированный код и ресурсы, но Google Play генерирует из него оптимизированные APK для каждого устройства (разделяя по плотности экрана, архитектуре процессора).

Типы сборок (Build Variants):

1. debug – для разработки и тестирования. Подписан отладочным ключом (автоматически).
2. Release – для публикации. Требует подписи ключом разработчика.

Подпись приложения (Signing)

Приложение должно быть подписано цифровым сертификатом, чтобы быть установленным на устройстве. Для публикации в маркетах требуется собственная подпись (Keystore).

Flavors (Продуктовые версии)

Позволяют собрать разные версии приложения из одной кодовой базы (например, бесплатная/платная версия, версия для разных регионов) с разными идентификаторами приложения, названиями и ресурсами.

Задание

1. Взять любой готовый проект из предыдущих работ.
2. Настроить Product Flavors в файле app/build.gradle.kts:
    - Создать два flavor: free и paid.
    - Для free версии изменить название приложения на "Мое приложение (Free)" и добавить баннер в ресурсах.

    - Для paid версии изменить applicationId (например, добавить суффикс .pro).
3. Создать подпись (Keystore) для релизной версии через Android Studio (Build -> Generate Signed Bundle / APK).
4. Настроить подпись в Gradle: создать конфигурацию signingConfigs с указанием путей к Keystore, паролем и алиасом (пароли лучше вынести в отдельный файл keystore.properties).
5. Собрать APK для freeRelease версии.
6. Собрать App Bundle (AAB) для paidRelease версии.
7. Сравнить размеры полученных файлов. Убедиться, что AAB меньше, чем универсальный APK.
8. Установить APK на устройство (или эмулятор) и проверить работоспособность.

Контрольные вопросы

1. В чем разница между APK и AAB? Почему Google Play рекомендует AAB?
2. Что произойдет, если потерять Keystore, которым было подписано опубликованное приложение?
3. Для чего нужны Product Flavors?
4. Какие файлы в проекте не должны попадать в Git (систему контроля версий) и почему?
5. Как проверить, что собранный APK подписан нужным ключом?

Лабораторная работа №14
«Интеграция с Yandex Maps и Geo API»

Теоретическая часть

Yandex Maps SDK позволяет встраивать интерактивные карты в мобильные приложения. Для работы требуется API-ключ.

Основные возможности:

1. Отображение карты с различными слоями (схема, спутник).
2. Управление камерой (перемещение, масштабирование).
3. Добавление меток (MapObject) и обработка кликов по ним.
4. Получение текущего местоположения пользователя.
5. Поиск мест и построение маршрутов (через Yandex Geocoder API).

Разрешения:

1. ACCESS_FINE_LOCATION / ACCESS_COARSE_LOCATION – для определения местоположения.
2. INTERNET – для загрузки карт и поиска.

Задание

1. Зарегистрироваться в кабинете разработчика Yandex и получить API-ключ для Maps SDK.
2. Создать новый проект и добавить зависимость Yandex Maps: implementation("com.yandex.android:maps.mobile:4.4.0-lite")
3. Добавить разрешения в манифест и API-ключ в <application> тег:

```
<meta-data
  android:name="com.yandex.maps.ApiKey"
  android:value="ВАШ_КЛЮЧ" />
```

1. В макете Activity добавить com.yandex.mapkit.mapview.MapView.
2. Инициализировать MapKit в onCreate() (вызов MapKitFactory.initialize(context)) и MapKitFactory.getInstance().
3. Реализовать следующий функционал:
    - При открытии карта центрируется на местоположении пользователя (запросить разрешения).

- Добавить долгое нажатие на карту – установка метки в этой точке.
- Добавить поиск по адресу (простой EditText и кнопка) – найти введенный адрес через Yandex Geocoder и переместить камеру.

4. Добавить кнопку для смены типа карты (схема/спутник).
5. Реализовать кластеризацию меток при добавлении большого количества точек.

Контрольные вопросы

1. Как получить текущие координаты устройства?
2. В чем разница между ACCESS_FINE_LOCATION и ACCESS_COARSE_LOCATION?
3. Что такое кластеризация меток и для чего она нужна?
4. Как обработать ситуацию, если пользователь запретил доступ к геолокации?
5. Какие ограничения существуют у бесплатного тарифа Yandex Maps SDK?

Лабораторная работа №15
«Работа с Bluetooth-сканером»

Теоретическая часть

Bluetooth в Android делится на два типа:

1. Classic Bluetooth – для потоковой передачи данных (аудио, файлы).
2. BLE (Bluetooth Low Energy) – для работы с датчиками, сканерами штрих-кодов, умными часами. Энергоэффективен.

Работа с BLE строится по схеме GATT (Generic Attribute Profile):

1. Device discovery – поиск устройств (сканирование).
2. Gatt Server / Client – устройство может быть сервером (предоставляет данные) или клиентом (читает данные).
3. Services и Characteristics: Сервисы – это логические группы данных. Характеристики – это сами данные (температура, показания сканера). Каждая характеристика имеет UUID.

Разрешения:

1. BLUETOOTH_SCAN, BLUETOOTH_CONNECT, ACCESS_FINE_LOCATION (для Android 12+).
2. ACCESS_FINE_LOCATION требуется для сканирования, так как BLE-устройства могут использоваться для определения местоположения.

Задание

1. Создать новый проект с кнопками: "Сканировать устройства", "Подключиться", "Начать считывание".
2. Добавить необходимые разрешения в манифест и запросить их в рантайме.
3. Инициализировать BluetoothManager и получить BluetoothAdapter.
4. Реализовать сканирование BLE-устройств:
    - Использовать BluetoothLeScanner и ScanCallback.
    - Отображать найденные устройства в RecyclerView (название и MAC-адрес).

5. При выборе устройства из списка – подключиться к нему с помощью connectGatt().
6. После подключения:
    - Обнаружить сервисы discoverServices().
    - Найти сервис и характеристику для передачи данных (обычно для сканеров штрих-кодов это UUID: 0000ffe1-0000-1000-8000-00805f9b34fb или аналогичный – зависит от модели сканера).
    - Включить уведомления (setCharacteristicNotification) на этой характеристике, чтобы получать данные при сканировании.
7. Реализовать обработчик onCharacteristicChanged для приема считанных штрих-кодов и отображения их на экране.
8. Добавить кнопку "Отключиться", корректно закрывающую GATT-соединение.

Контрольные вопросы

1. В чем разница между Classic Bluetooth и BLE?
2. Почему для сканирования BLE требуется разрешение на геолокацию?
3. Что такое UUID, сервис и характеристика в BLE?
4. Как подписаться на изменения характеристики (уведомления)?
5. Что произойдет с подключением при выходе из приложения? Как это правильно обработать?

Лабораторная работа №16
«Интеграция push-уведомлений (Firebase Cloud Messaging, RuStore Push)»

Теоретическая часть

Push-уведомления – это сообщения, которые отправляются сервером на мобильное устройство даже когда приложение закрыто.

Основные игроки на Android:

1. FCM (Firebase Cloud Messaging) – глобальный стандарт от Google. Требует сервисы Google Play на устройстве.
2. RuStore Push – решение от VK для устройств без Google Play (Huawei, некоторые китайские бренды). Работает через RuStore.
3. HMS Push (Huawei Mobile Services) – для устройств Huawei.

Принцип работы FCM:

1. Приложение получает токен устройства (Registration Token) от FCM.
2. Токен отправляется на ваш сервер.
3. Сервер отправляет запрос к FCM с токеном и данными сообщения.
4. FCM доставляет уведомление на устройство.

Типы сообщений:

1. Notification messages – автоматически отображаются в системном трее (когда приложение в фоне).
2. Data messages – обрабатываются приложением (кодом) в любом состоянии.

RuStore Push – альтернатива для РФ. Требует регистрации в консоли RuStore, аналогичный принцип работы, но не зависит от Google Play Services.

Задание

1. Создать новый проект или использовать существующий.
2. Настройка Firebase:

- Зарегистрировать проект в console.firebase.google.com.
- Добавить Android-приложение, указать package name.
- Скачать google-services.json и поместить в папку app/.
- В корневой build.gradle добавить плагин Google Services.

3. Интеграция FCM:
   - Добавить зависимость implementation("com.google.firebase:firebase-messaging-ktx:23.4.0").
   - Создать сервис MyFirebaseMessagingService, наследующий от FirebaseMessagingService().
   - Переопределить onNewToken() – здесь токен отправляется на ваш сервер (или логируется для теста).
   - Переопределить onMessageReceived() – обработка входящих сообщений.
4. Запрос разрешения: Для Android 13+ нужно запрашивать разрешение POST_NOTIFICATIONS.
5. Тестирование:
   - Запустить приложение, получить токен из логов.
   - Отправить тестовое уведомление через консоль Firebase (Cloud Messaging).
6. RuStore Push (дополнительно):
   - Зарегистрироваться в консоли RuStore.
   - Интегрировать SDK ru.rustore.sdk:pushkit:...
   - Реализовать аналогичный сервис для получения токена RuStore.

Контрольные вопросы

1. В чем разница между Notification и Data сообщениями в FCM?
2. Что такое токен устройства и как часто он обновляется?
3. Почему на устройствах без Google Play FCM не работает?
4. Какие разрешения нужны для push-уведомлений на Android 13+?
5. Как обработать нажатие пользователя на уведомление?

Лабораторная работа №17
«Создание CI-сборки на GitHub Actions»

Теоретическая часть

CI/CD (Continuous Integration / Continuous Delivery) – практика автоматизации сборки, тестирования и доставки кода.

GitHub Actions – платформа автоматизации, встроенная в GitHub. Позволяет создавать рабочие процессы (workflows), которые запускаются по событиям (push, pull request, по расписанию).

Типичные задачи для CI в Android:

1. Автоматическая сборка APK при каждом пуше в ветку develop.
2. Прогон unit-тестов.
3. Проверка стиля кода (ktlint, detekt).
4. Автоматическая выкладка в тестовые каналы (Firebase App Distribution).

Компоненты GitHub Actions:

1. Workflow – YAML-файл в папке .github/workflows.
2. Job – набор шагов, выполняющихся на одном виртуальном сервере (runner).
3. Step – отдельная команда или действие.
4. Action – готовый блок для переиспользования (например, actions/checkout).

Задание

1. Создать репозиторий на GitHub и загрузить туда проект из лабораторных работ.
2. Создать файл .github/workflows/build.yml.
3. Написать workflow, который:
    - Запускается по событию push в ветку main и по созданию pull_request.
    - Использует ubuntu-latest в качестве runner'a.
    - Устанавливает JDK 17.
    - Кеширует зависимости Gradle для ускорения сборки.
    - Выдает права на выполнение (chmod +x gradlew).

- Запускает сборку debug APK (./gradlew assembleDebug).
- Запускает тесты (./gradlew test).

4. Дополнительно: Настроить загрузку артефактов (APK) после сборки.

– name: Upload APK
uses: actions/upload-artifact@v4
with:
name: app-debug
path: app/build/outputs/apk/debug/

5. Секреты (Secrets): Научиться добавлять в репозиторий секретные переменные (например, ключ подписи) через Settings -> Secrets and variables.
6. Дополнительно (сложный уровень): Настроить подпись релизного APK с использованием зашифрованного Keystore и секретов GitHub.

Контрольные вопросы

1. Что такое CI/CD и зачем это нужно?
2. Какие события могут запускать workflow в GitHub Actions?
3. Как безопасно хранить Keystore и пароли в GitHub Actions?
4. Чем assembleDebug отличается от assembleRelease?
5. Что такое артефакты сборки и как их сохранить?

Лабораторная работа №18
«Адаптация под планшеты (multi-pane интерфейс)»

Теоретическая часть

Планшеты имеют больший экран, и интерфейс, скопированный со смартфона, выглядит на них неэффективно (растянутые списки, пустое пространство).

Подходы к адаптации

1. Альтернативные ресурсы (ресурсные квалификаторы):
   - layout-sw600dp/ – папка с макетами для экранов шириной от 600dp (7-дюймовые планшеты).
   - layout-sw720dp/ – для 10-дюймовых планшетов.
   - values-sw600dp/ – альтернативные размеры шрифтов, отступов.
2. Multi-pane (двухпанельный интерфейс):
   - Master-Detail Flow: Слева список (master), справа детали (detail) выбранного элемента.
   - Реализуется через SlidingPaneLayout или androidx.fragment.app.FragmentContainerView.
3. Адаптивные Layout-компоненты:
   - ConstraintLayout – позволяет создавать гибкие интерфейсы.
   - Flow (из ConstraintLayout) – для автоматического переноса элементов.

Задание

1. Взять проект со списком и деталями (например, из ЛР №12 – список постов и детали поста).
2. Создать две версии макета для главной Activity:
   - Для телефонов (layout/activity_main.xml): Один FragmentContainerView, в котором будет либо список, либо детали (в зависимости от навигации).
   - Для планшетов (layout-sw600dp/activity_main.xml): Два FragmentContainerView рядом (слева 30 % ширины, справа 70 %).

3. Использовать SlidingPaneLayout для более гибкого поведения:

```
<androidx.slidingpanelayout.widget.SlidingPaneLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <androidx.fragment.app.FragmentContainerView
      android:id="@+id/list_pane"
      android:layout_width="280dp"
      android:layout_height="match_parent" />

   <androidx.fragment.app.FragmentContainerView
      android:id="@+id/detail_pane"
      android:layout_width="match_parent"
      android:layout_height="match_parent" />

</androidx.slidingpanelayout.widget.SlidingPaneLayout>
```

4. В коде Activity определить, доступна ли панель деталей (открыта ли она):
    - Если детали видны (планшетная ориентация), при клике на элемент списка загружать фрагмент деталей в правую панель.
    - Если детали не видны (телефонная ориентация), запускать новую Activity или заменять фрагмент.
5. Протестировать на эмуляторе телефона и планшета (можно создать виртуальное устройство Pixel C или Nexus 10).
6. Дополнительно: Использовать CurrentScreenWidth для программного определения ширины и выбора поведения.

Контрольные вопросы

1. Что означает квалификатор sw600dp?
2. Как проверить в коде, является ли устройство планшетом?
3. В чем преимущество SlidingPaneLayout перед двумя FrameLayout?
4. Как сохранить состояние выбранного элемента при повороте экрана на планшете?

5. Какие еще квалификаторы ресурсов можно использовать для адаптации?

Список литературы

1. Филлипс, Б. Android. Программирование для профессионалов / Б. Филлипс, К. Стюарт, К. Марсикано. – 4-е изд. – Санкт-Петербург : Питер, 2021. – 688 с. – (Серия «Для профессионалов»). – ISBN 978-5-4461-1688-5.
2. Смарт, Д. Android. Разработка приложений для чайников / Д. Смарт. – Москва : Диалектика, 2020. – 368 с. – ISBN 978-5-907144-67-9.
3. Мэрфи, М. Android. Программирование для профессионалов / М. Мэрфи. – Москва : Эксмо, 2019. – 512 с. – (Серия «Мировой компьютерный бестселлер»). – ISBN 978-5-04-100524-5.
4. Доусон, С. Программируем для Android / С. Доусон. –Москва : Вильямс, 2019. – 272 с. – ISBN 978-5-907144-98-3.
5. Гриффитс, Д. Head First. Программирование для Android / Д. Гриффитс. Дон Гриффитс – 2-е изд. – Санкт-Петербург : Питер, 2022. – 912 с. – ISBN 978-5-4461-1905-3.
6. Горнаков, С. Г. Программирование мобильных приложений под Android / С. Г. Горнаков. – Москва : ДМК Пресс, 2020. – 368 с. – ISBN 978-5-97060-852-8.
7. Комлев, Н. Ю. Разработка приложений под Android. Основы Kotlin / Н. Ю. Комлев. – Москва : Горячая линия – Телеком, 2021. – 246 с. – ISBN 978-5-9912-0932-4.
8. Майер, Р. Android 4. Программирование приложений для планшетов и смартфонов / Р. Майер. – Москва : Эксмо, 2018. – 816 с. – (Серия «Мировой компьютерный бестселлер»). – ISBN 978-5-699-48759-2.

Оглавление