

Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет имени Евфросинии Полоцкой»



К. Я. Раханов
Д. М. Васильева

ПРОГРАММИРОВАНИЕ МОБИЛЬНЫХ СИСТЕМ

Методические указания
к лабораторным работам
для студентов специальностей
6-05-0611-05, 6-05-0612-01, 6-05-0611-01

Текстовое электронное издание

Новополоцк
Полоцкий государственный университет
имени Евфросинии Полоцкой
2026

Об издании – [1](#), [2](#)

1 – дополнительный титульный экран – сведения об издании

УДК 004(075.8)(0.034)

Одобрено и рекомендовано к изданию
советом факультета информационных технологий (протокол № 5 от 20.05.2026)

Кафедра вычислительных систем и сетей

Предназначены для студентов специальностей 6-05-0611-05 «Компьютерная инженерия» с профилизацией «Вычислительные машины, системы и сети», 6-05-0612-01 «Программная инженерия» и «Программирование для мобильных устройств» для студентов специальности 6-05-0611-01 «Информационные системы и технологии» с профилизацией «Информационные системы и технологии в игровой индустрии».

© Раханов К. Я., Васильева Д. М., 2026
© Полоцкий государственный университет
имени Евфросинии Полоцкой, 2026

2 – дополнительный титульный экран – производственно-технические сведения

Для создания текстового электронного издания «Программирование мобильных систем» К. Я. Раханова, Д. М. Васильевой использованы текстовый процессор Microsoft Word и программа Adobe Acrobat XI Pro для создания и просмотра электронных публикаций в формате PDF.

РАХАНОВ Константин Яковлевич
ВАСИЛЬЕВА Диана Михайловна

ПРОГРАММИРОВАНИЕ МОБИЛЬНЫХ СИСТЕМ

Методические указания
к лабораторным работам
для студентов специальностей
6-05-0611-05, 6-05-0612-01, 6-05-0611-01

Редактор *Т. А. Дарьянова*

Подписано к использованию 15.10.2026.
Объем издания 5,07 Мб. Заказ 432.

Издатель и полиграфическое исполнение:
учреждение образования «Полоцкий государственный университет
имени Евфросинии Полоцкой».

Свидетельство о государственной регистрации
издателя, изготовителя, распространителя печатных изданий
№ 1/305 от 22.04.2014.

ЛП № 02330/278 от 27.05.2004.

211440, ул. Блохина, 29,
г. Новополоцк,
Тел. 8 (0214) 59-95-41, 59-95-44
<http://www.psu.by>

Содержание

ВВЕДЕНИЕ.....	5
ЛАБОРАТОРНАЯ РАБОТА 1	
Создание приложений с элементами пользовательского интерфейса в Android Studio.....	6
ЛАБОРАТОРНАЯ РАБОТА 2	
Создание многоэкранного Android приложения с помощью намерений.	
Работа с различными видами уведомлений	36
ЛАБОРАТОРНАЯ РАБОТА 3	
Вывод, обработка и хранение данных в локальной базе данных	71
ЛАБОРАТОРНАЯ РАБОТА 4	
Использование готовых backend-решений Firebase в мобильных приложениях.....	98
ЛАБОРАТОРНАЯ РАБОТА 5	
Работы с сообщениями через Firebase Cloud Messaging в мобильных приложениях.....	133
ПРИЛОЖЕНИЕ.....	142
Литература	148

ВВЕДЕНИЕ

Данные методические указания являются руководством к выполнению лабораторных работ по курсам «Программирование мобильных систем» для студентов специальностей 6-05-0611-05 «Компьютерная инженерия» с профилизацией «Вычислительные машины, системы и сети», 6-05-0612-01 «Программная инженерия» и «Программирование для мобильных устройств» для студентов специальности 6-05-0611-01 «Информационные системы и технологии» с профилизацией «Информационные системы и технологии в игровой индустрии». Содержание работ соответствует учебной программе, разработанной на основе образовательного стандарта ОСВО 6-05-0612-01-2023 и учебных планов ФИТ (рег. № 20-23/уч., 04-23/уч.з., 05-23/уч.з. от 04.04.2023).

Основная *цель* лабораторного практикума – формирование у студентов теоретических знаний и закрепление практических навыков, необходимых для профессиональной разработки приложений под мобильную операционную систему Android.

В ходе выполнения лабораторных работ студенты решают ключевые *задачи* дисциплины:

- изучают архитектуру мобильных платформ и жизненный цикл приложений;
- осваивают основные элементы пользовательского интерфейса и методы работы с данными (файлы, базы данных, настройки);
- знакомятся с программными интерфейсами для телефонии, геолокации и картографических сервисов;
- приобретают навыки оптимизации и тестирования программного кода.

В результате студент должен овладеть языком программирования Kotlin, инструментарием Android SDK, языком разметки XML и навыками использования элементов Jetpack Compose, а также интеграции в свои приложения готовых решений Firebase, что позволит ему проектировать и создавать конкурентоспособные мобильные приложения. Знания и умения, полученные в ходе лабораторных работ, могут быть использованы при выполнении индивидуальных заданий и в дипломном проектировании.

ЛАБОРАТОРНАЯ РАБОТА 1

Создание приложений с элементами пользовательского интерфейса в Android Studio

Цель работы: получить навыки создания мобильных приложений под ОС Android, компиляции кода с использованием интегрированной среды разработки Android Studio.

Теоретическая часть

I. Среда программирования Android Studio.

Android Studio – среда разработки мобильных приложений под операционную систему Android, основанная на интегрированной среде разработки программного обеспечения IntelliJ IDEA. Android Studio содержит встроенные инструменты для создания и отладки мобильных приложений [1].

Дополнительно ко всем возможностям в Android Studio реализованы:

- поддержка системы автоматической сборки Gradle;
- уникальная система рефакторинга кода (набор инструментов, методов и правил, предназначенных для улучшения структуры, читаемости и поддерживаемости программного кода без изменения его внешнего поведения);
- инструменты для поиска и устранения различных проблем;
- окно предварительного просмотра, которое показывает запущенное приложение сразу на нескольких устройствах и в реальном времени;
- поддержка облачной платформы Google Cloud Platform.

II. Интерфейс среды разработки.

После завершения установки (см. Приложение 1) можно переходить к изучению интерфейса среды разработки Android Studio (рисунок 1.1) и этапу создания первого приложения для операционной системы Android. Для этого запустим Android Studio.

На приветственном экране слева располагается боковое меню, состоящее:

- 1) из Projects – список уже существующих проектов;
- 2) Customize – настройка внешнего вида и поведения Android Studio (темы, шрифт кода и горячие клавиши);
- 3) Plugins – установка и управление плагинами;
- 4) Learn – уроки по разработке приложений в Android Studio.

В центре экрана три кнопки:

- New Project – создание нового проекта;
- Open – открытие уже существующего проекта;
- Clone Repository – клонирование проекта из репозитории (Git, GitHub).

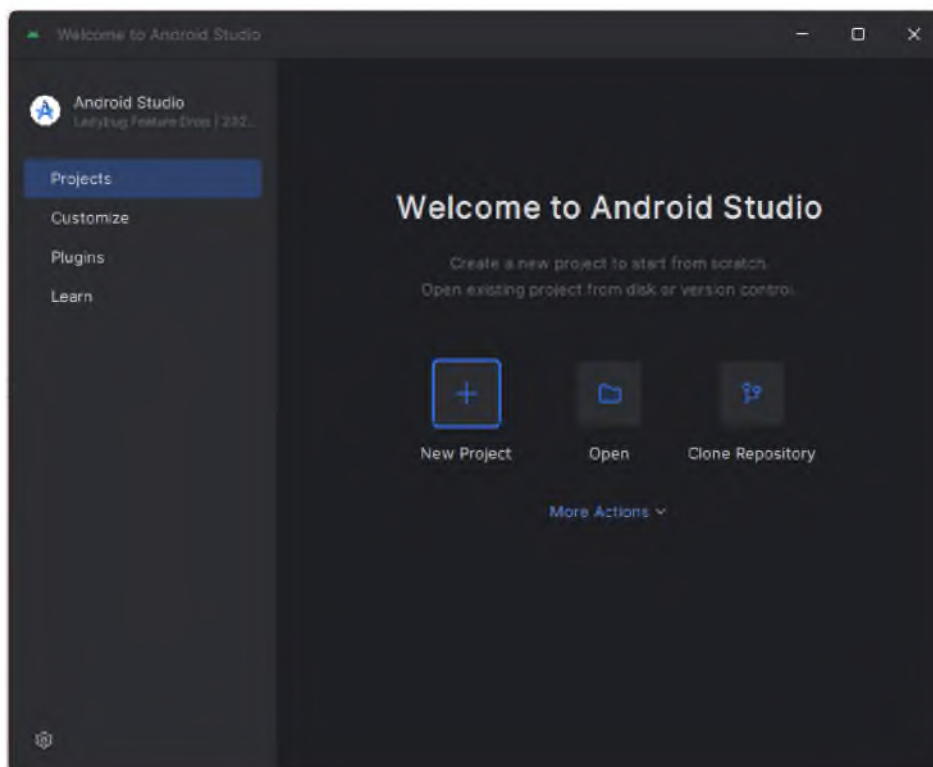


Рисунок 1.1. – Создание нового проекта

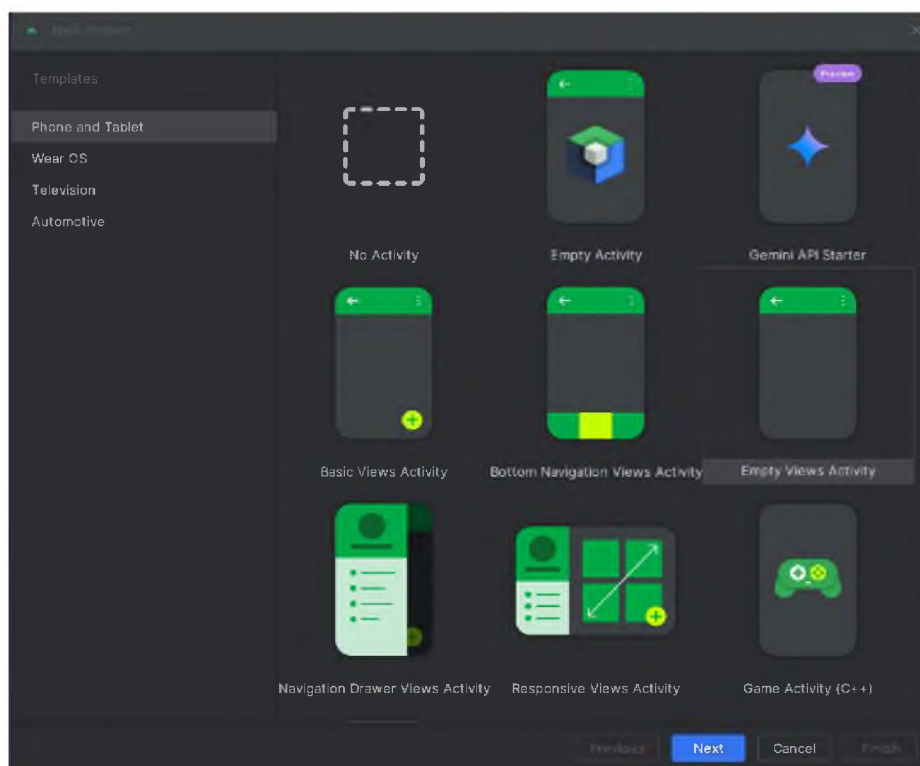


Рисунок 1.2. – Выбор шаблона проекта

При создании нового проекта Android Studio предложит выбрать шаблон проекта. Android Studio предоставляет ряд шаблонов для различных ситуаций (см. рисунок 1.2).

- No Activity – проект без активности;
- Empty Activity – проект с пустой активностью;
- Basic Views Activity – проект с активностью, на которой уже расположены кнопка и текстовое поле;
- Bottom Navigation Views Activity – проект с активностью, на которой расположена навигационная панель с кнопками;
- Empty Views Activity – Empty Activity, но с использованием XML-разметки страницы;
- Navigation Drawer Views Activity – проект с активностью, на которой расположено боковое меню;
- Responsive Views Activity – проект с адаптивным интерфейсом под разные размеры экранов.

Для создания первого приложения на языке программирования Kotlin чаще используют шаблоны Empty Activity и Empty Views Activity.

После выбора шаблона проекта и нажатия на кнопки Next отобразится окно настроек проекта. Необходимо определить следующие поля:

1. Name. Название проекта.
2. Package name. Имя пакета. Два приложения могут иметь одинаковые названия, но разные имена пакетов, в таком случае Android Studio будет считать эти приложения разными и сможет установить их рядом.
3. Save location. Расположение проекта.
4. Language. Язык разработки (если шаблон поддерживает язык разработки Java и Kotlin, если данного поля нет, то автоматически выбирается язык Kotlin).
5. Minimum SDK. Минимальный SDK – самая низкая версия Android, которую будет поддерживать приложение.
6. Build configuration language. Язык конфигурации сборки.

После нажатия кнопки Finish настройка проекта завершается и Android Studio создает новый проект (рисунок 1.3).

Интерфейс среды разработки представлен на рисунке 1.4.

Наиболее важным является отображение структуры проекта (боковое меню слева, рисунок 1.4, п. 1) и окно редактора кода (рисунок 1.4, п. 2). В нижней области, в зависимости от выбранной опции, будут отображаться: окно терминала, логи сборки проекта, логи запущенного приложения, выводы приложения (рисунок 1.4, п. 3).

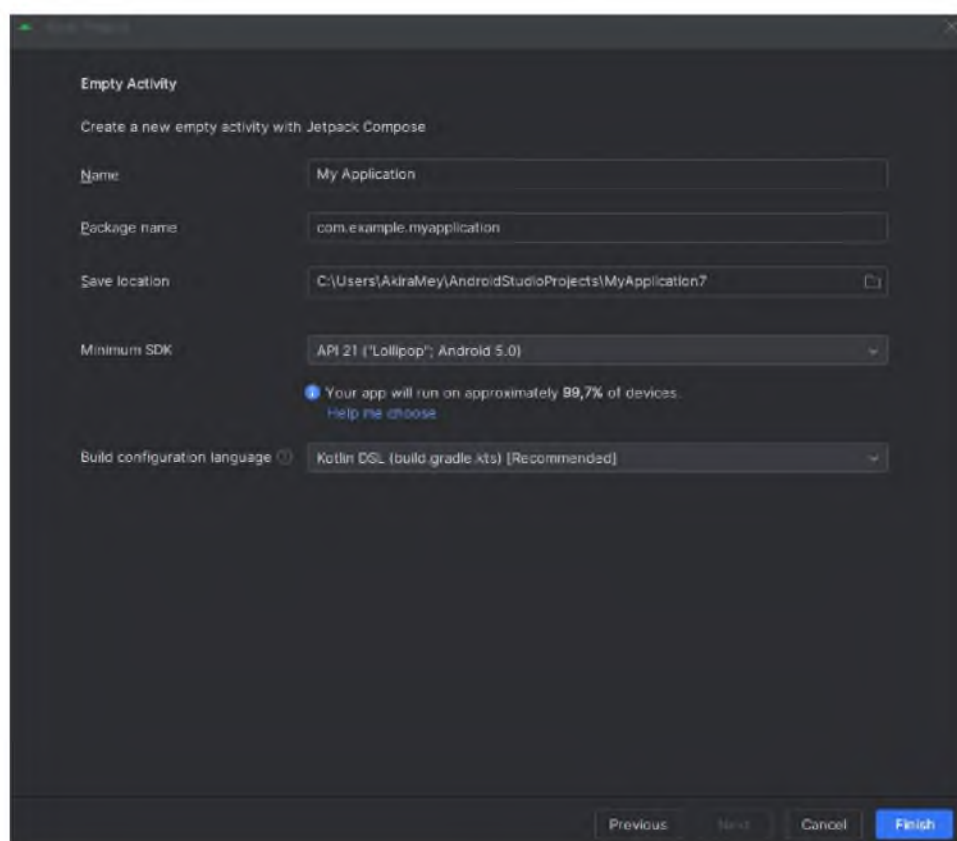


Рисунок 1.3. – Окно настроек нового проекта

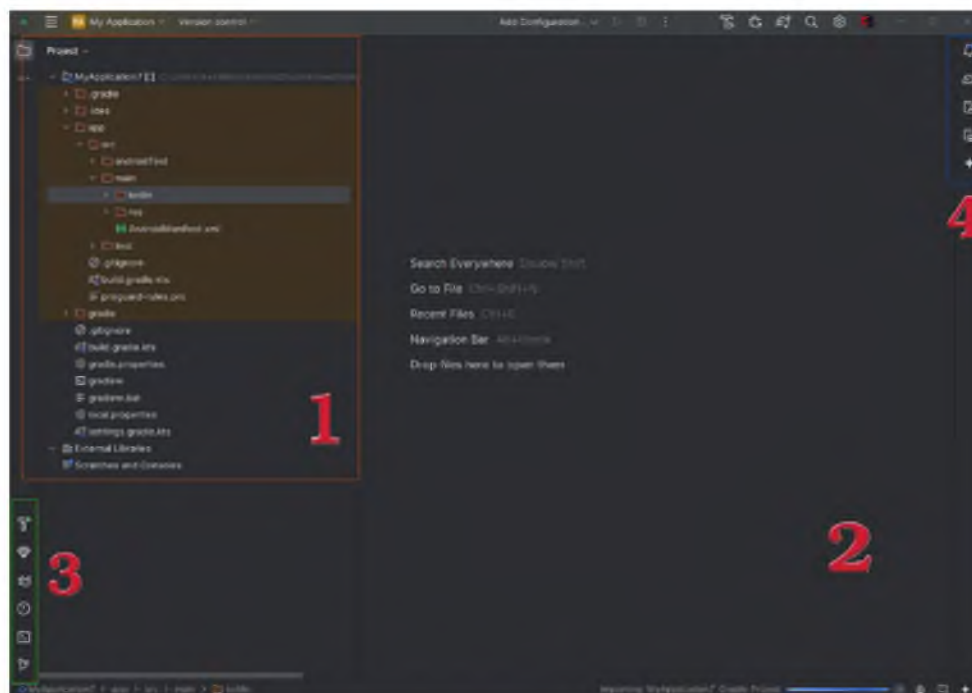


Рисунок 1.4. – Интерфейс среды разработки

В Android Studio структура проекта состоит:

1) из **manifests**.

AndroidManifest.xml: один из самых важных файлов в проекте. Он содержит информацию о приложении, такую как его компоненты (активности, сервисы, приемники), разрешения, версия SDK и другие метаданные. Этот файл необходим для сборки и запуска приложения;

2) **kotlin**.

– *com.example.myapplication*: основной пакет, где находится исходный код вашего приложения. Здесь вы будете писать код на Kotlin, включая активности, фрагменты, сервисы и другие классы;

– *com.example.myapplication (androidTest)*: каталог предназначен для инструментальных тестов, которые запускаются на реальном устройстве или эмуляторе. Эти тесты проверяют поведение приложения в реальных условиях;

– *com.example.myapplication (test)*: здесь находятся модульные тесты, которые выполняются на локальной машине без необходимости запуска на устройстве. Они используются для тестирования отдельных компонентов приложения;

3) **res**.

Каталог res (сокращение от resources) в проекте Android Studio предназначен для хранения всех ресурсов, которые используются в приложении. Ресурсы – это дополнительные файлы и статический контент, такие как изображения, строки, макеты, стили, анимации и другие элементы, которые необходимы для работы приложения, но не являются частью исходного кода. Основными подкаталогами являются:

– *drawable*: хранит графические ресурсы, такие как изображения, иконки и другие drawable-ресурсы. Они используются для оформления интерфейса приложения;

– *layout*: содержит XML-файлы, которые определяют структуру пользовательского интерфейса. Каждый файл соответствует макету экрана или части экрана;

– *mipmap*: используется для хранения иконок приложения с разным разрешением. Это позволяет приложению использовать оптимальные иконки для различных плотностей экрана;

– *values*: хранит ресурсы, такие как строки, цвета, стили и размеры. Они используются для централизованного управления внешним видом и текстами приложения;

– *xml*: другие XML-файлы, такие как настройки предпочтений или конфигурации;

4) **Gradle Scripts**.

Это набор файлов конфигурации, которые используются системой сборки Gradle для управления процессом сборки, зависимостями и настройками проекта.

Gradle – мощный инструмент автоматизации сборки, который позволяет разработчикам определять, как их приложение должно быть собрано, какие библиотеки и зависимости использовать, а также настраивать различные параметры сборки. Состоит:

- из *build.gradle (Project)*: содержит настройки, которые применяются ко всему проекту, такие как репозитории и зависимости;
- *build.gradle (Module)*: содержит настройки для конкретного модуля приложения, включая зависимости, версии SDK и другие параметры сборки;
- *gradle-wrapper.properties*: определяет версию Gradle, которая будет использоваться для сборки проекта;
- *settings.gradle*: определяет, какие модули включены в проект.

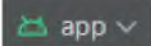
Интерфейс редактора кода содержит множество полезных кнопок и элементов управления, облегчающие работу с кодом и проектом в целом (см. рисунок 1.4, п. 2). Основные элементы панели (рисунок 1.5):

1. Вкладки файлов:

- Pixel 8 API 35 – показывает текущее эмулируемое устройство;
- *activity_main.xml* – вкладка с XML-макетом;
- *MainActivity.kt* – вкладка с Kotlin-кодом.

Служат для переключения между открытыми файлами и эмулируемыми устройствами.

2. Панель инструментов над кодом:

-  *app* – запуск и отладка проекта;
-  запуск проекта без отладки;
-  отладка проекта;
-  кнопка для быстрой сборки проекта, которая перекомпилирует только измененные файлы;
-  кнопка позволяет внести изменения перезапустив активность, но без полного перезапуска приложения, работает только при подключенном устройстве и debug-режиме;
-  кнопка позволяет внести мелкие изменения (цвет/текст кнопки) без перезапуска активности и приложения, работает только при подключенном устройстве и debug-режиме;
-  кнопка позволяет подключить отладчик (debugger) к уже запущенному процессу вашего приложения на устройстве или эмуляторе;

-  кнопка синхронизирует настройки проекта с файлами build.gradle, загружая новые зависимости и применяя изменения конфигурации;
-  кнопка для поиска по проекту;
-  кнопка загрузки обновлений для IDE и настроек проекта.

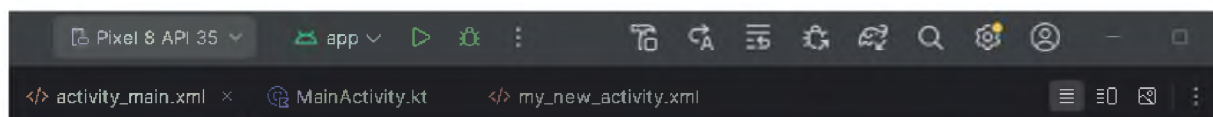


Рисунок 1.5. – Интерфейс редактора кода

III. Jetpack Compose.

Jetpack Compose – это современный набор инструментов Google для создания собственного пользовательского интерфейса Android. Это часть набора библиотек Jetpack, цель которого – следовать передовым практикам, сокращать количество шаблонного кода и писать код, который одинаково работает на всех версиях Android и устройствах.

Для создания проекта в Android Studio с использованием Jetpack Compose необходимо:

1. Создать новый проект с выбором шаблона Empty Activity или Empty Activity (Compose).
2. Android Studio (в последних версиях) автоматически генерирует:
 - правильную структуру проекта;
 - настройки Gradle для Compose.

Для проверки, файл должен содержать следующие строки:

```
android {
    buildFeatures {
        compose = true
    }
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.3" // Для Kotlin 1.9.20
    }
}

dependencies {
    // BOM-версионирование
    implementation(platform("androidx.compose:compose-bom:2024.05.00"))
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.material3:material3")
    implementation("androidx.activity:activity-compose:1.8.2")
}
```

```
// Инструменты разработчика
debugImplementation("androidx.compose.ui:ui-tooling")
implementation("androidx.compose.ui:ui-tooling-preview")
}
```

– базовый Compose-код в MainActivity.kt. Основной код в MainActivity.kt:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyAppTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    MyAppTheme {
        Greeting("Preview")
    }
}
```

– настройка темы. В файле ui/theme/Theme.kt:

```
private val DarkColorScheme = darkColorScheme(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)

@Composable
```

```
fun MyAppTheme(  
    content: @Composable () -> Unit  
) {  
    MaterialTheme(  
        colorScheme = DarkColorScheme,  
        typography = Typography,  
        content = content  
    )  
}
```

IV. Элементы Jetpack Compose.

Основой пользовательского интерфейса в Jetpack Compose являются компоненты, которые оформлены в виде функции с аннотацией `@Composable`. При вызове `@Composable`-компонента обычно передаются некоторые данные и набор свойств, которые определяют отображение и поведение определенной части пользовательского интерфейса. Эти функции не возвращают значения в традиционном смысле функции Kotlin, а вместо этого определяют элементы пользовательского интерфейса для рендеринга.

`@Composable`-функции могут вызывать другие подобные компоненты для создания иерархии компонентов. При разработке приложений с помощью Compose можно как определять свои собственные компоненты, так и использовать встроенные. Типичный пользовательский интерфейс на основе Compose обычно состоит из комбинации встроенных и кастомных `@Composable`-компонентов.

Встроенные компоненты, которые по умолчанию входят в состав Compose, можно разделить на три категории: Layout, Foundation и Material Design.

В категорию *Layout* входят компоненты, которые определяют макет приложения, позволяют определить расположение других компонентов на экране. В частности, это следующие компоненты:

- `Box` – контейнер, позволяющий размещать дочерние элементы друг над другом;
- `BoxWithConstraints` – расширяет `Box`, предоставляя доступ к макс./мин. ширине и высоте контейнера;
- `Column` – контейнер, размещающий элементы вертикально – сверху вниз (рисунок 1.5);
- `ConstraintLayout` – контейнер, позволяющий создавать сложные макеты с взаимосвязанными элементами;
- `Row` – контейнер, располагающий элементы горизонтально – слева направо (рисунок 1.6).

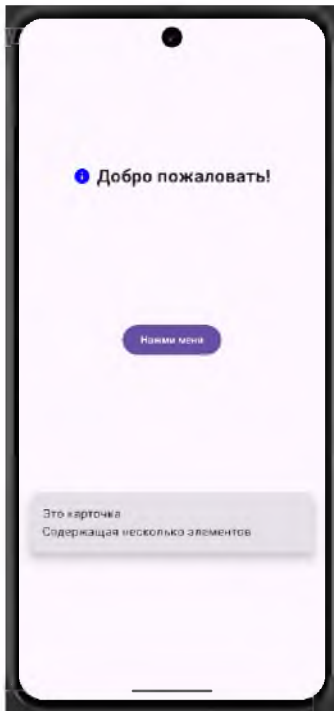


Рисунок 1.5. – Размещение трех элементов при помощи Column

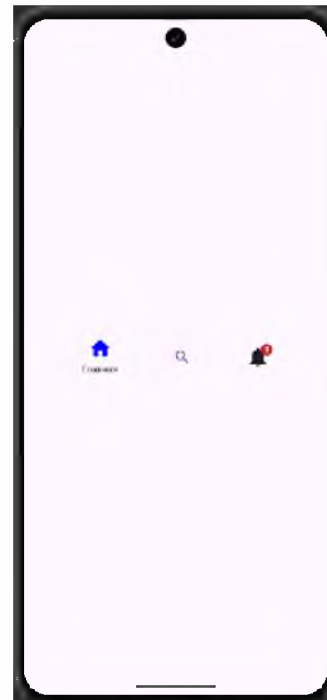


Рисунок 1.6. – Размещение трех элементов при помощи Row

Листинг 1.1. – Размещение трех элементов при помощи Column

```
@Composable
fun ColumnExample() {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.SpaceEvenly,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // Элемент 1 - Текст с иконкой
        Row(verticalAlignment = Alignment.CenterVertically) {
            Icon(
                imageVector = Icons.Default.Info,
                contentDescription = "Info",
                tint = Color.Blue
            )
            Spacer(modifier = Modifier.width(8.dp))
            Text(
                text = "Добро пожаловать!",
                fontSize = 24.sp,
                fontWeight = FontWeight.Bold
            )
        }
    }
}
```

```

// Элемент 2 - Кнопка
Button(
    onClick = { /* Действие при нажатии */ },
    colors = ButtonDefaults.buttonColors(
        containerColor = MaterialTheme.colorScheme.primary
    )
){
    Text("Нажми меня")
}

// Элемент 3 - Карточка
Card(
    modifier = Modifier.fillMaxWidth(),
    elevation = CardDefaults.cardElevation(8.dp)
){
    Column(
        modifier = Modifier.padding(16.dp)
    ){
        Text("Это карточка")
        Text("Содержащая несколько элементов")
    }
}
}
}

```

Листинг 1.2. – Размещение трех элементов при помощи Row

```

@Composable
fun RowExample() {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp),
        verticalAlignment = Alignment.CenterVertically, // Вертикальное выравнивание
        horizontalArrangement = Arrangement.SpaceBetween // Распределение пространства
    ){
        Button(onClick = { /* Действие */ }) {
            Text("Кнопка 1")
        }
        Button(onClick = { /* Действие */ }) {
            Text("Кнопка 2")
        }
    }
}

```

Компоненты *Foundation* – это набор минимальных компонентов, которые обеспечивают базовую функциональность пользовательского интерфейса. Например:

- `BaseTextField` – базовый компонент для создания кастомных текстовых полей (низкоуровневый аналог `TextField`);
- `Canvas` – холст для кастомной векторной графики;
- `Image` – компонент для отображения изображений;
- `LazyColumn` – вертикальный скроллящийся список;
- `LazyRow` – горизонтальный аналог `LazyColumn`;
- `Shape` – абстракция для определения форм компонентов (прямоугольник, круг, скругленные углы);
- `Text` – основной компонент для отображения текста в виде текстовой строки с поддержкой стилизации (шрифт, цвет и т.д.). Использование текстовой строки продемонстрировано на рисунке 1.5.

Листинг 1.3. – Использование текстовой строки

```
@Composable
fun TextExample() {
    // Text - основной компонент для отображения текста
    Text(
        text = "Привет, Jetpack Compose!",
        modifier = Modifier.padding(16.dp),
        color = Color.Blue, // Цвет текста
        fontSize = 20.sp, // Размер шрифта
        fontWeight = FontWeight.Bold, // Жирность
        fontStyle = FontStyle.Italic, // Стил (курсив)
        textAlign = TextAlign.Center, // Выравнивание текста
        overflow = TextOverflow.Ellipsis, // Обрезка с многоточием
        maxLines = 1 // Максимальное количество строк
    )
}
```

Компоненты *Material Design* разработаны так, чтобы они соответствовали рекомендациям визуальному стилю Google Material. Например:

- `AlertDialog` – диалоговое окно с заголовком, текстом и кнопками для подтверждения/отмены действий;
- `ViewModel` – компонент для хранения и обработки данных, а также отображения их на экране;
- `Button` – кликабельная кнопка;
- `Card` – контейнер с закругленными углами и тенью для группировки связанного контента;

- CircularProgressIndicator – круговой индикатор загрузки (спиннер);
 - DropdownMenu – выпадающее меню;
 - Checkbox – флажок для выбора одного или нескольких вариантов;
 - FloatingActionButton – плавающая кнопка действия;
 - LinearProgressIndicator – горизонтальный индикатор загрузки;
 - ModalDrawer – боковое выдвигающееся меню;
 - RadioButton – переключатель для выбора одного варианта из нескольких;
 - Scaffold – каркас приложения с местами для TopAppBar, BottomBar, FAB и основного контента;
 - Slider – ползунок для выбора значения в диапазоне;
 - Snackbar – временное уведомление в нижней части экрана с кратким сообщением;
 - Switch – переключатель On/Off для изменения состояния;
 - TextField – поле для ввода и редактирования текста;
 - TopAppBar – верхняя панель приложения (шапка) с заголовком и действиями;
 - BottomNavigation – нижняя навигационная панель с иконками и подписями.
- Button – элемент графического интерфейса в виде стандартной кнопки. Использование кнопки продемонстрировано на рисунке 1.5.

Листинг 1.4. – Использование кнопки

```
@Composable
fun ButtonExample() {
    var clickCount by remember { mutableStateOf(0) }

    // Button - интерактивный элемент для действий пользователя
    Button(
        onClick = { clickCount++ }, // Обработчик нажатия
        modifier = Modifier.padding(16.dp),
        enabled = true, // Активна ли кнопка
        colors = ButtonDefaults.buttonColors(
            containerColor = Color.Blue, // Цвет фона
            contentColor = Color.White // Цвет содержимого
        ),
        border = BorderStroke(2.dp, Color.Black), // Граница
        shape = RoundedCornerShape(8.dp) // Скругление углов
    ) {
        // Содержимое кнопки (может быть любым Composable)
        Row {
            Icon(
                imageVector = Icons.Default.Favorite,
                contentDescription = "Иконка"
            )
        }
    }
}
```

```

    )
    Spacer(modifier = Modifier.width(8.dp))
    Text("Нажато $clickCount раз")
  }
}
}

```

TextField – элемент графического интерфейса в виде поля для ввода текста. Использование TextField продемонстрировано на рисунке 1.7.

Листинг 1.5. – Использование элемента TextField

```

@Composable
fun TextFieldApp() {
    var text by remember { mutableStateOf("") } // Состояние для хранения текста

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // Основной TextField
        TextField(
            value = text,
            onChange = { newText -> text = newText },
            label = { Text("Введите текст") },
            modifier = Modifier.fillMaxWidth(),
            singleLine = true
        )

        Spacer(modifier = Modifier.height(16.dp))

        // Показ введенного текста
        Text(
            text = if (text.isNotEmpty()) "Вы ввели: $text" else "Введите текст выше",
            style = MaterialTheme.typography.bodyLarge
        )
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewTextFieldApp() {
    MaterialTheme {
        TextFieldApp()
    }
}

```

```
}  
}
```

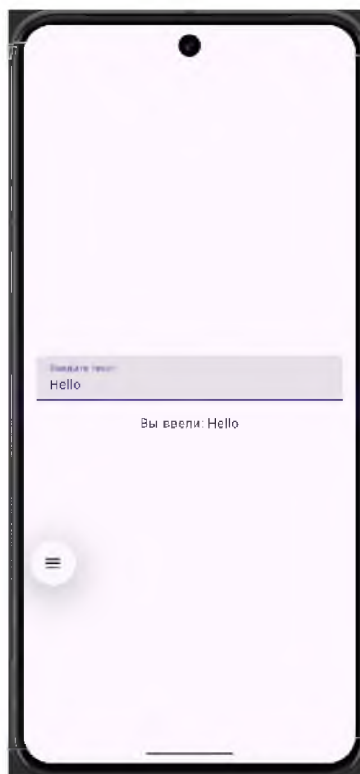


Рисунок 1.7. – Использование элемента TextField

V. Основы языка Kotlin.

Синтаксис Kotlin отличается простотой и читаемостью, делая его привлекательным для разработчиков, уже знакомых с Java и другими современными языками программирования. Основные аспекты синтаксиса Kotlin включают:

1. Объявление переменных: в Kotlin используются ключевые слова `val` для объявления неизменяемых переменных (аналогично `final` в Java) и `var` для изменяемых. Одна из ключевых особенностей Kotlin – это возможность опускать явное указание типа данных при объявлении переменных.

Листинг 1.6. – Объявление переменных в коде

```
val name1: String = "Kotlin" // Неизменяемая переменная  
var version1: Int = 1 // Изменяемая переменная  
val name2 = "Kotlin" // Тип String выдается автоматически  
var version2 = 1 // Тип Int выдается автоматически
```

2. В Kotlin есть несколько видов циклов, среди которых наиболее часто используются `for`, `while` и `do-while`. Разберем каждый из них: цикл `for` в Kotlin часто

используется для итерации по диапазонам, массивам, коллекциям и другим итерируемым объектам. Пример итерации по диапазону чисел от 1 до 100:

Листинг 1.7. – Пример использования цикла for

```
for (i in 1 .. 100) {  
    println ("Значение счетчика: $i")  
}
```

Цикл while выполняется, пока условие истинно, например:

Листинг 1.8. – Пример использования цикла for

```
var x = 5  
while (x > 0) {  
    println (x)  
    x -- // Уменьшаем x на единицу после каждой итерации  
}
```

Цикл do-while похож на while, но он гарантированно выполнится хотя бы один раз, поскольку условие проверяется после выполнения тела цикла. Пример:

Листинг 1.9. – Пример использования цикла do-while

```
var y = 5  
do {  
    println (y)  
    y --  
} while (y > 0)
```

3. Функции объявляются с помощью ключевого слова fun. Kotlin поддерживает функции верхнего уровня, т.е. не требует создания класса для определения функции.

Листинг 1.10. – Пример создания функций

```
// функция возвращающая значение  
fun hello (name: String): String  
// Явное указание возвращаемого типа данных  
{  
    return "Hello, $name!"  
}  
// функция, которая ничего не возвращает  
fun printMessageShort (message: String) {  
    println (message)  
}
```

4. Классы в Kotlin объявляются через ключевое слово `class` и могут содержать свойства, методы, конструкторы и многое другое.

Листинг 1.11. – Пример создания класса

```
class Book {  
    // Свойства класса  
    var title: String  
    var author: String  
    var yearPublished: Int  
  
    // Основной конструктор  
    constructor (title: String, author: String, yearPublished: Int)  
    {  
        this.title = title  
        this. author = author  
        this.yearPublished = yearPublished  
    }  
    // Метод для отображения информации о книге  
    fun displayInfo () {  
        println ("Книга: \"$title\", Автор: $author, Год издания:  
$yearPublished")  
    }  
    // Метод для обновления года издания  
    fun updateYear (newYear: Int) {  
        yearPublished = newYear  
    }  
}
```

Листинг 1.12. – Создание объекта класса Book

```
fun main () {  
    // Создаем объект класса Book  
    val myBook = Book ("Преступление и наказание", "Ф.М.  
Достоевский", 1866)  
    // Отображаем информацию о книге  
    myBook.displayInfo ()  
    // обновляем год издания с помощью метода updateYear  
    myBook.updateYear (2021)  
    myBook.displayInfo ()  
}
```

5. Объявление списков `List` и `MutableList` в Kotlin через ключевое слово `listOf` и `mutableListOf`.

Листинг 1.13. – Создание списков

```
val numbers = listOf(1, 2, 3)    // Неизменяемый
val mutableList = mutableListOf(1, 2, 3)
mutableList.add(4)

// Функциональные операции
val doubled = numbers.map { it * 2 } // [2, 4, 6]
val even = numbers.filter { it % 2 == 0 } // [2]
```

VI. Жизненный цикл Активности.

Активность – это экран, который приложение может показывать пользователям для взаимодействия с ним. Чем сложнее создаваемое приложение, тем больше экранов (Активностей) потребуется. При создании приложения потребуется, как минимум, начальный (главный) экран, который обеспечивает основу пользовательского интерфейса приложения. При необходимости этот интерфейс дополняется второстепенными Активностями, предназначенными для ввода информации, ее вывода и предоставления дополнительных возможностей. Запуск (или возврат из) новой Активности приводит к «перемещению» между экранами UI.

Большинство Активностей проектируются таким образом, чтобы использовать все экранное пространство, но можно также создавать полупрозрачные или плавающие диалоговые окна.

Для создания приложений, правильно управляющих ресурсами предоставляющих пользователю удобный интерфейс, важно хорошее понимание жизненного цикла Активности. Это связано с тем, что приложения Android не могут контролировать свой жизненный цикл, ОС сама управляет всеми процессами и, как следствие, Активностями внутри них. При этом, состояние Активности помогает ОС определить приоритет родительского для этой Активности Приложения (Application). А приоритет Приложения влияет на то, с какой вероятности его работа (и работа дочерних Активностей) будет прервана системой.

Стеки Активностей

Состояние каждой Активности определяется ее позицией в стеке (LIFO) Активностей, запущенных в данный момент. При запуске новой Активности предоставляемый ею экран помещается на вершину стека. Если пользователь нажимает кнопку «назад» или эта Активности закрывается каким-то другим образом, на вершину стека перемещается (и становится активной) нижележащая Активность. Данный процесс показан на диаграмме (рисунок 1.8).

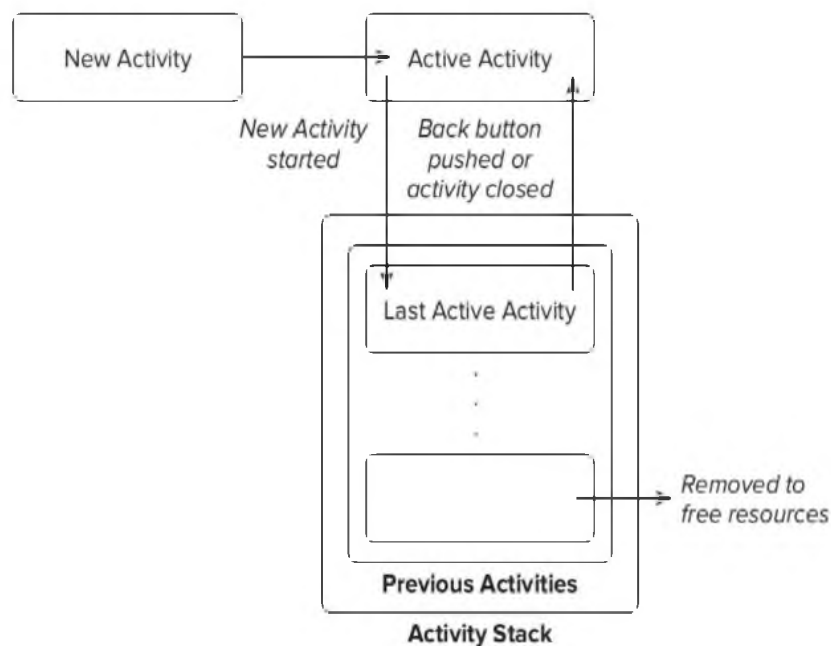


Рисунок 1.8

На приоритет приложения влияет его самая приоритетная Активность. Когда диспетчер памяти ОС решает, какую программу закрыть для освобождения ресурсов, он учитывает информацию о положении Активности в стеке для определения приоритета приложения.

Основные состояния жизненного цикла активности (рисунок 1.9):

- *Создание (Create)*: когда активность создается, вызывается метод `onCreate()`, где можно инициализировать пользовательский интерфейс, а также важные компоненты, такие как модели данных. В этом методе выполняется установка макета с помощью `setContentView()`.

- *Запуск (Start)*: после создания активности, вызывается метод `onStart()`. В этом состоянии активность становится видимой для пользователя, но не осуществляется взаимодействие с пользователем. Это место для кода, который будет выполняться, когда активность становится активной, но все еще в фоновом режиме.

- *Возобновление (Resume)*: когда пользователь начинает взаимодействовать с активностью, вызывается метод `onResume()`. Это состояние означает, что активность теперь полностью видима и активна. Здесь можно инициализировать компоненты, необходимые для взаимодействия с пользователем, например, обновить данные.

- *Приостановка (Pause)*: если пользователь переключается на другую активность или приложение уходит в фон, вызывается метод `onPause()`. Это идеальное место для сохранения данных, остановки анимаций и завершения других процессов, которые не должны продолжаться, пока активность не в фокусе.

– *Остановка (Stop)*: если активность больше не видна пользователю, вызывается метод `onStop()`. Здесь можно освобождать ресурсы, сохранять состояние и останавливать все процессы, которые не требуются, когда активность не видна.

– *Закрывие (Destroy)*: когда активность окончательно уничтожается, вызывается метод `onDestroy()`. В этом методе можно выполнять окончательную очистку, например, освобождать ресурсы, отменять классы обратного вызова и т.д.

– *Восстановление (Restart)*: если активность была остановлена и затем снова становится видимой, вызывается метод `onRestart()`. Этот метод отработывает перед методом `onStart()`, что позволяет выполнить необходимую инициализацию перед показом активности.

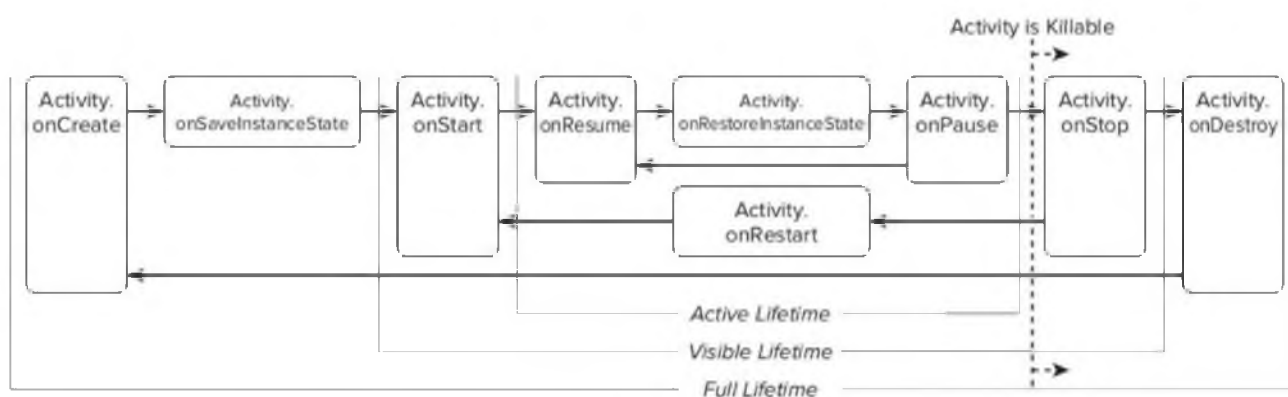


Рисунок 1.9. – Жизненный цикл Активности

Изменение состояния приложения – недетерминированный процесс и управляется исключительно менеджером памяти Android. При необходимости Android вначале закрывает приложения, содержащие неактивные Активности, затем остановленные и, в крайнем случае, приостановленные.

Для обеспечения полноценного интерфейса приложения, изменения его состояния должны быть незаметными для пользователя. Меняя свое состояние с приостановленного на остановленное или с неактивного на активное, Активность не должна внешне меняться. При остановке или приостановке работы Активности разработчик должен обеспечить сохранение состояния Активности, чтобы его можно было восстановить при выходе Активности на передний план. Для этого в классе `Activity` имеются обработчики событий, переопределение которых позволяет разработчику отслеживать изменение состояний Активности.

Отслеживание изменений состояния активности в Android – это важный аспект разработки, позволяющий реагировать на изменения в жизненном цикле

активности. Это может включать в себя изменение состояния из видимого в невидимое, приостановку и возобновление активности. Такой подход позволяет эффективно управлять ресурсами, сохранять важные данные, а также повышать общий пользовательский опыт.

Основные методы, которые используются для отслеживания изменений состояния активности, включают `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, и `onDestroy()`. Каждому из этих методов соответствует определённая фаза жизненного цикла активности, и разработчик может переопределить их, чтобы выполнить соответствующую логику.

VII. Запуск приложения.

Эмулятор – это программа для компьютера, с помощью которой можно имитировать существующие устройства, такие как, например, смартфоны и планшеты. С его помощью можно запускать и отлаживать проекты, которые потом будут установлены на реальные устройства. В эмуляторе можно задавать основные параметры устройств: размер и разрешение экрана, объем оперативной памяти, наличие разнообразных датчиков и многое другое. С помощью эмулятора можно создавать не только существующие модели девайсов, но и модели, не существующие в природе, например, такие, какие только собираются выпустить на производстве.

Android Studio поддерживает встроенный эмулятор. Он называется Device Manager. Чтобы его запустить, выберите на панели справа Manager > Значок «+» > Create Virtual Device (рисунок 1.10).

В открывшемся окне выбора устройства, необходимо выбрать: тип устройства и само устройство по характеристикам (рисунок 1.11).

После нажать кнопку Next и дождаться окончания установки (при работе с эмулятором может понадобиться включение визуализации на компьютере через BIOS).

Второй способ запуска приложения - использование физического устройства. Для этого необходимо:

1. Включить режим разработчика на устройстве.
2. Подключить устройство по USB к компьютеру.
3. Выбрать отладку по USB на устройстве как тип передачи.
4. Запустить приложение через Android Studio.

Также можно использовать Wi-Fi отладку. Для этого:

1. Подключить устройство по USB к компьютеру.
2. В Android Studio выбрать Device Manager > Pair using Wi-Fi.
3. Следовать инструкциям на экране.

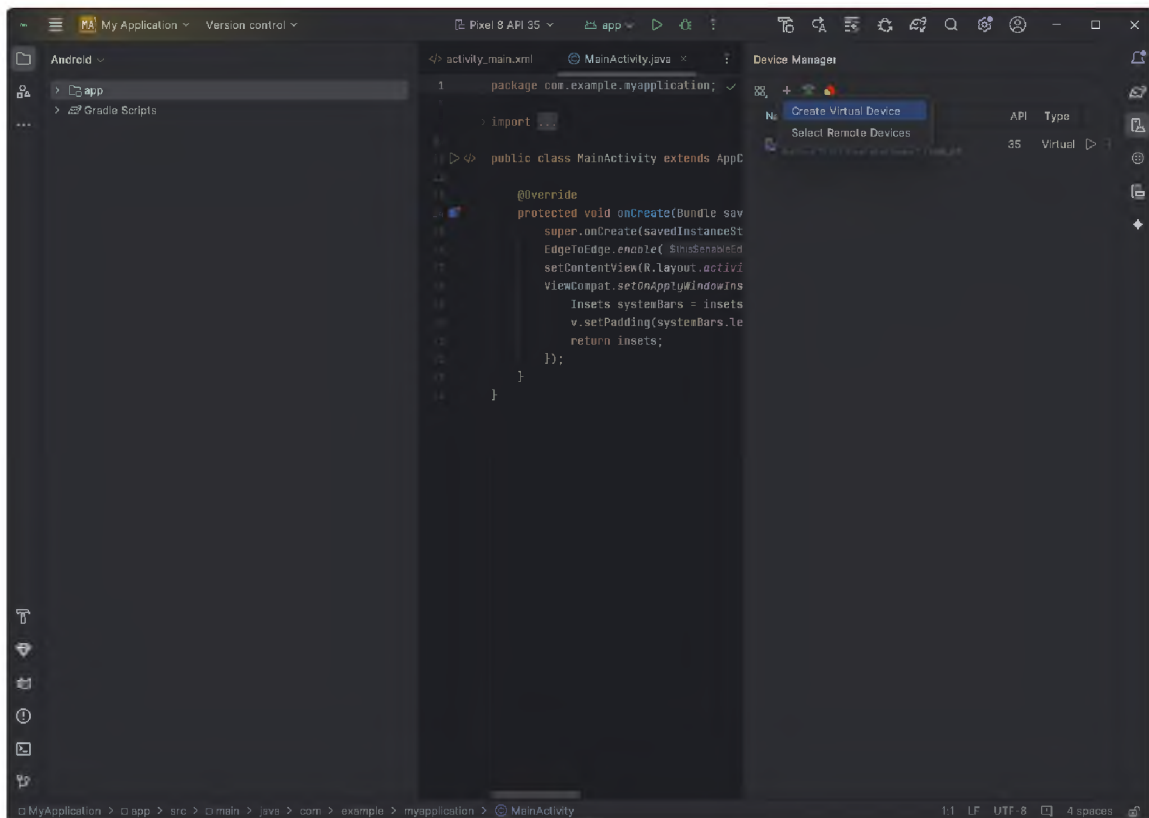


Рисунок 1.10. – Device Manager

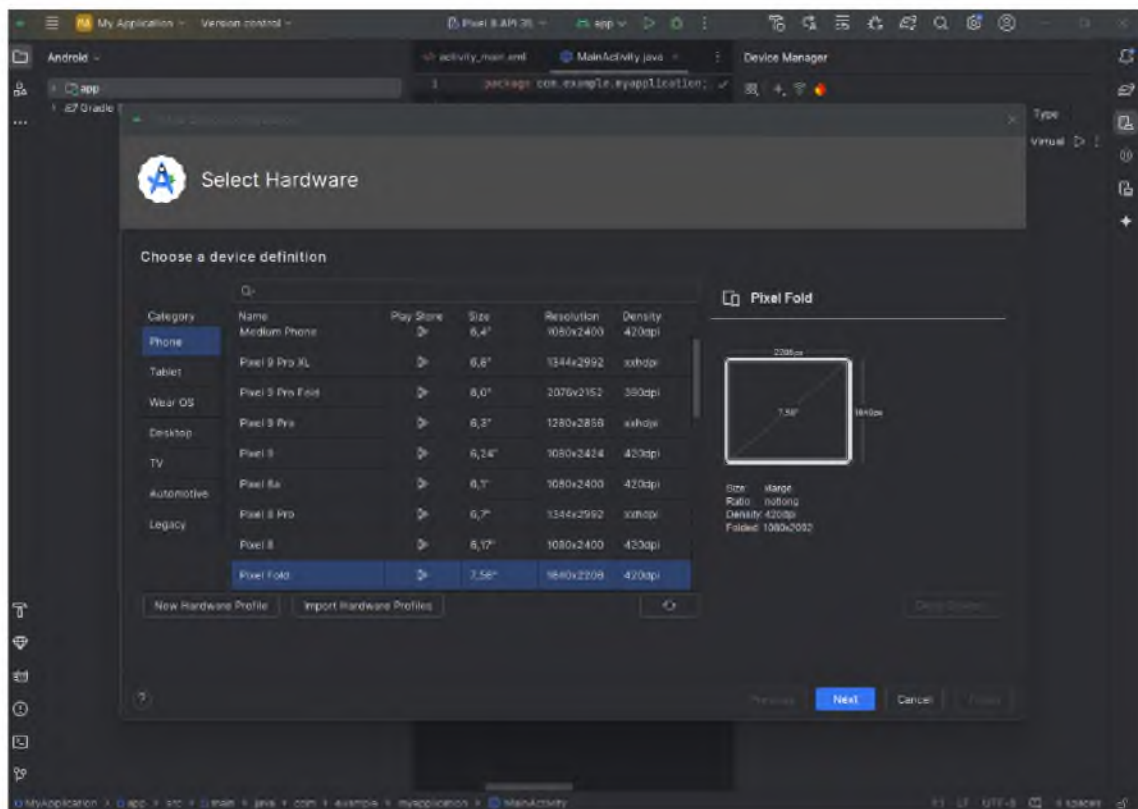


Рисунок 1.11. – Окно выбора устройства

VIII. Пример создания приложения.

В качестве примера создания приложения в Android Studio на языке Kotlin с использованием Jetpack Compose, рассмотрим разработку To-Do List.

1. Для начала создадим новый проект в Android Studio с применением шаблона Empty Activity.

2. Далее в каталог app/kotlin добавим новый файл – Kotlin class/File, и определим в нем модель данных для будущих задач в листе.

Листинг 1.14. – Модель данных

```
data class Task(  
    val id: Int,           // Уникальный ID для идентификации задачи  
    val title: String,     // Текст задачи (например: "Купить молоко")  
    var isCompleted: Boolean = false // Статус выполнения (можно изменять)  
)
```

3. Для хранения и отображения данных используем ViewModel, добавив новый файл Kotlin class/File в каталог app/kotlin.

Листинг 1.15. – ViewModel

```
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.viewModelScope  
import kotlinx.coroutines.flow.MutableStateFlow  
import kotlinx.coroutines.flow.StateFlow  
import kotlinx.coroutines.launch  
  
class TaskViewModel : ViewModel() {  
    // Приватное состояние списка задач  
    private val _tasks = MutableStateFlow<List<Task>>(emptyList())  
  
    // Публичное состояние для UI  
    val tasks: StateFlow<List<Task>> = _tasks  
  
    // Счетчик для генерации ID  
    private var nextId = 1  
  
    /**  
     * Добавление новой задачи  
     * @param title Текст задачи  
     */  
    fun addTask(title: String) {  
        viewModelScope.launch {  
            _tasks.value = _tasks.value + Task(  
                id = nextId++,  
                title = title.trim()  
            )  
        }  
    }  
}
```

```

    )
  }
}

/**
 * Удаление задачи
 * @param task Удаляемая задача
 */
fun deleteTask(task: Task) {
  viewModelScope.launch {
    _tasks.value = _tasks.value.filter { it.id != task.id }
  }
}

/**
 * Переключение статуса выполнения
 * @param task Задача для изменения
 */
fun toggleTaskCompletion(task: Task) {
  viewModelScope.launch {
    _tasks.value = _tasks.value.map {
      if (it.id == task.id) it.copy(isCompleted = !it.isCompleted) else it
    }
  }
}
}

```

4. Там же создадим новый файл для основного экрана приложения.

Листинг 1.16. – MainScreen

```

import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Add
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextDecoration
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel

@Composable
fun TodoApp() {
  val viewModel: TaskViewModel = viewModel()

```

```

val tasks by viewModel.tasks.collectAsState()
var newTaskText by remember { mutableStateOf("") }

Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp)
) {
    // Поле ввода новой задачи
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.fillMaxWidth()
    ) {
        OutlinedTextField(
            value = newTaskText,
            onChange = { newTaskText = it },
            label = { Text("Новая задача") },
            modifier = Modifier.weight(1f)
        )

        Spacer(modifier = Modifier.width(8.dp))

        IconButton(
            onClick = {
                if (newTaskText.isNotBlank()) {
                    viewModel.addTask(newTaskText)
                    newTaskText = ""
                }
            }
        ) {
            Icon(Icons.Default.Add, "Добавить")
        }
    }

    Spacer(modifier = Modifier.height(16.dp))

    // Список задач
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        items(tasks) { task ->
            TaskItem(
                task = task,
                onToggle = { viewModel.toggleTask(task) },
                onDelete = { viewModel.deleteTask(task) }
            )
        }
    }
}

```

```

    }
}

@Composable
fun TaskItem(
    task: Task,
    onToggle: () -> Unit,
    onDelete: () -> Unit
){
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp)
    ){
        Checkbox(
            checked = task.isCompleted,
            onCheckedChange = { onToggle() }
        )

        Text(
            text = task.title,
            modifier = Modifier
                .weight(1f)
                .padding(start = 8.dp),
            style = if (task.isCompleted) {
                MaterialTheme.typography.bodyLarge.copy(
                    textDecoration = TextDecoration.LineThrough,
                    color = MaterialTheme.colorScheme.onSurface.copy(alpha = 0.5f)
                )
            } else {
                MaterialTheme.typography.bodyLarge
            }
        )

        IconButton(onClick = onDelete) {
            Icon(
                imageVector = Icons.Default.Delete,
                contentDescription = "Удалить"
            )
        }
    }
}

```

Листинг 1.17. – Обновление MainActivity

```
import TodoApp
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.ui.Modifier
/**
 * Точка входа в приложение.
 * Устанавливает Compose как способ отрисовки UI и применяет тему.
 */
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        // Устанавливаем Compose контент
        setContent {
            // Применяем Material Design тему
            MaterialTheme {
                // Surface - базовый контейнер с фоном согласно теме
                Surface(
                    modifier = Modifier.fillMaxSize(), // Занимает весь экран
                    color = MaterialTheme.colorScheme.background // Цвет фона из темы
                ) {
                    TodoApp() // Отображаем главный экран
                }
            }
        }
    }
}
```

5. Настройка зависимостей в gradle.

Листинг 1.18. – app/build.gradle.kts

```
dependencies {
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.2") // ViewModel для Compose
    implementation("androidx.compose.material3:material3:1.1.2") // Material Design 3
}
```

Результат выполнения программы представлен на рисунке 1.12.



Рисунок 1.12. — To-Do List

Задание

1. Ознакомиться с теоретическим материалом.
2. Установить Android Studio.
3. Получить вариант у преподавателя.
4. Создать новый проект согласно теме варианта.
5. Перегрузить методы обработчиков состояний активности для вывода сообщения с соответствующим состоянием активности.
6. Схематично изобразить макет приложения.
7. При помощи представлений и разметок реализовать интерфейс приложения.
8. Добавить в приложение смену ориентации активности (интерфейс приложения должен отличаться в зависимости от выбранной ориентации).

Ход работы

1. Изучить теоретический материал.
2. Выполнить задания.
3. Оформить отчет.
4. Ответить на контрольные вопросы.
5. Загрузить отчет в Classroom.
6. Защитить работу у преподавателя.

Вариант	Тема
1	Путеводитель по музеям. Приложение позволяет пользователям планировать и отмечать посещение музеев в различных городах, пометать их выставки, события и особенности
2	Садоводческий помощник. Это приложение поможет пользователям планировать и ухаживать за своими садами и приусадебными участками. Оно будет содержать список имеющихся растений и напоминать о поливе
3	Кулинарная книга. Приложение, которое сочетает в себе удобный интерфейс для хранения и создания рецептов, а также выбора избранных рецептов из списка
4	Личный финансовый помощник. Приложение для управления личным бюджетом, которое поможет пользователям отслеживать доходы и расходы с возможностью отображения всего списка или только расходов/доходов
5	Тренировочный план для домашних питомцев. Приложение с программами тренировок для домашних животных позволяет хранить и создавать список тренировок, а также выбирать тренировки из списка
6	Архив личных воспоминаний. Простая и удобная платформа для записи и хранения воспоминаний пользователей
7	Каталог фильмов и сериалов. Приложение для создания списка фильмов и сериалов, а также выбора уже просмотренных
8	Организатор задач для студентов. Приложение для управления учебным процессом позволяет составлять расписание по дням
9	Дневник настроения. Приложение для отслеживания эмоций и настроения позволяет указывать, какое и в какой день было настроение у пользователя
10	Организатор мероприятий. Приложение для планирования и управления событиями с возможностью создания списка событий с датой проведения

Контрольные вопросы

1. Что такое Android Studio и на какой среде разработки она основана?
2. Какие дополнительные возможности предоставляет Android Studio помимо стандартных функций IDE?
3. Какие шаги необходимо выполнить для создания нового проекта в Android Studio?
4. Какие параметры необходимо указать при создании нового проекта в Android Studio?

5. Какие основные каталоги и файлы входят в структуру проекта Android Studio?
6. Что такое AndroidManifest.xml и какую информацию он содержит?
7. Какие типы ресурсов хранятся в каталоге res и для чего они используются?
8. Какие файлы входят в состав Gradle Scripts и какую роль они играют в проекте?
9. Какие два способа запуска приложения в Android Studio описаны в документе?
10. Какие основные состояния жизненного цикла активности в Android?
11. Какие методы используются для отслеживания изменений состояния активности?
12. Как зарегистрировать активность в файле манифеста (AndroidManifest.xml)?
13. Какие шаги необходимо выполнить для установки Android Studio и настройки SDK?

Содержание отчета

1. Титульный лист.
2. Индивидуальное задание.
3. Краткое описание стратегии решения.
4. Листинг программы.
5. Тестовые примеры, результаты тестирования.
6. Выводы.

ЛАБОРАТОРНАЯ РАБОТА 2
Создание многоэкранного Android приложения
с помощью намерений.
Работа с различными видами уведомлений

Цель работы: реализовать многоэкранные Android приложения; ознакомиться с различными видами уведомлений в Android приложениях.

Теоретическая часть

I. Navigation Component.

Навигация относится к взаимодействиям, которые позволяют пользователям перемещаться по различным частям контента приложения.

Компонент «Навигация» обрабатывает варианты использования навигации от простых нажатий кнопок до более сложных шаблонов, таких как панели приложений и панель навигации.

Каждый экран обычно представляет собой отдельный компонент или объект Activity. Каждое приложение имеет главный экран (home screen), который появляется после запуска приложения и после появления любого экрана-заставки. На этом главном экране пользователь обычно выполняет задачи, которые приводят к появлению других экранов или пунктов назначения.

Для отслеживания переходов по различным экранам Android использует стек навигации. При первом запуске приложения начальный экран помещается в стек. Когда пользователь переходит к другому экрану, то этот экран помещается на верхушку стека, к другим экранам — они также помещаются в стек. Когда пользователь перемещается обратно по экранам с помощью системной кнопки «Назад», каждый компонент извлекается из стека до тех пор, пока главный экран не окажется единственным элементом в стеке. Извлекать элементы из стека можно также программно.

II. Преимущества и особенности.

Компонент «Навигация» предоставляет ряд других преимуществ и функций, в т.ч. следующие:

- **Анимации и переходы:** предоставляет стандартизированные ресурсы для анимации и переходов.
- **Глубокие ссылки:** реализуют и обрабатывают глубокие ссылки, которые направляют пользователя непосредственно к месту назначения.

- **Шаблоны пользовательского интерфейса:** поддерживает такие шаблоны, как панели навигации и нижняя навигация, с минимальной дополнительной работой.
- **Безопасность типов:** включает поддержку передачи данных между пунктами назначения с безопасностью типов.
- **Поддержка ViewModel:** позволяет ограничить **ViewModel** навигационным графом для обмена данными, связанными с пользовательским интерфейсом, между пунктами назначения графа.
- **Транзакции фрагментов:** полностью поддерживает и обрабатывает транзакции фрагментов.
- **Назад и вверх:** по умолчанию корректно обрабатываются действия назад и вверх.

III. Состояние приложения (App's state).

Состояние приложения предназначено для навигации между экранами, показом всплывающих сообщений (Snackbar) и другими глобальными процессами.

Для работы с состоянием приложения в директории main создадим класс `OrderNowState` – центральный State Holder (листинг 2.1).

Листинг 2.1. – Центральный State Holder

```
/**
 * Композируемая функция для создания и запоминания состояния приложения.
 *
 * @param scaffoldState состояние Scaffold (по умолчанию создается новое с помощью
 rememberScaffoldState)
 * @param navController контроллер навигации (по умолчанию создается новый с помощью
 rememberNavController)
 * @param resources ресурсы приложения (по умолчанию получают через функцию
 resources())
 * @param coroutineScope область видимости для корутин (по умолчанию создается новая
 с помощью rememberCoroutineScope)
 * @return экземпляр OrderNowState, который будет сохранен и повторно использован при
 последующих вызовах
 */
@Composable
fun rememberAppState(
    scaffoldState: ScaffoldState = rememberScaffoldState(),
    navController: NavHostController = rememberNavController(),
    resources: Resources = resources(),
    coroutineScope: CoroutineScope = rememberCoroutineScope()
) = remember(scaffoldState, navController, resources, coroutineScope) {
```

```

    OrderNowState(scaffoldState, navController, resources, coroutineScope)
}

/**
 * Класс, представляющий состояние приложения "OrderNow".
 *
 * @property scaffoldState состояние Scaffold, используемое для управления состоянием
каркаса приложения
 * @property navController контроллер навигации, управляющий переходами между экра-
нами
 * @property resources ресурсы приложения, используемые для доступа к строкам, изобра-
жениям и другим ресурсам
 * @param coroutineScope область видимости для корутин, используемая для асинхронных
операций
 */
class OrderNowState(
    val scaffoldState: ScaffoldState,
    val navController: NavHostController,
    private val resources: Resources,
    coroutineScope: CoroutineScope
)

```

Далее необходимо интегрировать состояния в OrderNowScreen. Для этого добавим композируемую функцию, представляющую основной экран приложения "OrderNow" (листинг 2.2). Функция реализует:

- применение темы приложения OrderNowTheme;
- создание базовой поверхности с заполнением всего доступного пространства;
- инициализацию состояния приложения;
- отображение структуры Scaffold с верхней и нижней панелями.

Листинг 2.2. – Функция представляющая основной экран приложения

```

@Composable
fun OrderNowScreen() {
    // Применение кастомной темы приложения
    OrderNowTheme {
        // Создание базовой поверхности, занимающей все доступное пространство
        Surface(modifier = Modifier.fillMaxSize()) {
            // Инициализация состояния приложения
            val appState = rememberAppState()

            // Построение каркаса приложения (Scaffold)
            Scaffold(
                // Привязка состояния Scaffold к состоянию приложения
                scaffoldState = appState.scaffoldState,

```

```

        // Установка кастомной верхней панели приложения
        topBar = { OrderNowTopBar() },
        // Установка кастомной нижней панели приложения
        bottomBar = { OrderNowBottomBar() }
    ) { padding ->
        // Область для размещения основного контента экрана
        // Внутренние отступы (padding) обрабатываются автоматически
    }
}
}
}
}

```

Ключевые изменения:

- appState становится единым источником истины для навигации и UI;
- Scaffold теперь управляется через appState.scaffoldState.

IV. Создание навигационного контроллера.

Для управления стеком навигации и самой навигацией применяется контроллер навигации, представленным классом NavHost.

Навигационный контроллер – одно из ключевых понятий в навигации. Он содержит граф навигации и предоставляет методы, которые позволяют вашему приложению перемещаться между пунктами назначения в графе.

Навигация в OrderNow строится:

- на NavHost – контейнер для экранов;
- Граф навигации (AppSoGraph) – определяет переходы между экранами;
- Маршруты (Routes) – уникальные пути к экранам (например, "home", "cart").

Для создания маршрутов определим в common/navigation/OrderNowScreenRoute.kt класс, определяющий маршруты экранов приложения "OrderNow" (листинг 2.3). Класс реализует:

- базовую структуру для навигационных маршрутов;
- типизированные объекты маршрутов с фиксированными значениями;
- единую точку определения всех возможных переходов в приложении.

Листинг 2.3. – Class OrderNowScreenRoute

```

/**
 * @property route строковый идентификатор маршрута для навигационного контроллера
 */
sealed class OrderNowScreenRoute(val route: String) {
    /**
     * Объект, представляющий маршрут главного экрана приложения.
     */
    object Home : OrderNowScreenRoute("home")
}

```

```

/**
 * Объект, представляющий маршрут экрана корзины товаров.
 */
object Cart : OrderNowScreenRoute("cart")

/**
 * Объект, представляющий маршрут экрана списка товаров.
 */
object ProductList : OrderNowScreenRoute("product_list")

/**
 * Объект, представляющий маршрут экрана деталей товара.
 */
object ProductDetail : OrderNowScreenRoute("product_detail")

```

Далее необходимо реализовать сам NavHost для управления переходами между экранами приложения (листинг 2.4). Функция OrderNowNavHost выполняет:

- инициализацию навигационного хоста;
- установку стартового экрана приложения;
- подключение графа навигации.

А также, реализовать функцию-расширение для построения графа навигации приложения NavGraphBuilder.appSoGraph, которая осуществляет:

- определение всех возможных экранов приложения;
- привязку маршрутов к соответствующим экранам;
- настройку навигационных переходов.

Листинг 2.4. – Функции для управления переходами между экранами приложения

```

/**
 * @param appState объект состояния приложения, содержащий навигационный контрол-
 * лер
 */
@Composable
fun OrderNowNavHost(appState: OrderNowState) {
    NavHost(
        // Навигационный контроллер из состояния приложения
        navController = appState.navController,
        // Стартовый экран - главная страница приложения
        startDestination = OrderNowScreenRoute.Home.route
    ) {
        // Подключение пользовательского графа навигации
        appSoGraph(appState)
    }
}

```

```

/**
 * @param appState объект состояния приложения для передачи в экраны
 */
fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {
    // Регистрация экрана "Главная страница"
    composable(OrderNowScreenRoute.Home.route) {
        HomeScreen()
    }

    // Регистрация экрана "Корзина"
    composable(OrderNowScreenRoute.Cart.route) {
        CartScreen()
    }

    // Дополнительные экраны приложения регистрируются аналогичным образом
    // ... остальные экраны
}

```

Что важно:

- NavHost использует navController из appState.
- startDestination указывает стартовый экран (Home).

V. Переход через BottomBar.

BottomBar (нижняя панель навигации) – это стандартный компонент интерфейса в Android-приложениях, который располагается в нижней части экрана и обеспечивает быстрый доступ к основным разделам приложения.

Чтобы связать BottomBar с навигацией, необходимо создать вспомогательный класс NavigationBarSection, определяющий секции нижней навигационной панели приложения (листинг 2.5). Класс содержит:

- текстовое название секции (в виде ресурса строки);
- визуальное представление секции (иконку);
- маршрут для навигации к соответствующему экрану.

Листинг 2.5. – Класс NavigationBarSection

```

/**
 * @property title идентификатор строкового ресурса для названия секции
 * @property icon векторное изображение, представляющее секцию
 * @property route строковый идентификатор маршрута навигации
 */
sealed class NavigationBarSection(
    @StringRes val title: Int,
    val icon: ImageVector,
    val route: String
)

```

```

) {
    /**
     * Объект, представляющий секцию "Главная" в навигационной панели.
     * Соответствует экрану с основным содержимым приложения.
     */
    object Home : NavigationBarSection(
        title = R.string.home,
        icon = Icons.Default.Home,
        route = OrderNowScreenRoute.Home.route
    )

    /**
     * Объект, представляющий секцию "Корзина" в навигационной панели.
     * Соответствует экрану с товарами, добавленными в корзину.
     */
    object Cart : NavigationBarSection(
        title = R.string.cart,
        icon = Icons.Default.ShoppingCart,
        route = OrderNowScreenRoute.Cart.route
    )
}

```

Обновим функцию `OrderNowBottomBar`, реализующую нижнюю навигационную панель приложения (листинг 2.6).

Листинг 2.6. – Функция `OrderNowBottomBar`

```

/**
 * @param navController контроллер навигации для управления переходами между экранами
 */
@Composable
fun OrderNowBottomBar(navController: NavHostController) {
    // Получение текущего маршрута из стека навигации
    val currentRoute = navController.currentBackStackEntry?.destination?.route

    // Создание контейнера нижней навигационной панели
    BottomNavigation {
        // Итерация по всем разделам навигации
        NavigationBarSection.sections.forEach { section ->
            // Создание элемента навигации
            BottomNavigationBarItem(
                // Иконка раздела
                icon = {
                    Icon(
                        imageVector = section.icon,
                        contentDescription = null // Декоративный элемент, не требует описания
                    )
                }
            )
        }
    }
}

```

```

    )
    },
    // Текстовое название раздела
    label = { Text(stringResource(section.title)) },
    // Состояние выбора (активен/неактивен)
    selected = currentRoute == section.route,
    // Обработчик нажатия
    onClick = {
        navController.navigate(section.route) {
            // Очистка стека навигации до стартового экрана
            popUpTo(navController.graph.startDestinationId)
            // Предотвращение создания дубликатов экрана
            launchSingleTop = true
        }
    }
}
)
}
}
}
}

```

Итог:

- BottomBar переключается между Home и Cart.
- launchSingleTop предотвращает дублирование экземпляров.

VI. Навигация из других компонентов приложения.

Для реализации навигации из других компонентов приложения (кнопок, ссылок и т.д.) необходимо расширить OrderNowState (листинг 2.7).

Листинг 2.7. – Расширения для управления навигацией в приложении OrderNow

```

/**
 * Расширение для выполнения стандартного перехода на указанный маршрут
 *
 * @param route целевой маршрут для навигации
 *
 * Особенности реализации:
 * - Использует стратегию launchSingleTop для предотвращения дублирования экранов
 * - Не изменяет стек навигации (не выполняет pop операций)
 */
fun OrderNowState.navigate(route: String) {
    navController.navigate(route) {
        launchSingleTop = true
    }
}

```

```

* Расширение для перехода на указанный маршрут с очисткой стека навигации
*
* @param route целевой маршрут для навигации
* @param popUpRoute маршрут, до которого будет очищен стек (включительно)
*
* Особенности реализации:
* - Выполняет полную очистку стека до указанного маршрута (inclusive = true)
* - Использует стратегию launchSingleTop
* - Типичное применение: переход на главный экран с очисткой всей истории навигации
*/
fun OrderNowState.navigateAndPopUp(route: String, popUpRoute: String) {
    navController.navigate(route) {
        popUpTo(popUpRoute) {
            inclusive = true
        }
        launchSingleTop = true
    }
}

```

Пример использования в HomeScreen – композируемая функция, реализующая экран «Главная страница»:

```

/**
 * @param goToProductList лямбда-функция для обработки перехода к списку товаров
 *
 * Функционал:
 * - Отображает кнопку перехода к списку товаров
 * - Делегирует обработку навигации вызывающей стороне
 */
@Composable
fun HomeScreen(goToProductList: () -> Unit) {
    Button(
        onClick = goToProductList,
        modifier = Modifier.padding(16.dp)
    ) {
        Text(
            text = stringResource(R.string.go_to_products),
            style = MaterialTheme.typography.button
        )
    }
}

/**
 * Регистрация экрана "Главная страница" в графе навигации.
 *
 * Особенности реализации:
 * - Навигация осуществляется через централизованное состояние приложения (appState)
 */

```

```

* - Используется ранее объявленная функция navigate для обеспечения согласованного по-
ведения
*/
composable(OrderNowScreenRoute.Home.route) {
    HomeScreen(
        goToProductList = {
            appState.navigate(OrderNowScreenRoute.ProductList.route)
        }
    )
}

```

VII. Работа с уведомлениями.

Уведомления играют важную роль в разработке Android-приложений. *Уведомлением* называется сообщение, которое отображается операционной системой вне интерфейса приложения для предоставления пользователю актуальной информации из него. К таковой информации могут относиться сообщения из социальных сетей, письма электронной почты, прогресс проигрывания музыкальной композиции и т.д.

Уведомления автоматически отображаются пользователям в разных местах и форматах. Уведомление отображается в виде значка в строке состояния, более подробной записи в панели уведомлений и значка на значке приложения. Уведомления также появляются на сопряженных носимых устройствах.

VIII. Строка состояния и ящик уведомлений.

Когда вы отправляете уведомление, оно сначала отображается в виде значка в строке состояния (рисунок 2.1).

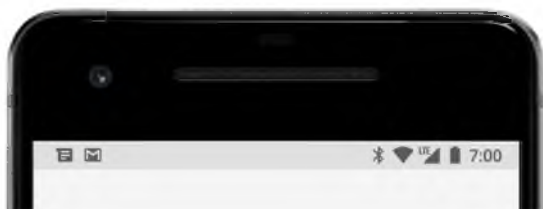


Рисунок 2.1. — Значки уведомлений появляются в левой части строки состояния

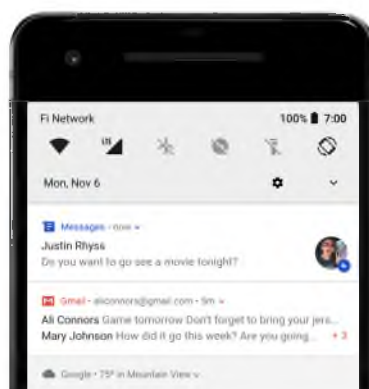


Рисунок 2.2. — Уведомления в панели уведомлений

Пользователи могут перетащить уведомление в ящике вниз, чтобы открыть расширенное представление, в котором отображается дополнительный контент

и кнопки действий, если они предусмотрены. Начиная с Android 13, это расширенное представление включает кнопку, позволяющую пользователям остановить приложение, в котором есть текущие службы на переднем плане.

Уведомление остается видимым в панели уведомлений до тех пор, пока оно не будет закрыто приложением или пользователем (см. рисунок 2.2).

IX. Предварительное уведомление.

Начиная с Android 5.0, уведомления могут ненадолго появляться в плавающем окне (рисунок 2.3), называемом хедз-ап-уведомлением. Обычно такое поведение предназначено для важных уведомлений, о которых пользователь должен знать немедленно, и оно появляется только в том случае, если устройство разблокировано.

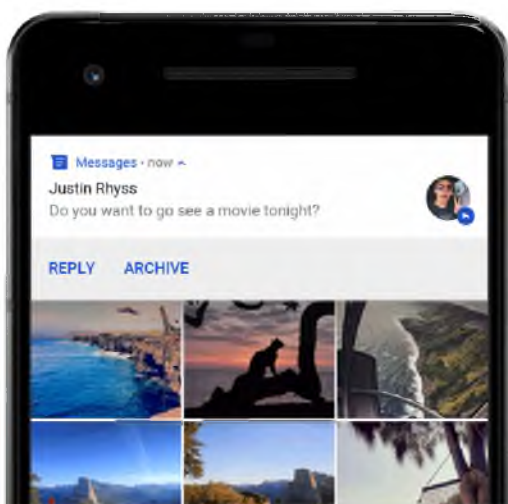


Рисунок 2.3. – Перед приложением на переднем плане появляется всплывающее уведомление

Информационное уведомление появляется, когда ваше приложение отправляет уведомление. Через некоторое время он исчезает, но остается видимым в панели уведомлений, как обычно.

Условия, которые могут вызвать хедз-ап уведомления:

- Действия пользователя происходят в полноэкранном режиме, например, когда приложение использует `fullScreenIntent`.
- Уведомление имеет высокий приоритет и использует мелодии звонка или вибрацию на устройствах под управлением Android 7.1 (уровень API 25) и ниже.
- Канал уведомлений имеет большое значение на устройствах под управлением Android 8.0 (уровень API 26) и выше.

X. Экран блокировки.

Начиная с Android 5.0, уведомления могут появляться на экране блокировки.

Вы можете программно указать, будут ли уведомления, публикуемые вашим приложением, отображаться на безопасном экране блокировки; если да, то какой уровень детализации будет отображаться.

Пользователи могут использовать настройки системы, чтобы выбрать уровень детализации, отображаемый в уведомлениях на экране блокировки, или отключить все уведомления на экране блокировки. Начиная с Android 8.0, пользователи могут отключать или включать уведомления на экране блокировки для каждого канала уведомлений.



Рисунок 2.4. – Уведомления на экране блокировки со скрытым конфиденциальным содержанием

XI. Значок приложения.

В поддерживаемых средствах запуска на устройствах под управлением Android 8.0 (уровень API 26) и выше значки приложений обозначают новые уведомления цветным значком, известным как точка уведомления, на соответствующем значке средства запуска приложений.

Пользователи могут коснуться и удерживать значок приложения, чтобы просмотреть уведомления этого приложения (рисунок 2.5). Пользователи могут отклонять уведомления или действовать в соответствии с ними из этого меню, аналогично панели уведомлений.

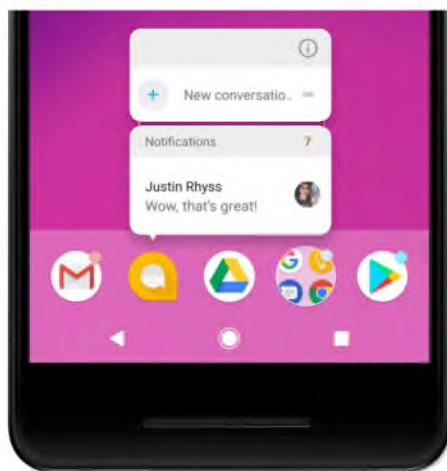


Рисунок 2.5. – Значки уведомлений и меню касания и удержания

XII. Структура уведомлений.

Дизайн уведомления определяется системными шаблонами, а ваше приложение определяет содержимое каждой части шаблона. Некоторые сведения об уведомлении отображаются только в развернутом виде.

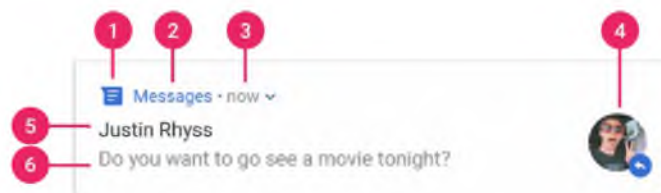


Рисунок 2.6. – Уведомление с основными сведениями

Наиболее распространенные части уведомления (см. рисунок 2.6):

1. Маленький значок: обязателен; установить с помощью `setSmallIcon()`.
2. Имя приложения: предоставляется системой.
3. Метка времени: предоставляется системой, но вы можете переопределить ее с помощью `setWhen()` или скрыть с помощью `setShowWhen(false)`.
4. Большой значок: необязательно; обычно используется только для фотографий контактов. Не используйте его для значка вашего приложения. Установите с помощью `setLargeIcon()`.
5. Название: необязательно; установить с помощью `setContentTitle()`.
6. Текст: необязательно; установить с помощью `setContentText()`.

XIII. Действия по оповещению.

Хотя это и не обязательно, рекомендуется для каждого уведомления открывать соответствующее действие приложения при его нажатии. В дополнение

к этому действию уведомления по умолчанию вы можете добавить кнопки действий, которые выполняют задачу, связанную с приложением, из уведомления — часто без открытия действия (рисунок 2.7).

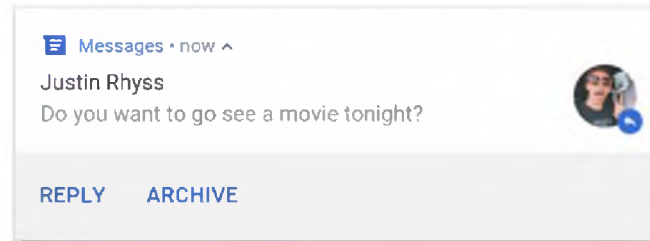


Рисунок 2.7. – Уведомление с кнопками действий

Начиная с Android 7.0 (уровень API 24), вы можете добавить действие для ответа на сообщения или ввести другой текст непосредственно из уведомления.

Начиная с Android 10 (уровень API 29), платформа может автоматически создавать кнопки действий с предлагаемыми действиями на основе намерений.

XIV. Создание уведомлений.

Уведомление в своей самой простой и компактной форме, также известной как свернутая форма, отображает значок, заголовок и небольшое количество текстового контента (рисунок 2.8). В этом разделе показано, как создать уведомление, которое пользователь может нажать, чтобы запустить действие в вашем приложении.

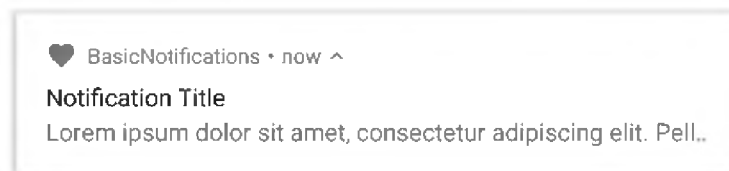


Рисунок 2.8. – Уведомление со значком, заголовком и текстом

Android 13 (уровень API 33) и более поздних версий поддерживает разрешение среды выполнения для публикации не исключенных уведомлений (в т.ч. службы переднего плана (FGS)) из приложения.

Разрешение, которое вам необходимо объявить в файле манифеста вашего приложения, отображается в следующем фрагменте кода:

```
<manifest ...>
  <uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
  <application ...>
    ...
  </application>
</manifest>
```

Для начала задайте содержимое и канал уведомления с помощью объекта `NotificationCompat.Builder`. В примере ниже показано, как создать уведомление со следующим:

- небольшой значок, установленный с помощью `setSmallIcon()`. Это единственный необходимый контент, видимый пользователю;
- заголовок, заданный с помощью `setContentTitle()`;
- основной текст, заданный с помощью `setContentText()`;
- приоритет уведомления, установленный `setPriority()`. Приоритет определяет, насколько навязчивым будет уведомление на Android 7.1 и более ранних версиях. Для Android 8.0 и более поздних версий вместо этого установите важность канала, как показано в следующем фрагменте исходного кода.

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle(textTitle)
    .setContentText(textContent)
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

Конструктор `NotificationCompat.Builder` требует, чтобы вы указали идентификатор канала. Это необходимо для совместимости с Android 8.0 (уровень API 26) и более поздних версий, но игнорируется более ранними версиями.

По умолчанию текстовое содержимое уведомления обрезается до одной строки. Вы можете показать дополнительную информацию, создав расширяемое уведомление.

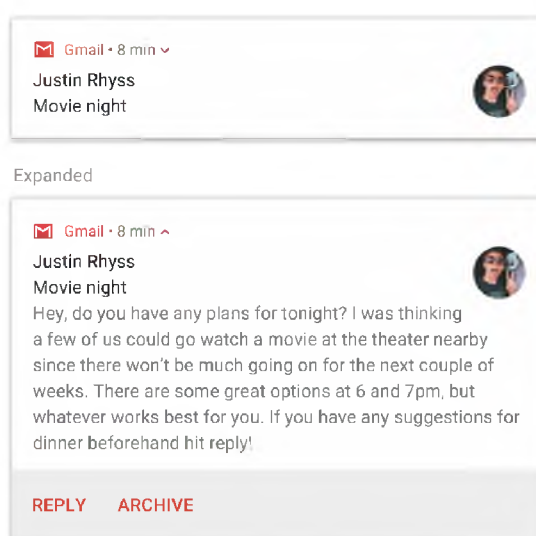


Рисунок 2.9. – Расширяемое уведомление в свернутом и развернутом виде

Если вы хотите, чтобы ваше уведомление было длиннее, вы можете включить расширяемое уведомление, добавив шаблон стиля с помощью `setStyle()`. Например, следующий код создает большую текстовую область:

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.notification_icon)
    .setTitle("My notification")
    .setText("Much longer text that cannot fit one line...")
    .setStyle(NotificationCompat.BigTextStyle()
        .bigText("Much longer text that cannot fit one line..."))
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

Прежде чем вы сможете доставить уведомление на Android 8.0 и более поздних версиях, зарегистрируйте канал уведомлений вашего приложения в системе, передав экземпляр `NotificationChannel` в `createNotificationChannel()`. Следующий код заблокирован условием версии `SDK_INT`:

```
private fun createNotificationChannel() {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is not in the Support Library.
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
        val descriptionText = getString(R.string.channel_description)
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
            description = descriptionText
        }
        // Register the channel with the system.
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

Каждое уведомление должно реагировать на нажатие, обычно для открытия действия в вашем приложении, соответствующего уведомлению. Для этого укажите намерение содержимого, определенное с помощью объекта `PendingIntent`, и передайте его в `setContentIntent()`.

В следующем фрагменте показано, как создать базовое намерение открыть действие, когда пользователь касается уведомления:

```
// Create an explicit intent for an Activity in your app.
val intent = Intent(this, AlertDetails::class.java).apply {
    flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
}
```

```

val pendingIntent: PendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG_IMMUTABLE)

val builder = NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("My notification")
    .setContentText("Hello World!")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    // Set the intent that fires when the user taps the notification.
    .setContentIntent(pendingIntent)
    .setAutoCancel(true)

```

Этот код вызывает `setAutoCancel()`, который автоматически удаляет уведомление, когда пользователь нажимает на него.

Флаги намерения в предыдущем примере сохраняют ожидаемые возможности навигации пользователя после того, как пользователь откроет ваше приложение с помощью уведомления. Возможно, вы захотите использовать его в зависимости от типа деятельности, которую вы начинаете, что может быть одним из следующих:

- действие, которое существует исключительно для ответов на уведомления. Нет причин, по которым пользователь переходит к этому действию во время обычного использования приложения, поэтому действие запускает новую задачу вместо того, чтобы добавляться к существующей задаче вашего приложения и обратному стеку. Это тип намерения, созданный в предыдущем примере;

- действие, которое существует в обычном потоке приложения вашего приложения. В этом случае при запуске действия создается обратный стек, поэтому ожидания пользователя в отношении кнопок «Назад» и «Вверх» сохраняются.

Чтобы уведомление появилось, вызовите `NotificationManagerCompat.notify()`, передав ему уникальный идентификатор уведомления и результат `NotificationCompat.Builder.build()`:

```

with(NotificationManagerCompat.from(this)) {
    if (ActivityCompat.checkSelfPermission(
        this@MainActivity,
        Manifest.permission.POST_NOTIFICATIONS
    ) != PackageManager.PERMISSION_GRANTED
    ) {
        // TODO: Consider calling
        // ActivityCompat#requestPermissions
        // here to request the missing permissions, and then overriding

```

```

// public fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>,
//                                     grantResults: IntArray)
// to handle the case where the user grants the permission. See the documentation
// for ActivityCompat#requestPermissions for more details.

return@with
}
// notificationId is a unique int for each notification that you must define.
notify(NOTIFICATION_ID, builder.build())
}

```

Сохраните идентификатор уведомления, который вы передаете в `NotificationManagerCompat.notify()`, поскольку он понадобится вам, когда вы захотите обновить или удалить уведомление.

Кроме того, чтобы протестировать базовые уведомления на устройствах под управлением Android 13 и выше, включите уведомления вручную или создайте диалог для запроса уведомлений.

XV. Добавление кнопки в уведомление.

Уведомление может содержать до трех кнопок действий (рисунок 2.10), которые позволяют пользователю быстро реагировать, например, отложить напоминание или ответить на текстовое сообщение. Но эти кнопки действий не должны дублировать действие, выполняемое при нажатии пользователем уведомления.

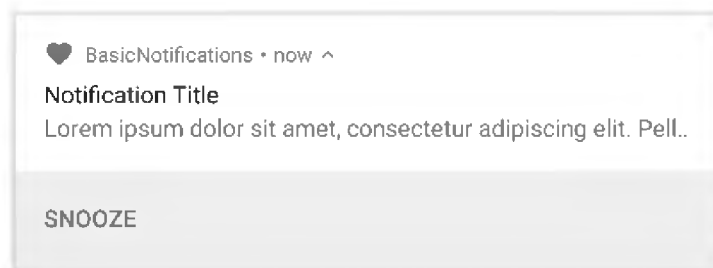


Рисунок 2.10. – Уведомление с одной кнопкой действия

Чтобы добавить кнопку действия, передайте `PendingIntent` в метод `addAction()`. Это похоже на настройку действия касания по умолчанию для уведомления, за исключением того, что вместо запуска действия вы можете выполнять другие действия, например, запускать `BroadcastReceiver`, который выполняет задание в фоновом режиме, чтобы действие не прерывало работу уже открытого приложения.

Например, следующий код показывает, как отправить широковебательную рассылку определенному получателю:

```

val ACTION_SNOOZE = "snooze"

val snoozeIntent = Intent(this, MyBroadcastReceiver::class.java).apply {
    action = ACTION_SNOOZE
    putExtra(EXTRA_NOTIFICATION_ID, 0)
}
val snoozePendingIntent: PendingIntent =
    PendingIntent.getBroadcast(this, 0, snoozeIntent, 0)
val builder = NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("My notification")
    .setContentText("Hello World!")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .setContentIntent(pendingIntent)
    .addAction(R.drawable.ic_snooze, getString(R.string.snooze),
        snoozePendingIntent)

```

XVI. AlertDialog.

Составной компонент AlertDialog предоставляет удобный API для создания тематического диалогового окна Material Design. AlertDialog имеет специальные параметры для обработки определенных элементов диалога:

- title: текст, который появляется в верхней части диалогового окна;
- text: текст, который отображается по центру диалогового окна;
- icon: рисунок, который появляется в верхней части диалогового окна;
- onDismissRequest: функция, вызываемая, когда пользователь закрывает диалоговое окно, например, нажав за его пределами;
- dismissButton: составной элемент, который служит кнопкой закрытия;
- confirmButton: составной элемент, который служит кнопкой подтверждения.

В следующем примере в диалоговом окне предупреждения реализованы две кнопки: одна закрывает диалоговое окно, а другая подтверждает его запрос.

Листинг 2.8. – Функция, реализующая диалоговое окно с подтверждением действия

```

/**
 * @param onDismissRequest обработчик закрытия диалога (при нажатии вне области или
 *     кнопки отмены)
 * @param onConfirmation обработчик подтверждения действия
 * @param dialogTitle заголовок диалогового окна
 * @param dialogText основной текст сообщения
 * @param icon иконка для визуального выделения типа диалога
 *

```

```

* Особенности реализации:
* - Соответствует рекомендациям Material Design для диалоговых окон
* - Поддерживает два варианта действий: подтверждение и отмена
* - Обеспечивает доступность через contentDescription для иконки
*/
@Composable
fun AlertDialogExample(
    onDismissRequest: () -> Unit,
    onConfirmation: () -> Unit,
    dialogTitle: String,
    dialogText: String,
    icon: ImageVector,
) {
    AlertDialog(
        // Визуальная иконка типа диалога
        icon = {
            Icon(
                imageVector = icon,
                contentDescription = "Example Icon" // Описание для accessibility сервисов
            )
        },
        // Заголовок диалога
        title = {
            Text(
                text = dialogTitle,
                style = MaterialTheme.typography.h6
            )
        },
        // Основной текст сообщения
        text = {
            Text(
                text = dialogText,
                style = MaterialTheme.typography.body1
            )
        },
        // Обработчик закрытия диалога
        onDismissRequest = {
            onDismissRequest()
        },
        // Кнопка подтверждения действия
        confirmButton = {
            TextButton(
                onClick = {
                    onConfirmation()
                }
            ) {
                Text(
                    text = stringResource(R.string.confirm), // Использование строкового ресурса

```

```

        style = MaterialTheme.typography.button
    )
}
},
// Кнопка отмены действия
dismissButton = {
    TextButton(
        onClick = {
            onDismissRequest()
        }
    ){
        Text(
            text = stringResource(R.string.dismiss), // Использование строкового ресурса
            style = MaterialTheme.typography.button
        )
    }
}
}
)
}

```

Эта реализация подразумевает родительский составной объект, который передает аргументы дочернему составному объекту следующим образом:

```

@Composable
fun DialogExamples() {
    // Инициализация состояния для управления видимостью диалога
    val openAlertDialog = remember { mutableStateOf(false) }

    // ... (другие элементы UI)

    // Условный блок для отображения диалоговых окон
    when {
        // ... (другие условия)

        // Условие отображения AlertDialog
        openAlertDialog.value -> {
            AlertDialogExample(
                // Обработчик закрытия диалога (при нажатии вне области или кнопки отмены)
                onDismissRequest = {
                    openAlertDialog.value = false
                },
                // Обработчик подтверждения действия
                onConfirmation = {
                    openAlertDialog.value = false
                    // Регистрация факта подтверждения (заменить на бизнес-логику)
                    println("Confirmation registered")
                    // Рекомендуется добавить обработку подтверждения:
                    // viewModel.handleConfirmation()
                }
            )
        }
    }
}

```

```

        // или
        // onConfirmAction.invoke()
    },
    // Параметры контента диалога
    dialogTitle = "Alert dialog example",
    dialogText = "This is an example of an alert dialog with buttons.",
    icon = Icons.Default.Info
)
}
}
}

```

Функция включает:

- управление состоянием видимости диалогового окна;
- логику обработки действий подтверждения/отмены;
- пример реализации AlertDialog с различными параметрами (рисунок 2.11).

Эта реализация выглядит следующим образом:

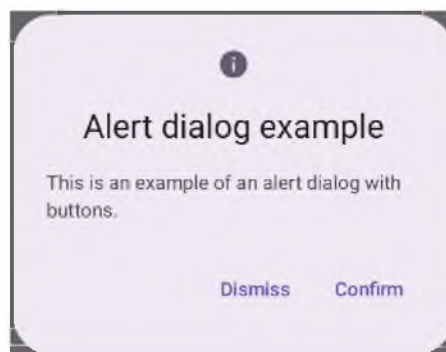


Рисунок 2.11. – AlertDialog

XVII. Snackbar в Material 3.

Snackbar – компонент, служит в качестве краткого уведомления, которое появляется в нижней части экрана (рисунок 2.12). Он обеспечивает обратную связь об операции или действии, не прерывая работу пользователя. Snackbars исчезают через несколько секунд. Пользователь также может закрыть их с помощью действия, например, нажав кнопку.

Три варианта использования Snackbar:

- Подтверждение действия: после того, как пользователь удаляет электронное письмо или сообщение, появляется Snackbar для подтверждения действия и предложения опции «Отменить».
- Статус сети: когда приложение теряет подключение к Интернету, появляется Snackbar, сообщающий, что оно теперь не в сети.
- Отправка данных: после успешной отправки формы или обновления настроек Snackbar отмечает, что изменение успешно сохранено.

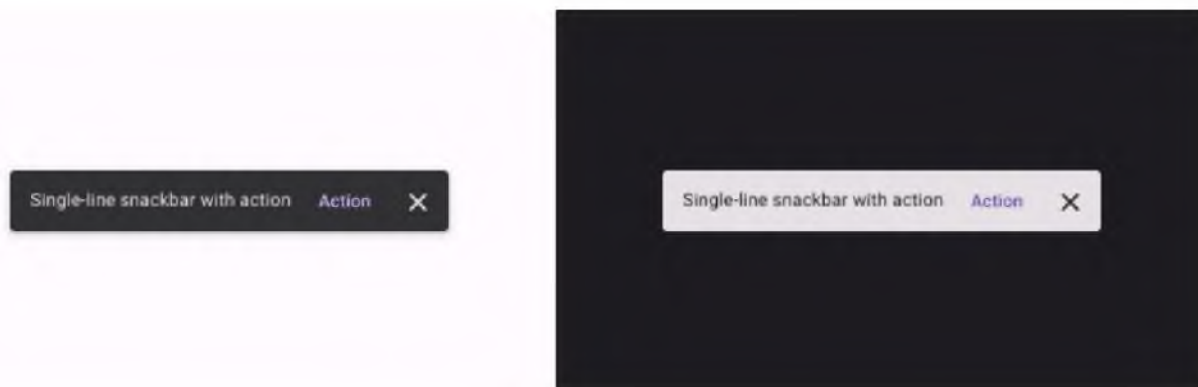


Рисунок 2.12. – Пример Snackbar

Чтобы реализовать Snackbar, сначала необходимо создать SnackbarHost, который включает свойство SnackbarHostState. SnackbarHostState предоставляет доступ к функции showSnackbar(), которую можно использовать для отображения Snackbar.

Для этой функции приостановки требуется CoroutineScope, например, при использовании rememberCoroutineScope – и ее можно вызывать в ответ на события пользовательского интерфейса, чтобы отобразить Snackbar в Scaffold.

Листинг 2.9. – Реализация Scaffold с Snackbar и FloatingActionButton

```
// Создание CoroutineScope для управления асинхронными операциями
val scope = rememberCoroutineScope()

// Инициализация состояния SnackbarHost
val snackbarHostState = remember { SnackbarHostState() }

// Основной каркас приложения
Scaffold(
    // Конфигурация SnackbarHost
    snackbarHost = {
        SnackbarHost(
            hostState = snackbarHostState,
            modifier = Modifier.testTag("snackbar_host")
        )
    },

    // Настройка расширенной плавающей кнопки действий
    floatingActionButton = {
        ExtendedFloatingActionButton(
            // Текстовое содержимое кнопки
            text = {
                Text(
                    text = stringResource(R.string.show_snackbar),
```

```

        style = MaterialTheme.typography.button
    )
},
// Иконка кнопки
icon = {
    Icon(
        imageVector = Icons.Filled.Image,
        contentDescription = stringResource(R.string.show_snackbar_cd)
    )
},
// Обработчик нажатия
onClick = {
    scope.launch {
        // Отображение Snackbar с анимацией
        snackbarHostState.showSnackbar(
            message = stringResource(R.string.snackbar_message),
            duration = SnackbarDuration.Short
        )
    }
},
// Модификатор для тестирования
modifier = Modifier.testTag("show_snackbar_fab")
)
}
) { contentPadding ->
    // Основное содержимое экрана
    Box(
        modifier = Modifier
            .padding(contentPadding)
            .fillMaxSize()
    ) {
        /* Контент страницы */
    }
}
}

```

XVIII. Пример создание многоэкранного приложения с уведомлением.

Ниже представлен код приложения с двум активностями, переход между которыми осуществляется нажатием кнопки (рисунки 2.12, 2.13). При переходе между активностями пользователь получает уведомление с данными об активности.

Создание MainActivity, главного экрана приложения:

Листинг 2.10. – MainActivity

```

// Объявление пакета приложения
package com.example.myapplication

```

```

// Импорт кастомной темы приложения
import com.example.myapplication.ui.theme.MyAppTheme

// Импорт компонентов Android
import android.Manifest
import android.app.NotificationChannel
import android.app.NotificationManager
import android.content.Context
import android.content.Intent
import android.content.pm.PackageManager
import android.os.Build
import android.os.Bundle
import android.util.Log

// Импорт компонентов Jetpack Compose
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.result.contract.ActivityResultContracts
import androidx.annotation.RequiresPermission
import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.unit.dp

// Импорт утилит совместимости
import androidx.core.app.ActivityCompat
import androidx.core.app.NotificationCompat
import androidx.core.app.NotificationManagerCompat
import androidx.core.content.ContextCompat

/**
 * Основная Activity приложения, наследующая ComponentActivity
 */
class MainActivity : ComponentActivity() {
    /**
     * Регистрация обработчика запроса разрешений.
     * Лямбда-выражение выполняется после получения ответа от пользователя.
     */
    private val requestPermissionLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { isGranted ->
        if (isGranted) {
            Log.i("Permission", "POST_NOTIFICATIONS granted")
        } else {
            Log.w("Permission", "POST_NOTIFICATIONS denied")
        }
    }
}

```

```

}

/**
 * Точка входа в Activity.
 * @param savedInstanceState Сохраненное состояние активности
 */
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Инициализация канала уведомлений
    createNotificationChannel()

    // Установка Compose-контента
    setContent {
        // Применение кастомной темы приложения
        MyAppTheme {
            // Отображение главного экрана с передачей обработчика запроса разрешений
            MainScreen(
                onRequestPermission = {
                    requestPermissionLauncher.launch(Manifest.permission.POST_NOTIFICATIONS)
                }
            )
        }
    }
}

/**
 * Создание канала уведомлений (требуется для Android 8.0+)
 */
private fun createNotificationChannel() {
    // Проверка версии Android
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        // Конфигурация канала уведомлений
        val channel = NotificationChannel(
            "transition_channel", // Идентификатор канала
            "Переходы между экранами", // Видимое имя канала
            NotificationManager.IMPORTANCE_DEFAULT // Уровень важности
        ).apply {
            description = "Уведомления при переходах между активностями" // Описание
        }

        // Регистрация канала в системе
        val notificationManager = getSystemService(NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}

/**

```

```

* Функция отображения уведомления о переходе.
* Требуется наличия разрешения POST_NOTIFICATIONS.
* @param context Контекст приложения
* @param message Текст уведомления
*/
@RequiresPermission(Manifest.permission.POST_NOTIFICATIONS)
fun showTransitionNotification(context: Context, message: String) {
    // Построение уведомления
    val notification = NotificationCompat.Builder(context, "transition_channel")
        .setSmallIcon(android.R.drawable.ic_dialog_info) // Иконка уведомления
        .setContentTitle("Произошел переход") // Заголовок
        .setContentText(message) // Текст сообщения
        .setPriority(NotificationCompat.PRIORITY_DEFAULT) // Приоритет
        .build()

    // Отображение уведомления
    NotificationManagerCompat.from(context).notify(1, notification)
}

/**
* Композиция главного экрана приложения.
* @param onRequestPermission Обработчик запроса разрешений
*/
@Composable
fun MainScreen(onRequestPermission: () -> Unit) {
    // Получение контекста композиции
    val context = LocalContext.current

    // Вертикальная компоновка элементов интерфейса
    Column(
        modifier = Modifier
            .fillMaxSize() // Заполнение всего доступного пространства
            .padding(16.dp), // Внутренние отступы
        verticalArrangement = Arrangement.Center, // Вертикальное выравнивание по центру
        horizontalAlignment = Alignment.CenterHorizontally // Горизонтальное выравнивание по центру
    ) {
        // Текстовый элемент заголовка
        Text(
            text = "Главный экран",
            style = MaterialTheme.typography.headlineMedium // Стиль из темы
        )

        // Разделительный элемент
        Spacer(modifier = Modifier.height(32.dp))

        // Кнопка перехода на второй экран
        Button(
            onClick = {

```

```

// Обработка различных сценариев для Android 13+
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
    when {
        // Проверка наличия разрешения
        ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.POST_NOTIFICATIONS
        ) == PackageManager.PERMISSION_GRANTED -> {
            // Действия при наличии разрешения
            showTransitionNotification(context, "Переход на SecondActivity")
            context.startActivity(Intent(context, SecondActivity::class.java))
        }
        // Проверка необходимости показа объяснения
        ActivityCompat.shouldShowRequestPermissionRationale(
            context as ComponentActivity,
            Manifest.permission.POST_NOTIFICATIONS
        ) -> {
            // Запрос разрешения с предварительным объяснением
            onRequestPermission()
        }
        else -> {
            // Прямой запрос разрешения
            onRequestPermission()
        }
    }
} else {
    // Стандартное поведение для версий ниже Android 13
    showTransitionNotification(context, "Переход на SecondActivity")
    context.startActivity(Intent(context, SecondActivity::class.java))
}
}
){
    // Текст кнопки
    Text("Перейти на второй экран")
}
}

```

Создание SecondActivity, второго экрана приложения:

Листинг 2.11. – SecondActivity

```

/**
 * Класс второй активности приложения, наследующий функциональность
 * ComponentActivity.
 * Обеспечивает отображение пользовательского интерфейса и обработку событий.
 */
class SecondActivity : ComponentActivity() {
    /**

```

```

* Метод инициализации активности.
* @param savedInstanceState Сохраненное состояние активности (может быть null)
*/
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    /**
     * Установка Compose-контента в качестве корневого представления активности.
     * Обеспечивает отрисовку пользовательского интерфейса средствами Jetpack Compose.
     */
    setContent {
        /**
         * Применение кастомной темы приложения ко всем дочерним элементам.
         * Обеспечивает единообразие визуального оформления.
         */
        MyAppTheme {
            /**
             * Инициализация и отображение основного композитного экрана.
             */
            SecondScreen()
        }
    }

    /**
     * Демонстрация системного уведомления при переходе на данную активность.
     * @param context Контекст текущей активности
     * @param message Текст информационного сообщения
     */
    showTransitionNotification(this, "Переход на SecondActivity")
}

/**
 * Композитная функция, реализующая интерфейс второго экрана приложения.
 * Отвечает за визуальное представление и обработку пользовательских действий.
 */
@Composable
fun SecondScreen() {
    /**
     * Получение контекста выполнения для доступа к системным сервисам и ресурсам.
     */
    val context = LocalContext.current

    /**
     * Вертикальная компоновка элементов интерфейса.
     * @param modifier Модификаторы для настройки внешнего вида контейнера
     * @param verticalArrangement Распределение элементов по вертикали
     * @param horizontalAlignment Выравнивание элементов по горизонтали

```

```

*/
Column(
    modifier = Modifier
        .fillMaxSize() // Занимает все доступное пространство
        .padding(16.dp), // Внутренние отступы 16dp
    verticalArrangement = Arrangement.Center, // Центральное выравнивание по верти-
        кали
    horizontalAlignment = Alignment.CenterHorizontally // Центральное выравнивание по го-
        ризонтالي
) {
    /**
     * Текстовый элемент заголовка экрана.
     * @param text Отображаемый текст
     * @param style Стиль текста из текущей темы
     */
    Text(
        text = "Второй экран",
        style = MaterialTheme.typography.headlineMedium // Использование стиля
            headlineMedium
    )

    /**
     * Разделитель между элементами интерфейса.
     * @param modifier Модификатор с указанием высоты 32dp
     */
    Spacer(modifier = Modifier.height(32.dp))

    /**
     * Интерактивная кнопка для возврата на предыдущий экран.
     * @param onClick Обработчик события нажатия
     */
    Button(
        onClick = {
            /**
             * Демонстрация системного уведомления при возврате.
             */
            showTransitionNotification(context, "Возврат на MainActivity")

            /**
             * Завершение текущей активности с возвратом на предыдущий экран.
             * Приведение контекста к типу Activity для доступа к методу finish().
             */
            (context as Activity).finish()
        }
    ) {
        /**
         * Текстовое содержимое кнопки.
         */
        Text("Вернуться назад")
    }
}

```

```

    }
  }
}

/**
 * Файл темы приложения (ui/theme/Theme.kt)
 * Содержит настройки цветовых схем и тему приложения.
 */

/**
 * Цветовая схема для темного режима интерфейса.
 * @property primary Основной цвет
 * @property secondary Вторичный цвет
 * @property tertiary Третичный цвет
 */
private val DarkColorScheme = darkColorScheme(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)

/**
 * Цветовая схема для светлого режима интерфейса.
 * @property primary Основной цвет
 * @property secondary Вторичный цвет
 * @property tertiary Третичный цвет
 */
private val LightColorScheme = lightColorScheme(
    primary = Purple40,
    secondary = PurpleGrey40,
    tertiary = Pink40
)

/**
 * Кастомная тема приложения.
 * Реализует адаптацию к системным настройкам темы (светлая/темная).
 * @param darkTheme Флаг темной темы (по умолчанию определяется системой)
 * @param content Лямбда-выражение с композитным контентом
 */
@Composable
fun MyAppTheme(
    darkTheme: Boolean = isSystemInDarkTheme(), // Автоматическое определение систем-
        ной темы
    content: @Composable () -> Unit
) {
    /**
     * Выбор цветовой схемы в зависимости от текущего режима.
     */
    val colorScheme = if (darkTheme) DarkColorScheme else LightColorScheme

```

```

/**
 * Применение Material-темы с настройками:
 * @param colorScheme Выбранная цветовая схема
 * @param typography Настройки типографики
 * @param content Пользовательский интерфейс
 */
MaterialTheme(
    colorScheme = colorScheme,
    typography = Typography,
    content = content
)
}

```

Объявление зависимостей между активностями и разрешения для уведомления:

Листинг 2.12. – AndroidManifest.xml

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapplication">
    <uses-permission android:name="android.permission.POST_NOTIFICATIONS" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportRtl="true"
        android:theme="@style/Theme.MyApp">

        <!-- Главная активность - точка входа в приложение -->
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- Вторая активность - не является точкой входа -->
        <activity
            android:name=".SecondActivity"
            android:exported="false" />
    </application>
</manifest>

```

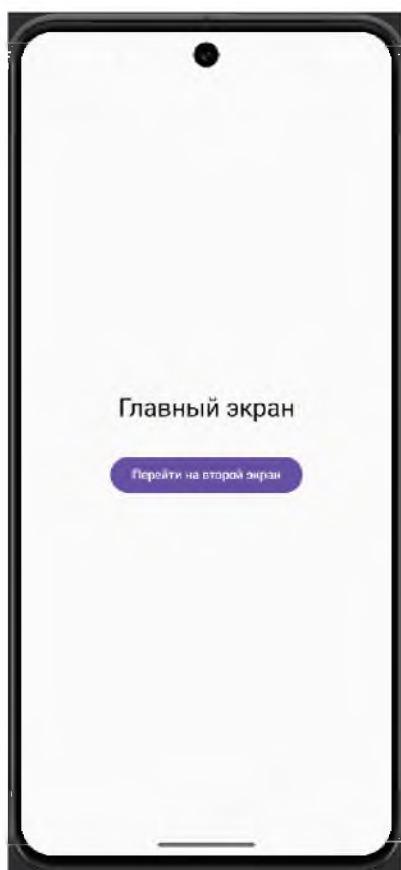


Рисунок 2.12 — Главный экран приложения

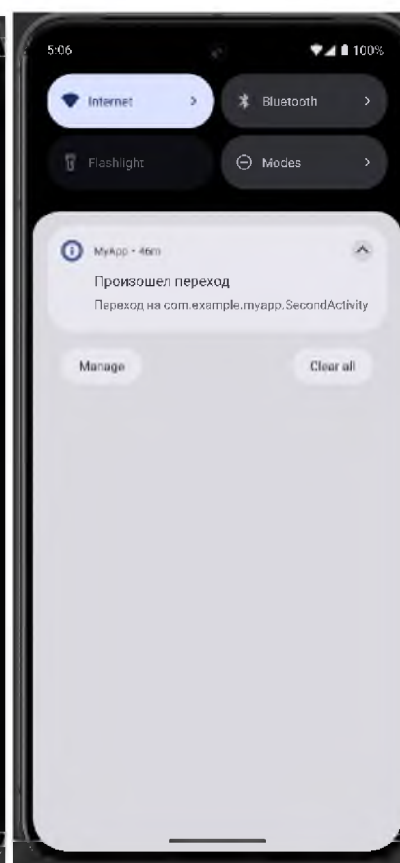
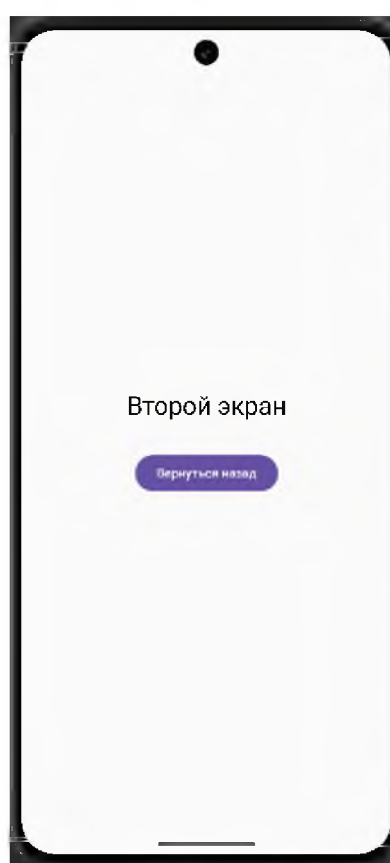


Рисунок 2.13 — Второй экран приложения и уведомление о переходе на второй экран

Задание

1. Продолжить разработку проекта из лабораторной работы 1.
2. Добавить в приложение несколько активностей.
3. Добавить в приложение уведомление Notification и Snackbar о переходе на новую активность и возвращении на главный экран.
4. Добавить в приложение переход к активности через уведомления.

Вариант	Тема
1	Путеводитель по музеям. Включить возможность создания личных заметок о посещенных музеях с прикреплением фотографий. Добавить открытие карты для поиска музеев
2	Садоводческий помощник. Включить возможность создания личных заметок о растениях с прикреплением фотографий. Добавить календарь с напоминаниями о поливе, подкормке и других уходных процедурах
3	Кулинарная книга.

	Добавить пошаговый режим приготовления с таймером для каждого шага. Реализовать уведомления о необходимости купить ингредиенты для запланированных блюд
4	Личный финансовый помощник. Добавить аналитику расходов при помощи графика с категориями. Реализовать уведомления о приближении лимита бюджета
5	Тренировочный план для домашних питомцев. Добавить видеоинструкции для упражнений. Реализовать напоминания о тренировках
6	Архив личных воспоминаний. Добавить возможность прикрепления фотографий и даты к воспоминаниям. Реализовать уведомления о годовщинах важных событий
7	Каталог фильмов и сериалов. Реализовать уведомления о выходе новых сезонов или похожих фильмах. Включить рекомендации на основе просмотренного
8	Организатор задач для студентов. Реализовать напоминания о дедлайнах и экзаменах. Добавить возможность отмечать сделанные и запланированные задания
9	Дневник настроения. Добавить график изменения настроения за неделю/месяц. Реализовать уведомления с мотивационными цитатами в плохие дни
10	Организатор мероприятий. Добавить возможность приглашения друзей через соцсети или email. Реализовать напоминания о предстоящих событиях за день/час

Ход работы

1. Изучить теоретический материал.
2. Выполнить задания.
3. Оформить отчет.
4. Ответить на контрольные вопросы.
5. Загрузить отчет в Classroom.
6. Защитить работу у преподавателя.

Контрольные вопросы

1. Что такое Navigation Component в Android? Какие преимущества он предоставляет?
2. Какие основные элементы включает граф навигации в Navigation Component?
3. Как создать и использовать NavHostController в Jetpack Compose?
4. Какие методы навигации поддерживает NavController? Чем они отличаются?
5. Для чего нужен параметр launchSingleTop в навигации?

6. Как передавать данные между экранами с помощью Navigation Component?
7. Что такое каналы уведомлений (Notification Channels)? Как их создать?
8. Какие разрешения необходимы для отображения уведомлений в Android 13 и выше?
9. Как добавить кнопки действий (actions) в уведомление?
10. Чем отличается NotificationCompat.Builder от Notification.Builder?
11. Как реализовать глубокие ссылки (deep links) в Navigation Component?
12. Какие анимации переходов между экранами поддерживает Navigation Component?
13. Как сохранить состояние экрана при навигации между фрагментами или активностями?
14. Какие виды уведомлений поддерживает Android, и в чем их особенности?
15. Как создать и отобразить Snackbar в Jetpack Compose?

Содержание отчета

1. Титульный лист.
2. Индивидуальное задание.
3. Краткое описание стратегии решения.
4. Листинг программы.
5. Тестовые примеры, результаты тестирования.
6. Выводы.

ЛАБОРАТОРНАЯ РАБОТА 3

Вывод, обработка и хранение данных в локальной базе данных

Цель работы: ознакомиться с этапами создания базы данных в Android приложениях; реализовать CRUD-операции для работы с SQLite.

Теоретическая часть

I. Базы данных в android приложениях.

Большинство приложений имеют данные, которые необходимо хранить, обновлять и удалять, например, плейлист песен, элементы списка дел, записи расходов и доходов и т.д. Для таких случаев необходимо интегрировать базу данных в Android приложение.

Room – это уровень абстракции поверх базы данных SQLite. SQLite использует специализированный язык (SQL) для выполнения операций с базой данных. Вместо того чтобы использовать SQLite напрямую, Room упрощает рутинные задачи по настройке базы данных, конфигурированию и взаимодействию с приложением. Room также обеспечивает проверки SQLite во время компиляции.

Уровень абстракции – это набор функций, скрывающих базовую реализацию/сложность. Он предоставляет интерфейс к существующему набору функциональности, например, SQLite.

II. Обзор стартового приложения.

Стартовое приложение используется как шаблон, в который пошагово будет добавляться база данных и функционал для работы с ней.

Для использования стартового приложения необходимо клонировать репозиторий GitHub:

```
$ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app.git
$ cd basic-android-kotlin-compose-training-inventory-app
$ git checkout starter
```

Либо загрузить архив по ссылке: <https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app/archive/refs/heads/starter.zip>

После установки:

1. Откройте проект со стартовым кодом в Android Studio.
2. Запустите приложение на устройстве Android или эмуляторе. Убедитесь, что эмулятор или подключенное устройство работает с API уровня 26 или выше.

3. Проверьте, что приложение не отображает данные об инвентаре.

4. Нажмите плавающую кнопку действия (FAB), которая позволяет добавлять новые элементы в базу данных.

Приложение перейдет на новый экран, где вы сможете ввести данные о новом элементе (рисунок 3.1).

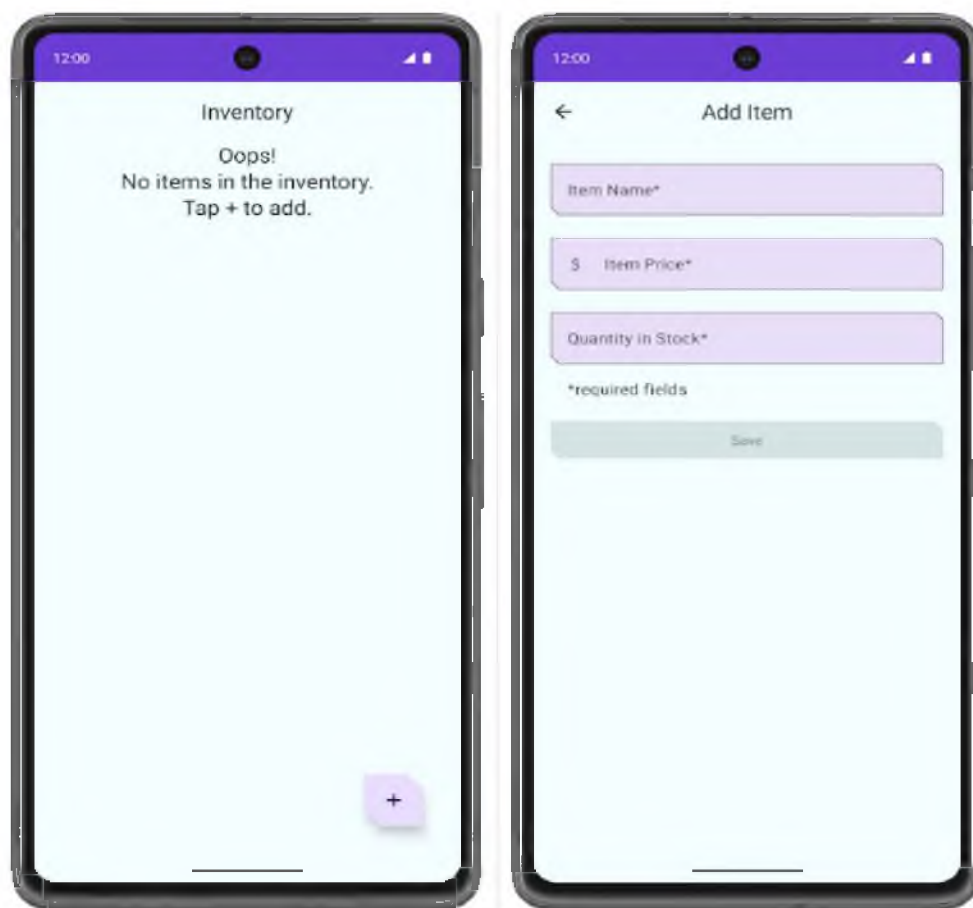


Рисунок 3.1. – Демонстрация работы стартового приложения

Руководство по некоторым файлам проекта:

1. `ui/home/HomeScreen.kt`

Этот файл – главный экран или первый экран в приложении (рисунок 3.2), содержащий компоуемые элементы для отображения списка инвентаря. Он имеет FAB + для добавления новых элементов в список (рисунок 3.3). В этом файле определяются компоуемые элементы (UI-компоненты), ответственные за отображение списка инвентаря.

2. `ui/item/ItemEntryScreen.kt`

Этот экран похож на `ItemEditScreen.kt`. Они оба имеют текстовые поля для деталей элемента. Этот экран отображается при нажатии FAB на главном экране. `ItemEntryViewModel.kt` соответствует `ViewModel` этому экрану.

3. ui/navigation/InventoryNavGraph.kt

Этот файл представляет собой навигационный график для всего приложения.

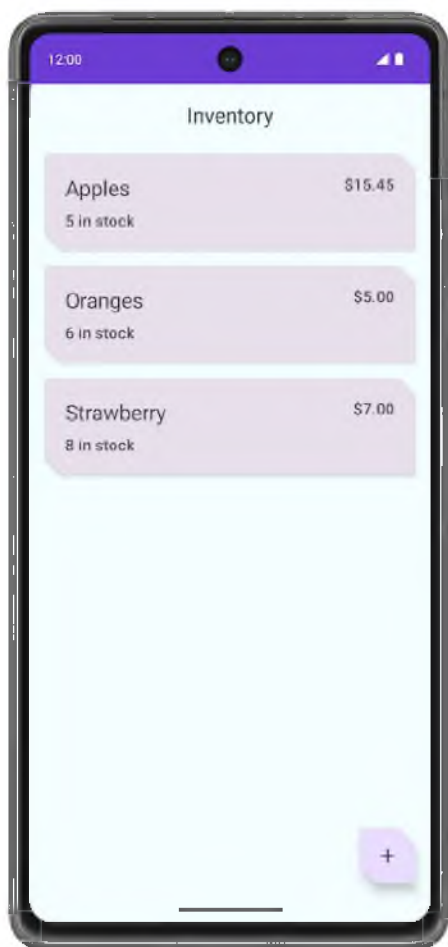


Рисунок 3.2. – Демонстрация главного экрана приложения

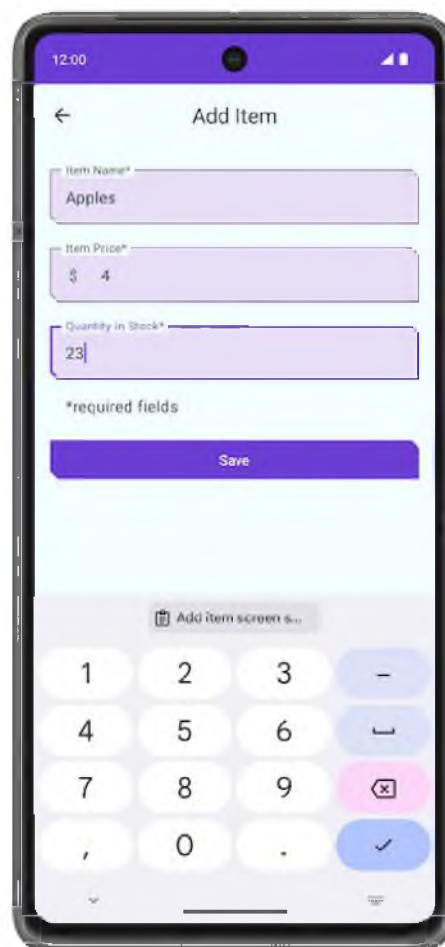


Рисунок 3.3. – Добавление нового элемента

III. Основные компоненты Room.

В языке программирования Kotlin для работы с данными предусмотрены специальные классы данных (data classes), которые обеспечивают удобный и лаконичный способ обработки информации в оперативной памяти. Однако при необходимости сохранения данных возникает потребность в их преобразовании в формат, совместимый с системами хранения информации.

Для организации структурированного хранения данных требуется:

- создание таблиц, соответствующих структуре сохраняемых данных;
- реализация запросов, обеспечивающих:
 - доступ к хранимым данным;
 - модификацию содержимого хранилища;
 - управление данными в соответствии с бизнес-логикой приложения.

Данный подход обеспечивает надежное хранение информации и эффективный доступ к ней при последующей работе приложения.

Три ключевых компонента Room обеспечивают бесперебойную работу с данными:

1. Сущности (Entities) Room: представляют таблицы в базе данных приложения. Позволяют:

- обновлять данные в существующих строках таблиц;
- создавать новые строки для вставки данных.

2. DAO (Data Access Objects) Room: предоставляют методы для выполнения операций с базой данных:

- получение данных;
- обновление информации;
- добавление новых записей;
- удаление данных.
- Класс Room Database: является основным классом базы данных, который:
 - предоставляет экземпляры DAO;
 - обеспечивает связь между DAO и конкретной базой данных.

Более детальная реализация и изучение данных компонентов будет рассмотрена далее. На представленной схеме (рисунок 3.4) демонстрируется взаимодействие компонентов Room при работе с базой данных.

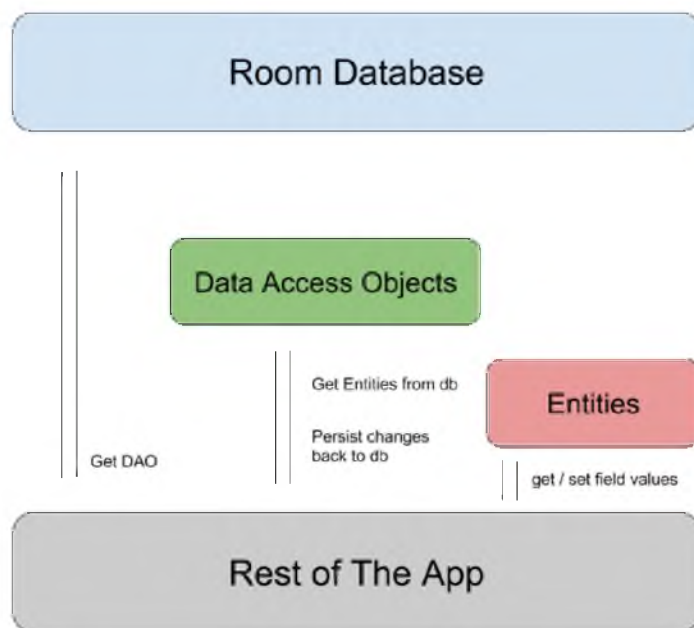


Рисунок 3.4. – Схема работы потока данных

Добавление зависимостей Room:

1. Откройте файл gradle на уровне модуля build.gradle.kts (Module: app) (рисунок 3.5).

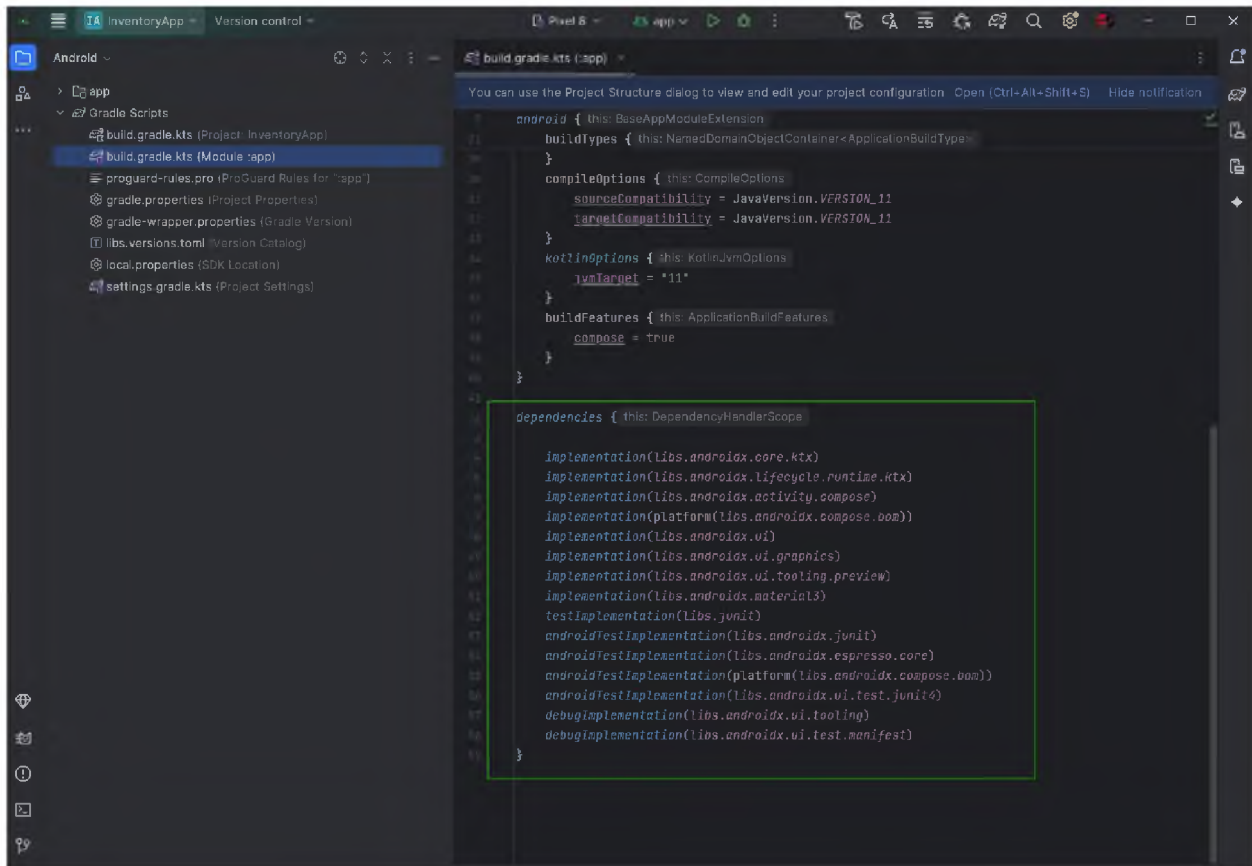


Рисунок 3.5. – gradle на уровне модуля build.gradle.kts

2. В dependencies блоке (см. рисунок 3.5) добавьте зависимости для библиотеки Room, показанные в следующем коде:

```
// Подключение основной библиотеки Room (рантайм) - содержит все основные классы и аннотации
implementation("androidx.room:room-runtime:${rootProject.extra["room_version"]}")

// Подключение компилятора Room для обработки аннотаций (@Entity, @Dao и т.д.) и генерации кода
ksp("androidx.room:room-compiler:${rootProject.extra["room_version"]}")

// Подключение Kotlin-расширения для Room (поддержка Coroutines, Flow и других Kotlin-фич)
implementation("androidx.room:room-ktx:${rootProject.extra["room_version"]}")
```

KSP – это мощный и в то же время простой API для анализа аннотаций Kotlin.

IV. Создание элемента ENTITY.

Класс Entity определяет структуру таблицы базы данных, где каждый экземпляр данного класса представляет отдельную строку в этой таблице. Класс Entity содержит специальные аннотации, которые указывают фреймворку Room (рисунок 3.6):

- как следует представлять данные в базе данных;
- каким образом взаимодействовать с этой информацией.

В контексте нашего приложения, Entity хранит данные о товарных позициях, включая:

- наименование товара;
- цену товара;
- количество доступных единиц.

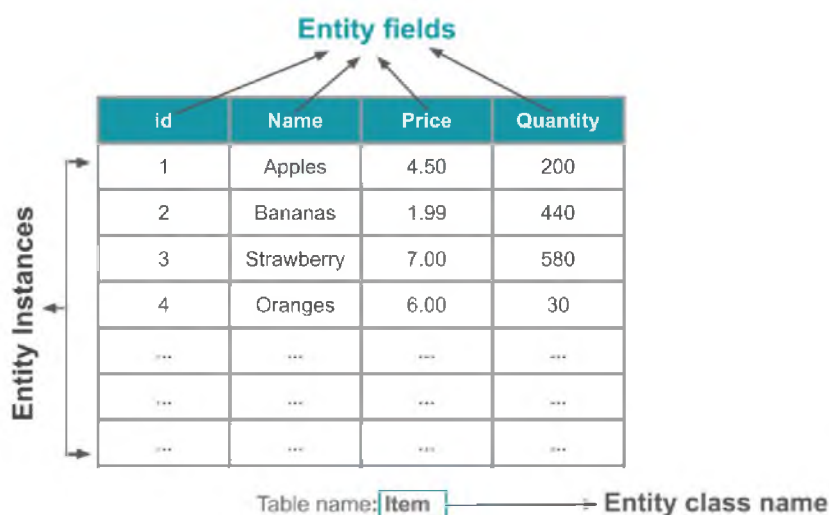


Рисунок 3.6. – Связь между реляционной базой данных и Entity классом в контексте библиотеки Room

Аннотация @Entity обозначает класс как сущность базы данных (Entity class). Для каждого такого класса приложение автоматически создает соответствующую таблицу в базе данных.

Основные характеристики:

- Структура таблицы.
 - Каждое поле класса Entity становится столбцом в таблице (если не указано иное, см. документацию по Entity).
- Первичный ключ (Primary Key).
 - Каждый экземпляр Entity в базе данных обязан иметь первичный ключ.
 - Первичный ключ обеспечивает уникальную идентификацию каждой записи.
 - После назначения ключ нельзя изменить - он сохраняется на весь срок существования объекта в базе.

В рамках данной задачи необходимо:

1. Создать класс Entity.
2. Определить поля для хранения информации о товарных позициях:
 - Int – для первичного ключа;
 - String – для наименования товара;
 - Double – для цены товара;
 - Int – для количества на складе.

Для этого:

1. Откройте стартовый проект в Android Studio.
2. Откройте Kotlin класс в папке data, который представляет сущность базы данных в вашем приложения, и добавьте следующие строки кода:

```
class Item(  
    val id: Int,  
    val name: String,  
    val price: Double,  
    val quantity: Int  
)
```

Вместо обычных классов для работы с базой данных чаще используются классы данных.

Классы данных (Data classes) в Kotlin предназначены для хранения и обработки информации. Они объявляются с использованием ключевого слова data.

Ключевые особенности:

1. Автоматическая генерация методов.

Компилятор Kotlin автоматически реализует стандартные служебные функции для работы с данными:

- toString() – преобразование в строковое представление;
 - copy() – создание копии объекта;
 - equals() – сравнение объектов.
2. Оптимизация разработки.

Использование классов данных позволяет:

- уменьшить объем шаблонного кода;
- повысить читаемость и поддерживаемость;
- обеспечить стандартизированное поведение объектов-контейнеров.

Для обеспечения корректности и предсказуемости генерируемого кода классы данных (data classes) в Kotlin должны соответствовать следующим обязательным требованиям:

1. Требования к первичному конструктору:

- должен содержать как минимум один параметр;
- все параметры первичного конструктора должны быть объявлены как `val` или `var`.

2. Ограничения на модификаторы:

- не могут быть объявлены как `abstract` (абстрактные);
- не могут быть объявлены как `open` (открытые для наследования);
- не могут быть объявлены как `sealed` (запечатанные).

Для реализации класса данных добавьте к уже существующему классу `Item` слово `data`:

```
data class Item(  
    val id: Int,  
    val name: String,  
    val price: Double,  
    val quantity: Int  
)
```

Над объявлением класса `Item` аннотируйте класс данных с помощью `@Entity`. Используйте `tableName` аргумент, чтобы задать `items` как имя таблицы SQLite:

```
import androidx.room.Entity  
  
@Entity(tableName = "items")  
data class Item(  
    ...  
)
```

Аннотируйте `id` свойство, `@PrimaryKey` чтобы сделать его `id` первичным ключом. Первичный ключ – это идентификатор, который уникально идентифицирует каждую запись/вход в вашей `Item` таблице:

```
import androidx.room.PrimaryKey  
  
@Entity(tableName = "items")  
data class Item(  
    @PrimaryKey  
    val id: Int,  
)
```

Назначьте id значение по умолчанию 0, которое необходимо для автоматической генерации id значений.

Добавьте autoGenerate параметр в @PrimaryKey аннотацию, чтобы указать, должен ли столбец первичного ключа быть сгенерирован автоматически. Если autoGenerate установлено значение true, то Room автоматически сгенерирует уникальное значение для столбца первичного ключа при вставке нового экземпляра сущности в базу данных. Это гарантирует, что каждый экземпляр сущности имеет уникальный идентификатор, без необходимости вручную назначать значения столбцу первичного ключа.

```
data class Item(  
    @PrimaryKey(autoGenerate = true)  
    val id: Int = 0,  
    // ...  
)
```

Теперь, когда класс Entity создан, можно переходить к созданию Data Access Object (DAO) для доступа к базе данных.

Объект доступа к данным (Data Access Object, DAO) представляет собой архитектурный шаблон, предназначенный для разделения уровня хранения данных от остальных компонентов приложения посредством предоставления абстрактного интерфейса (рисунок 3.7).

Ключевые характеристики:

1. Принцип единой ответственности:

- реализация DAO соответствует принципу единственной ответственности (Single Responsibility Principle), изученному в предыдущих практических занятиях;

- обеспечивает четкое разделение обязанностей между компонентами системы.

2. Основная функциональность:

- инкапсулирует всю сложность операций с базой данных;
- абстрагирует технические детали работы с уровнем persistence;
- предоставляет прозрачный интерфейс для манипуляции данными.

3. Преимущества использования:

- возможность модификации слоя данных без изменения бизнес-логики;
- упрощение тестирования компонентов;
- повышение гибкости архитектуры приложения;
- снижение степени связанности между компонентами системы.



Рисунок 3.7. – Архитектурный принцип доступа к данным в приложении

Рассмотрим создание DAO для Room. DAO – это основные компоненты Room, отвечающие за определение интерфейса, и предоставление удобных методов таких как `@Insert`, `@Delete` и `@Update`, которые выполняют добавление, удаление и обновление без необходимости написания оператора SQL (рисунок 3.8).

Для приложения «Инвентаризация» вам необходимо иметь возможность выполнять следующие действия:

- вставьте или добавьте новый элемент;
- обновите существующий товар, чтобы обновить название, цену и количество;
- получите определенный элемент на основе его первичного ключа id;
- соберите все предметы, чтобы вы могли их продемонстрировать;
- удалите запись в базе данных.

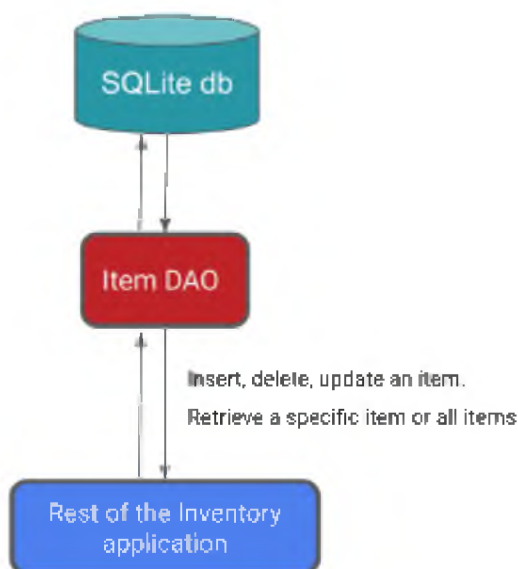


Рисунок 3.8. – Реализация архитектуры DAO для приложения «Инвентаризация»

Чтобы реализовать элемент DAO в приложении, выполните следующие действия:

1. Создайте интерфейс Kotlin ItemDao.kt в папке data.
2. Добавьте к интерфейсу аннотации @Dao:

```
import androidx.room.Dao

@Dao
interface ItemDao {
}
```

3. Внутри тела интерфейса добавьте @Insert аннотацию.

4. Ниже @Insert добавьте insert() в функцию, которая принимает экземпляр класса Entity в item в качестве аргумента.

5. Пометьте функцию ключевым словом suspend, чтобы разрешить ей выполняться в отдельном потоке.

Room не разрешает доступ к базе данных в основном потоке, т.к. операции с базой данных могут быть требовательными и привести к ошибке работы приложения. Для решения этих проблем операции с базой данных необходимо вынести в отдельный поток:

```
import androidx.room.Insert

@Insert
suspend fun insert(item: Item)
```

В процессе добавления записей в базу данных могут возникать конфликтные ситуации. Типичный сценарий:

- несколько компонентов системы одновременно пытаются изменить одну и ту же сущность (строку в таблице);
- параллельные операции содержат противоречащие значения (например, одинаковые первичные ключи).

6. Добавьте аргумент onConflict и присвойте ему значение OnConflictStrategy.IGNORE для игнорирования добавления новых элементов с конфликтующими значениями (например, с одинаковыми первичными ключами добавиться только первый элемент, второй будет проигнорирован):

```
import androidx.room.OnConflictStrategy

@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun insert(item: Item)
```

Теперь Room генерирует весь необходимый код для вставки item в базу данных. При вызове любой из функций DAO, отмеченных аннотациями Room, Room выполняет соответствующий SQL-запрос в базу данных. Например, при вызове указанного выше метода insert() Room выполняет SQL-запрос для вставки записи в базу данных.

7. Добавьте новую функцию с @Update аннотацией, которая принимает Item параметр as.

Обновляемая запись имеет тот же первичный ключ, что был до обновления.

8. Подобно методу insert(), отметьте эту функцию ключевым suspend словом:

```
import androidx.room.Update

@Update
suspend fun update(item: Item)
```

9. Добавьте еще одну функцию с @Delete аннотацией для удаления элементов и сделайте ее функцией приостановки.

@Delete аннотация удаляет элемент или списка элементов. Необходимо передать записи, которые нужно удалить:

```
import androidx.room.Delete

@Delete
suspend fun delete(item: Item)
```

Для оставшейся функциональности нет удобных аннотаций, поэтому придется использовать @Query аннотации и предоставлять запросы SQLite.

10. Напишите запрос SQLite для извлечения определенного элемента из таблицы элементов на основе заданного id. Следующий код представляет собой пример запроса, который выбирает все столбцы из items, где id соответствует определенному значению и id является уникальным идентификатором:

```
@Query("SELECT * from items WHERE id = :id")
```

11. После @Query аннотации добавьте getItem() функцию, которая принимает Int аргумент и возвращает Flow<Item>:

```
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Query("SELECT * from items WHERE id = :id")
fun getItem(id: Int): Flow<Item>
```

Лучше всего использовать Flow для работы с базой данных в Room по следующим причинам:

- Flow автоматически получает изменения из БД без повторных запросов;
- Room сам выполняет запросы в фоновом потоке и обновляет UI при изменениях;
- оптимальная работа с памятью и потоками.

12. Добавьте аннотацию `@Query` с `getAllItems()` функцией. Запрос SQLite должен возвращать все столбцы таблицы `item`, упорядоченные по возрастанию. `getAllItems()` возвращает список `Item`, т.к. Flow. Room обновляется Flow, что означает, что данные нужно получить явно только один раз:

```
@Query("SELECT * from items ORDER BY name ASC")  
fun getAllItems(): Flow<List<Item>>
```

V. Создание экземпляра базы данных ROOM.

В рамках данной задачи требуется реализовать класс `RoomDatabase`, который будет использовать ранее определенные `Entity` и `DAO`. Класс базы данных определяет перечень сущностей и объектов доступа к данным.

Назначение класса `Database`:

1. Предоставляет приложению экземпляры `DAO`.
2. Через `DAO` приложение получает доступ к данным в виде объектов соответствующих сущностей.

3. Позволяет выполнять операции:

- обновление существующих записей;
- создание новых строк в таблицах базы данных.

Процедура создания экземпляра `RoomDatabase`:

1. Определение абстрактного класса:

- класс должен быть объявлен как `public abstract`;
- наследоваться от `RoomDatabase`;
- выступает в роли контейнера для базы данных;
- абстрактность класса обусловлена автоматической генерацией реализации средой Room.

2. Аннотирование класса:

- использовать аннотацию `@Database`;
- указать в параметрах:
 - список записей;
 - номер версии базы данных.

3. Определение методов доступа:

- реализовать абстрактный метод/свойство, возвращающее экземпляр ItemDao;

- реализация будет сгенерирована автоматически.

4. Паттерн Singleton:

- требуется единственный экземпляр базы данных на все приложение;

- реализовать через Room.databaseBuilder;

- создавать базу данных item_database только при отсутствии существующего экземпляра;

- возвращать существующий экземпляр при повторных обращениях.

Данная реализация обеспечивает корректную работу с базой данных на протяжении всего жизненного цикла приложения.

Для реализации базы данных в нашем проекте необходимо следующее:

1. Создать класс Kotlin InventoryDatabase.kt в папке data проекта.

2. В файл InventoryDatabase.kt добавить абстрактный класс InventoryDatabase, расширяющий RoomDatabase, и аннотацию @Database:

```
import androidx.room.Database
import androidx.room.RoomDatabase

@Database
abstract class InventoryDatabase : RoomDatabase() {}
```

Для построения базы данных аннотации @Database требуется несколько аргументов Room.

3. Укажите Item как единственный класс в списке entities. Установите version как 1. Всякий раз, когда вы меняете схему таблицы базы данных, вам придется увеличивать номер версии. Установите exportSchema этот параметр false, чтобы не сохранять резервные копии истории версий схемы:

```
@Database(entities = [Item::class], version = 1, exportSchema = false)
```

4. Внутри тела класса объявите абстрактную функцию, которая возвращает, ItemDao чтобы база данных знала о DAO:

```
abstract fun itemDao(): ItemDao
```

5. Ниже абстрактной функции определите companion object, которая обеспечивает доступ к методам создания или получения базы данных и использует имя класса в качестве квалификатора:

```
companion object {}
```

Внутри companion объекта объявите частную переменную, допускающую значение NULL, Instance для базы данных и инициализируйте ее значением null.

Переменная Instance сохраняет ссылку на базу данных, когда она была создана. Это помогает поддерживать единственный экземпляр базы данных, открытый в определенное время, что является дорогим ресурсом для создания и обслуживания.

6. Аннотировать Instances помощью @Volatile.

Значение переменной volatile никогда не кэшируется, и все операции чтения и записи выполняются в основную память и из нее. Эти функции помогают гарантировать, что значение Instance всегда актуально и одинаково для всех потоков выполнения. Это означает, что изменения, внесенные одним потоком в Instance немедленно видны всем остальным потокам:

```
@Volatile
private var Instance: InventoryDatabase? = null
```

7. Определите стратегию миграции базы данных. При изменении структуры базы данных (добавлении новых полей, таблиц) необходимо обеспечить плавное обновление без потери данных:

```
// Пример миграции с версии 1 на версию 2 при добавлении нового поля "category"
private val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        // Добавляем новое поле в существующую таблицу
        database.execSQL("ALTER TABLE items ADD COLUMN category TEXT NOT NULL DEFAULT
'General'")
    }
}

// Пример миграции с версии 2 на версию 3 при добавлении поля "discount"
private val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE items ADD COLUMN discount REAL DEFAULT 0.0")
        // Обновляем существующие записи - устанавливаем скидку для товаров дороже 20
        database.execSQL("UPDATE items SET discount = 0.1 WHERE price > 20.0")
    }
}
```

8. Ниже Instance, оставаясь внутри companion объекта, определите getDatabase() метод с Context параметром, который необходим конструктору базы данных. Возвращает тип InventoryDatabase (появляется сообщение об ошибке, поскольку getDatabase() пока ничего не возвращает):

```
import android.content.Context

fun getDatabase(context: Context): InventoryDatabase {}
```

Несколько потоков могут потенциально запросить экземпляр базы данных одновременно, что приводит к появлению двух баз данных вместо одной. Эта проблема известна как состояние гонки. Оборачивание кода для получения базы данных внутри `synchronized` блока означает, что только один поток выполнения за раз может войти в этот блок кода, что гарантирует, что база данных будет инициализирована только один раз. Используйте `synchronized{}` блок, чтобы избежать состояния гонки.

9. Внутри `getDatabase()` верните `Instance` переменную – или; если `Instance` она равна `null`, инициализируйте ее внутри `synchronized{}` блока. Используйте оператор `elvis( ?: )`, чтобы сделать это. Передайте в `this`, сопутствующий объект. Вы исправите ошибку на последующих шагах:

```
return Instance ?: synchronized(this) { }
```

10. Внутри синхронизированного блока используйте конструктор баз данных, чтобы получить базу данных:

```
import androidx.room.Room

val instance = Room.databaseBuilder(
    context.applicationContext,
    InventoryDatabase::class.java,
    "item_database"
)
```

11. Добавьте стратегию миграции в конструктор. Для разработки можно использовать `.fallbackToDestructiveMigration()`, но для продакшена рекомендуется явно указывать все миграции:

```
// Для разработки (удаляет данные при изменении версии):
.fallbackToDestructiveMigration()

// Для продакшена (сохраняет данные):
.addMigrations(MIGRATION_1_2, MIGRATION_2_3)
```

12. Чтобы создать экземпляр базы данных, вызовите `.build()`. После этого `build()` добавьте also блок и назначьте его `Instance = it` для сохранения ссылки на недавно созданный экземпляр базы данных:

```
.build()  
Instance = instance  
instance // возвращаем созданный экземпляр
```

Важное примечание по миграциям:

- при каждом изменении структуры базы данных увеличивайте номер версии;
- для каждого изменения версии создавайте соответствующую миграцию;
- миграции выполняются автоматически при обновлении приложения;
- всегда тестируйте миграции на тестовых данных перед выпуском обновления.

VI. Реализация классов репозитория для работы с источником данных.

Для реализации методов `get`, `insert`, `delete`, и `update` для базы данных выполните следующие шаги:

1. В файле `ItemsRepository.kt` создайте интерфейс с функциями:

```
import kotlinx.coroutines.flow.Flow  
  
/**  
 * Репозиторий для операций добавления, обновления, удаления и получения [Item] из указанного источника данных.  
 */  
interface ItemsRepository {  
    /**  
     * Получить все элементы из источника данных.  
     */  
    fun getAllItemsStream(): Flow<List<Item>>  
  
    /**  
     * Получить элемент по указанному [id] из источника данных.  
     */  
    fun getItemStream(id: Int): Flow<Item?>  
  
    /**  
     * Добавить элемент в источник данных  
     */  
    suspend fun insertItem(item: Item)  
  
    /**
```

```

    * Удалить элемент из источника данных
    */
    suspend fun deleteItem(item: Item)

    /**
    * Обновить элемент в источнике данных
    */
    suspend fun updateItem(item: Item)
}

```

Создание интерфейса `ItemsRepository` обусловлено необходимостью соблюдения архитектурных принципов разработки программного обеспечения.

Ключевые преимущества:

- инкапсуляция реализации – абстрагирует уровень данных от бизнес-логики приложения;
- соблюдение принципа инверсии зависимостей – модули высокого уровня не зависят от модулей низкого уровня;
- обеспечение тестируемости – позволяет использовать mock-реализации для модульного тестирования;
- гибкость архитектуры – упрощает модификацию источника данных без изменения кода приложения;
- стандартизация доступа к данным – определяет единый контракт для операций CRUD.

Интерфейс служит формальным соглашением между уровнем данных и уровнем представления, обеспечивая стабильность API при возможном изменении реализации хранилища данных.

Данный подход соответствует современным стандартам разработки Android-приложений и рекомендациям Android Jetpack.

2. Передайте параметр конструктора типа `ItemDao` классу `OfflineItemsRepository` в файле `OfflineItemsRepository.kt`. В этом же классе переопределите функции из интерфейса `ItemsRepository`, и вызовите соответствующие функции из `ItemDao`:

```

import kotlinx.coroutines.flow.Flow

class OfflineItemsRepository(private val itemDao: ItemDao) : ItemsRepository {
    override fun getAllItemsStream(): Flow<List<Item>> = itemDao.getAllItems()

    override fun getItemStream(id: Int): Flow<Item?> = itemDao.getItem(id)

    override suspend fun insertItem(item: Item) = itemDao.insert(item)
}

```

```
override suspend fun deleteItem(item: Item) = itemDao.delete(item)

override suspend fun updateItem(item: Item) = itemDao.update(item)
}
```

Класс `OfflineItemsRepository` реализует интерфейс `ItemsRepository` и инкапсулирует работу с локальным источником данных через объект доступа к данным (DAO).

Ключевые аспекты реализации:

- инъекция зависимостей – конструктор принимает экземпляр `ItemDao` для доступа к данным;
- реализация контракта – класс предоставляет конкретную реализацию методов, объявленных в интерфейсе;
- делегирование операций – каждый метод репозитория перенаправляет вызов соответствующему методу DAO.

3. Создайте экземпляр базы данных, создав файл `AppContainer.kt` в пакете `data`, и реализуйте передачу DAO.

```
class AppContainer(private val context: Context) {
    /**
     * Ленивая инициализация репозитория с автоматической передачей зависимостей
     * - Создается при первом обращении
     * - Гарантирует единственный экземпляр на все приложение
     */
    override val itemsRepository: ItemsRepository by lazy {
        OfflineItemsRepository(
            InventoryDatabase.getDatabase(context).itemDao()
        )
    }
}
```

Класс `AppContainer` является контейнером зависимостей, который централизованно управляет созданием и предоставлением объектов данных для всего приложения.

Принципы работы:

- инициализация базы данных – получает доступ к DAO через `InventoryDatabase.getDatabase(context).itemDao()`;
- ленивое создание репозитория – использует `by lazy` для отложенной инициализации при первом обращении;
- инъекция зависимостей – передает созданный DAO в конструктор `OfflineItemsRepository`.

VII. Обновление ITEMENTRY VIEWMODEL.

Реализация передачи репозиторий в ItemEntryViewModel.kt файл и сохранение данных, выведенных на экране при добавлении новой записи.

1. Обратите внимание на validateInput() – закрытую функцию в ItemEntryViewModel классе. Вышеуказанная функция проверяет, являются ли name, price и quantity пустыми. Вы используете эту функцию для проверки ввода пользователя перед добавлением или обновлением сущности в базе данных.

2. Откройте ItemEntryViewModel класс и добавьте private параметр конструктора по умолчанию типа ItemsRepository:

```
import com.example.inventory.data.ItemsRepository

class ItemEntryViewModel(private val itemsRepository: ItemsRepository) : ViewModel() {
}
```

3. Обновите initializer модель представления записи элемента в ui/AppViewModelProvider.kt и передайте экземпляр репозитория в качестве параметра:

```
object AppViewModelProvider {
    val Factory = viewModelFactory {
        // Other Initializers
        // Initializer for ItemEntryViewModel
        initializer {
            ItemEntryViewModel(inventoryApplication().container.itemsRepository)
        }
        //...
    }
}
```

4. Перейдите в файл ItemEntryViewModel.kt и добавьте в конце класса ItemEntryViewModel функцию приостановки, которая вызывается saveItem() для вставки элемента в базу данных Room. Эта функция добавляет данные в базу данных неблокирующим способом. Внутри функции проверьте, itemUiState является ли она допустимой, и преобразуйте ее в Item тип, чтобы Room мог распознать данные. Вызовите insertItem() и передайте itemsRepository данные. Пользовательский интерфейс вызывает эту функцию, чтобы добавить детали элемента в базу данных:

```
suspend fun saveItem() {
    if (validateInput()) {
        itemsRepository.insertItem(itemUiState.itemDetails.toItem())
    }
}
```

Чтобы связать все вместе, добавьте обработчик щелчков к кнопке Save. Внутри обработчика щелчков вы запускаете сопрограмму и вызываете `saveltem()` сохранение данных в базе данных Room.

5. Внутри составной функции `ItemEntryScreen` файла `ItemEntryScreen.kt`, создайте `val` именованный объект `coroutineScope` с составной функцией `rememberCoroutineScope()`:

```
import androidx.compose.runtime.rememberCoroutineScope

val coroutineScope = rememberCoroutineScope()
```

`rememberCoroutineScope()` – это компонуемая функция, которая возвращает привязку к композиции `CoroutineScope`, где она вызывается. Вы можете использовать компонуемую функцию `rememberCoroutineScope()`, когда хотите запустить сопрограмму вне компонуемой и гарантировать, что сопрограмма будет отменена после того, как область действия покинет композицию.

6. Обновите вызов функции и запустите сопрограмму внутри лямбда-выражения `.ItemEntryBody().onSaveClick` и вызовите для `viewModel.saveltem()` сохранение элемента в базе данных:

```
ItemEntryBody(
    // ...
    onSaveClick = {
        coroutineScope.launch {
            viewModel.saveltem()
        }
    },
    modifier = modifier.padding(innerPadding)
)
```

Обратите внимание, что вы не использовали `viewModelScope.launch()` для `saveltem()` в файле `ItemEntryViewModel.kt`, однако это необходимо в `ItemEntryBody()` при вызове методов репозитория.

Ключевые моменты:

- Ограничения `suspend`-функций:
 - вызывать `suspend`-функции можно только из `Coroutines` или другой `suspend`-функции;
 - функция `viewModel.saveltem()` является `suspend`-функцией, поэтому требует соблюдения этого правила.
- Правильный вызов из UI:
 - в `ItemEntryBody()` вызов должен быть обернут в `Coroutines`.

Почему это важно:

- без `viewModelScope.launch` вызов `suspend`-функции приведет к ошибке компиляции;

- `viewModelScope` автоматически управляет жизненным циклом `Coroutines`, отменяя ее при очистке `ViewModel`.

7. Создайте и запустите свое приложение.

8. На экране «Добавить элемент» добавьте сведения об элементе и нажмите «Сохранить». Обратите внимание, что нажатие кнопки «Сохранить» не закрывает экран «Добавить элемент» (рисунок 3.9).

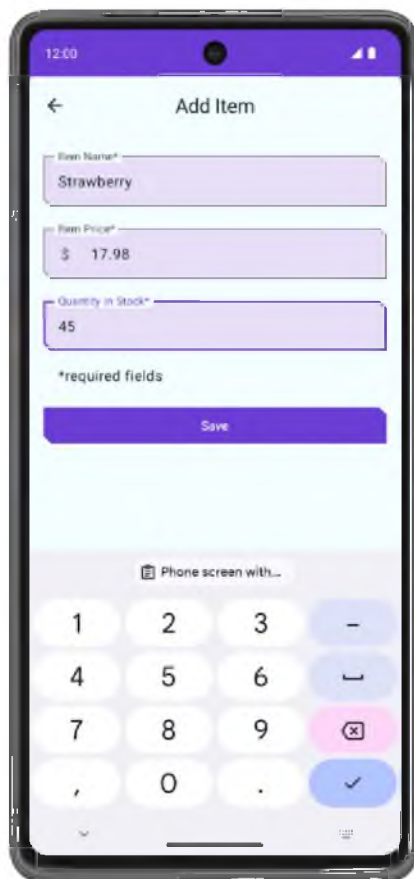


Рисунок 3.9. – Экран добавления элемента

9. В `onSaveClick` лямбде добавьте вызов `navigateBack()` после вызова для `viewModel.saveItem()` перехода обратно на предыдущий экран. Ваша `ItemEntryBody()` функция выглядит как следующий код:

```
ItemEntryBody(  
    itemUiState = viewModel.itemUiState,  
    onItemValueChange = viewModel::updateUiState,  
    onSaveClick = {  
        coroutineScope.launch {
```

```
viewModel.saveItem()
navigateBack()
}
},
modifier = modifier.padding(innerPadding)
)
```

10. Запустите приложение снова и выполните те же шаги для ввода и сохранения данных. Обратите внимание, что на этот раз приложение возвращается на экран Inventory (рисунок 3.10).



Рисунок 3.10. – Главный экран приложения

Это действие сохраняет данные, но вы не можете видеть данные инвентаризации в приложении. В следующей задаче вы используете Database Inspector для просмотра сохраненных вами данных.

VIII. Просмотр содержимого базы данных с помощью DATABASE INSPECTOR.

Инспектор базы данных позволяет проверять, выполнять запросы и изменять базы данных вашего приложения во время его работы. Эта функция особенно полезна для отладки работы с базами данных.

Ключевые особенности:

– Совместимость:

- с нативным SQLite;
- библиотеками на основе SQLite (включая Room).

– Оптимальная работа на эмуляторах/устройствах с API уровня 26 и выше.

Важное ограничение: инспектор (рисунок 3.11) работает только с SQLite-библиотекой, встроенной в операционную систему Android (API ≥ 26), и не поддерживает сторонние SQLite-библиотеки, которые могут быть включены в ваше приложение.

Примечание о совместимости: для достижения наилучших результатов рекомендуется использовать эмуляторы или устройства с API уровня 26 или выше. Инспектор не функционирует с альтернативными SQLite-реализациями, которые могут быть добавлены в приложение отдельно.

1. Запустите свое приложение на эмуляторе или подключенном устройстве с API уровня 26 или выше, если вы этого еще не сделали.

2. В Android Studio выберите в строке меню «Вид» > «Окна инструментов» > «Проверка приложений».

3. Выберите вкладку «Инспектор базы данных».

4. В панели Database Inspector выберите com.example.inventory из раскрывающегося меню, если он еще не выбран. Item_database в приложении Inventory появится на панели Databases.

5. Разверните узел item_database на панели Databases и выберите Item to inspect. Если панель Databases пуста, используйте эмулятор, чтобы добавить несколько элементов в базу данных с помощью экрана Add Item.

6. Установите флажок «Обновления в реальном времени» в инспекторе базы данных, чтобы автоматически обновлять представляемые данные при взаимодействии с запущенным приложением в эмуляторе или на устройстве.

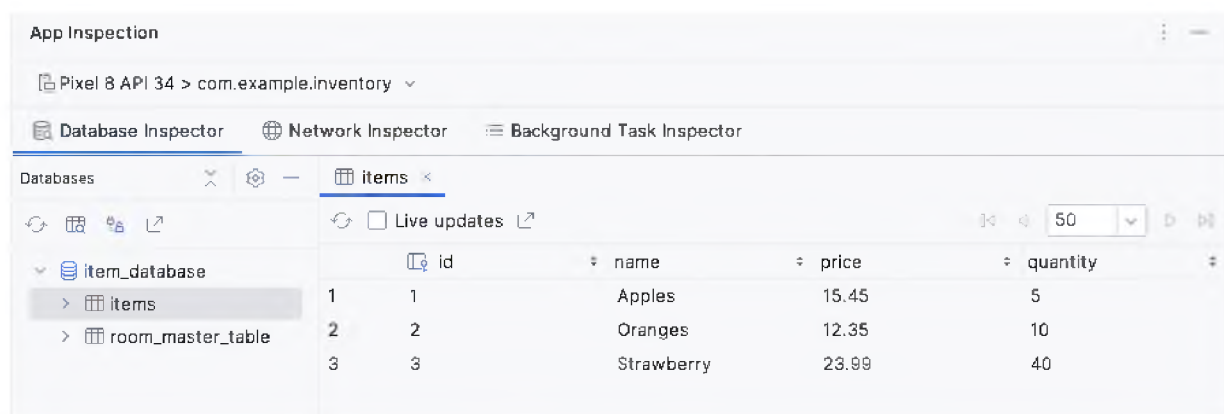


Рисунок 3.11. – Содержимое базы данных в окне Database Inspector

Итоговый вид приложения представлен на рисунке 3.12.

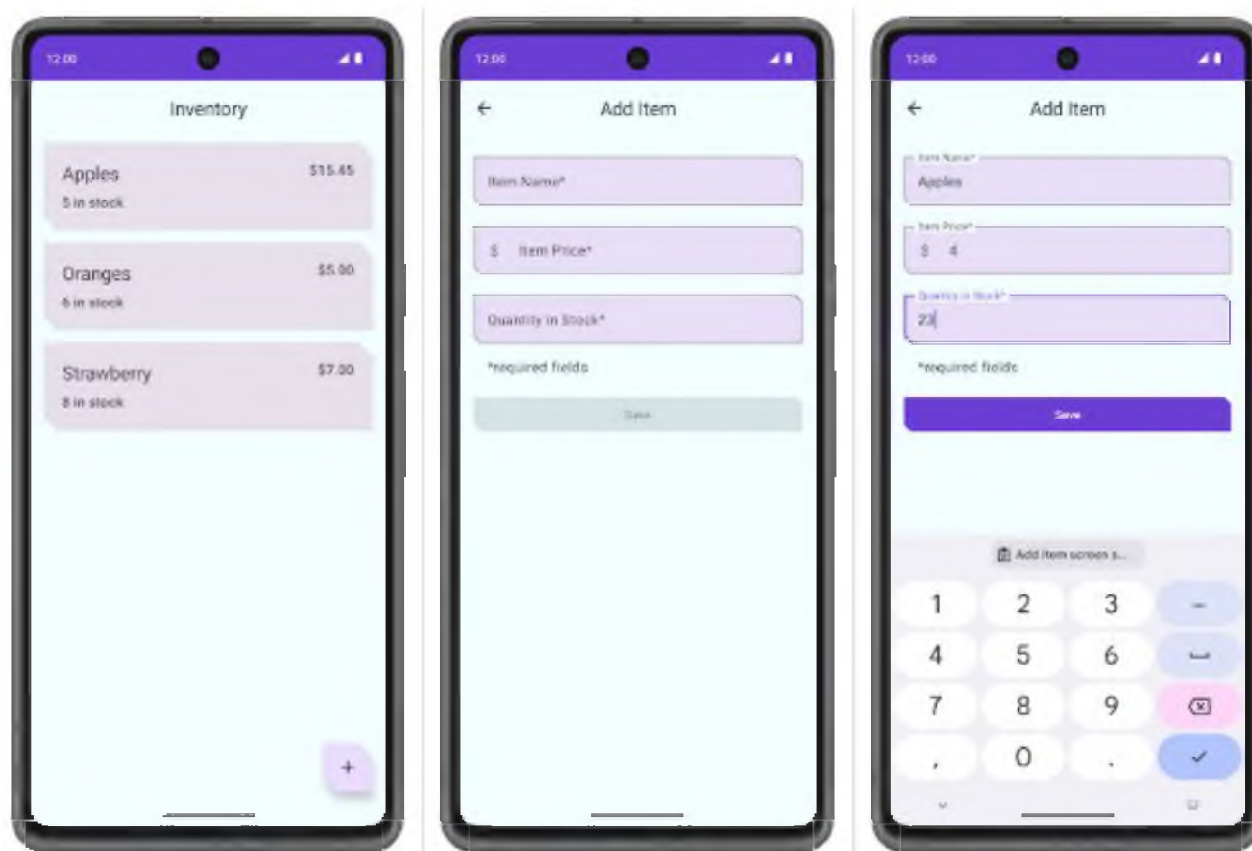


Рисунок 3.12. – Итоговый вид приложения

Задание

1. Продолжить разработку проекта из лабораторной работы 2.
2. Добавить в приложение несколько активностей.
3. Добавить в приложение уведомление Notification и Snackbar о переходе на новую активность и возвращении на главный экран.
4. Добавить в приложение переход к активности через уведомления.

Ход работы

1. Изучить теоретический материал.
2. Выполнить задания.
3. Оформить отчет.
4. Ответить на контрольные вопросы.
5. Загрузить отчет в Classroom.
6. Защитить работу у преподавателя.

Вариант	Тема
1	Путеводитель по музеям. Включить возможность создания личных заметок о посещенных музеях с прикреплением фотографий с использованием базы данных
2	Садоводческий помощник. Включить возможность создания личных заметок о растениях с прикреплением фотографий с использованием базы данных
3	Кулинарная книга. Добавить пошаговый режим приготовления с таймером для каждого шага. Сохранять готовые рецепты в базу данных
4	Личный финансовый помощник. Реализовать хранение финансовых операций в базе данных
5	Тренировочный план для домашних питомцев. Реализовать добавление, удаление и редактирование тренировок в базе данных
6	Архив личных воспоминаний. Реализовать добавление, удаление и редактирование воспоминаний в базе данных
7	Каталог фильмов и сериалов. Реализовать добавление, удаление и редактирование фильмов и сериалов в базе данных
8	Организатор задач для студентов. Реализовать добавление, удаление и редактирование задач для студентов в базе данных
9	Дневник настроения. Реализовать добавление, удаление и редактирование пометок о настроении в базе данных
10	Организатор мероприятий. Реализовать добавление, удаление и редактирование мероприятий для студентов в базе данных

Контрольные вопросы

1. Какие основные компоненты Room используются для работы с базой данных в Android?
2. Какой язык запросов используется в SQLite? Как Room упрощает работу с ним?
3. Какие аннотации Room используются для определения сущности (Entity) и какие обязательные параметры они требуют?
4. Почему класс Entity должен содержать первичный ключ? Как его можно автоматически генерировать?
5. Какие преимущества предоставляют классы данных (data classes) в Kotlin при работе с Room?

6. Какие требования предъявляются к классам данных для корректной работы с Room?
7. Что такое DAO (Data Access Object)? Какие методы он обычно включает?
8. Какие аннотации Room используются для CRUD-операций (вставка, обновление, удаление, запрос)?
9. Почему методы DAO, выполняющие операции с базой данных, должны быть suspend-функциями?
10. Как реализуется паттерн Singleton для базы данных Room? Почему это важно?
11. Какие параметры необходимо указать в аннотации @Database при создании базы данных?
12. Как работает стратегия разрешения конфликтов OnConflictStrategy.IGNORE при вставке данных?
13. Почему рекомендуется использовать Flow для работы с данными в Room?
14. Как проверить содержимое базы данных приложения с помощью Database Inspector?
15. Какие ограничения имеет Database Inspector? На каких устройствах он работает?

Содержание отчета

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

ЛАБОРАТОРНАЯ РАБОТА 4

Использование готовых backend-решений Firebase в мобильных приложениях

Цель работы: ознакомиться с этапами создания аутентификации в Android приложениях с использованием готовых backend-решений Firebase.

Теоретическая часть

Разработка надежных и высококачественных приложений для мобильных устройств, предполагает огромную самоотдачу, но что еще более важно, требует мощной и многофункциональной платформы для разработки.

Firebase, предоставляемая компанией Google, является одной из таких платформ, завоевавших прочные позиции среди разработчиков по всему миру.

Firebase предоставляет разработчикам множество возможностей для создания высоко эффективных и универсальных веб-приложений, а также приложений для платформ Android и iOS.

Firebase предоставляет кроссплатформенные SDK для разработки приложений, которые помогают создавать и запускать приложения для платформ Apple, Android, Web, Flutter, Unity и C++.

Некоторые из самых популярных функций платформы Google Firebase включают в себя базы данных, аутентификацию, push-уведомления, аналитику, хранение файлов и многое другое.

Firebase обладает широким спектром дополнительных функций для нужд разработчиков. Главные особенности можно классифицировать на категории *Сборка (Build)*, *Выпуск* и *Мониторинг (Release & Monitor)* и *Вовлечение (Engage)*. Все они используются на разных этапах разработки мобильных приложений.

К функциям сборки относятся:

– *Базы данных:*

- *Cloud Firestore*: облачная NoSQL база данных для хранения и синхронизации данных. Обеспечивает высокую производительность, надежность и автоматическое масштабирование. Данные структурированы в коллекции и документы. Отлично подходит для сложных, иерархических данных.

Пример: чат, где сообщения появляются у всех пользователей мгновенно; многопользовательское приложение с общим состоянием.

- *Realtime Database*: облачная база данных на основе JSON, которая синхронизирует данные в реальном времени между всеми подключенными клиентами, включая работу офлайн. Проще в освоении, чем Firestore, но менее гибкая

для сложных запросов. Идеальна для случаев, когда нужна минимальная задержка (например, в онлайн-играх).

- *Cloud Storage* – сервис для безопасного хранения и обмена пользовательскими файлами (такими как изображения, аудио, видео и другие объекты) непосредственно из приложения. Осуществляет загрузку и скачивание файлов любого размера с поддержкой возобновления при обрыве сети. Интегрируется с Firebase Authentication для управления доступом.

- *Аутентификация*: предоставляет готовые библиотеки и бэкенд для аутентификации пользователей с помощью паролей, телефонных номеров и провайдеров (Google, Facebook и др.).

- *Firebase Authentication* – готовый бэкенд для аутентификации пользователей с поддержкой самых популярных методов входа: по email/паролю, через Google, Facebook, Twitter, GitHub, а также по номеру телефона. Избавляет от необходимости самостоятельно реализовывать сложную и опасную логику хранения паролей.

- *App Check* – это защита бэкенда от неавторизованного доступа. Гарантирует, что запросы к вашим сервисам Firebase (базам данных, хранилищу) поступают только от вашего настоящего приложения, защищая от злоумышленников и ботов.

- *Firebase Security Rules* – язык правил для определения прав доступа к данным и файлам прямо на стороне сервера. Вы можете прописать, кто (какой пользователь) и при каких условиях может читать, писать или удалять данные в Firestore/Realtime Database и файлы в Cloud Storage. Не требует написания серверного кода.

- *Машинное обучение (Firebase ML)*: мобильный SDK, который позволяет легко интегрировать возможности машинного обучения Google в приложения для iOS и Android, без необходимости углубленных знаний.

- *Облачные функции (Cloud Functions)*: бессерверная архитектура для выполнения бэкенд-кода в ответ на события от Firebase или HTTPS-запросы.

- *Хостинг*: предоставляет масштабируемый хостинг для веб-приложений и статического контента с поддержкой SSL.

Выпуск и мониторинг – это набор инструментов для обеспечения качественного запуска приложений:

- *Crashlytics*: сервис для отслеживания, приоритизации и устранения сбоев в реальном времени. В реальном времени получаете детальные отчеты о каждом падении приложения: стектрейс, устройство, версия ОС и т.д. Помогает быстро находить, исправлять и приоритизировать баги.

- *Google Analytics*: инструмент для анализа поведения пользователей и принятия решений на основе данных. Отслеживание поведения пользователей (экраны, события, конверсии), аудитории (геолокация, устройства) и т.д. Позволяет понять, как пользователи взаимодействуют с вашим приложением.

- *Performance Monitoring*: помогает получать информацию о производительности приложения и находить узкие места. Автоматически собирает метрики времени запуска приложения, скорости работы сети, производительности экранов. Позволяет находить «узкие места» в работе приложения.

- *Test Lab*: облачная инфраструктура для тестирования приложений на различных устройствах и конфигурациях.

- *App Distribution*: сервис для распространения бета-версий приложений среди тестировщиков.

Инструменты вовлечения (Engage) – это набор функций для взаимодействия с пользователями и их удержания:

- *Удаленная настройка (Remote Config)*: позволяет изменять поведение и внешний вид приложения без публикации обновления. Можно менять цвета, тексты, включать/выключать функции для разных групп пользователей (например, для A/B тестирования).

- *A/B-тестирование*: дает возможность тестировать изменения в функциях и интерфейсе на части пользователей.

- *Прогнозирования (Predictions)*: использует машинное обучение для прогнозирования поведения пользователей и автоматизации действий.

- *Динамические ссылки (Dynamic Links)*: «умные» URL-адреса, которые перенаправляют пользователей в нужные разделы приложения. Если приложение установлено – ссылка откроет его на нужном экране; не установлено – пользователь попадет в магазин для установки, а после запуска откроет нужный контент.

- *Внутриприложенные сообщения (In-App Messaging)*: позволяет отправлять целевые сообщения активным пользователям прямо в приложении. Идеально для вовлечения пользователей, объявления о новых функциях или проведения опросов.

- *Сервис push-уведомлений (Cloud Messaging (FCM))*: позволяет отправлять уведомления на все устройства с вашим приложением. Можно отправлять как уведомления, так и данные для обработки внутри приложения. Интегрируется с Analytics для таргетированной рассылки.

I. Подготовка проекта.

Для начала работы с Firebase необходимо:

- установить или обновить Android Studio до последней версии;
- создать новый проект (Empty Activity, Kotlin);
- убедиться, что ваш проект соответствует этим требованиям. Обратите внимание, что к некоторым продуктам могут предъявляться более строгие требования:
 - целевой уровень API 21 (Lollipop) или выше;
 - использует Android 5.0 или выше;
 - использует Jetpack (AndroidX), который включает в себя соответствие следующим требованиям к версии:
 - com.android.tools.build:gradle v7.3.0 или более поздняя версия;
 - compileSdkVersion 28 или более поздняя версия;
- настроить физическое устройство или использовать эмулятор для запуска вашего приложения. Обратите внимание, что для Firebase SDK, зависящих от сервисов Google Play, на устройстве или эмуляторе должны быть установлены сервисы Google Play;
- войте в Firebase, используя свою учетную запись Google (рисунок 4.1).

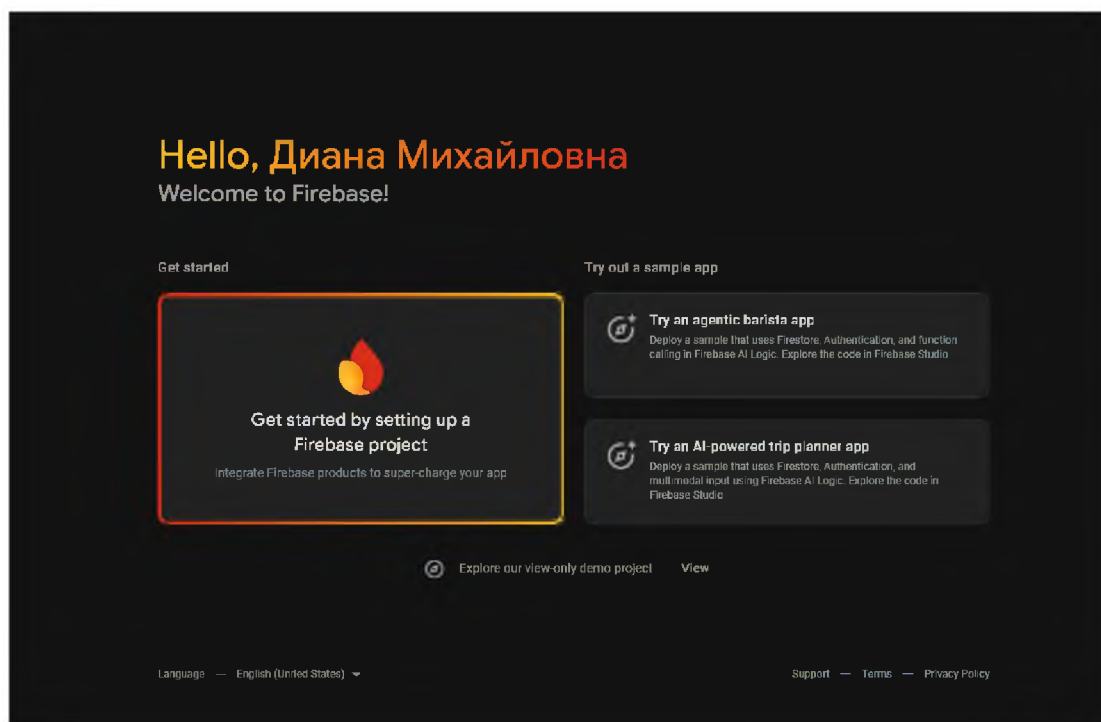


Рисунок 4.1. – Стартовая страница Firebase

Подключить Android-приложение к Firebase можно используя рабочий процесс настройки консоли Firebase.

II. Добавление FIREBASE с помощью консоли FIREBASE.

Добавление Firebase в ваше приложение включает в себя задачи как в консоли [Firebase](#), так и в вашем открытом проекте Android (например, вы загружаете файлы конфигурации Firebase из консоли, а затем перемещаете их в свой проект Android).

Шаг 1: Зарегистрируйте свое приложение в Firebase.

Чтобы использовать Firebase в вашем приложении для Android, вам необходимо зарегистрировать его в проекте Firebase (рисунок 4.2).

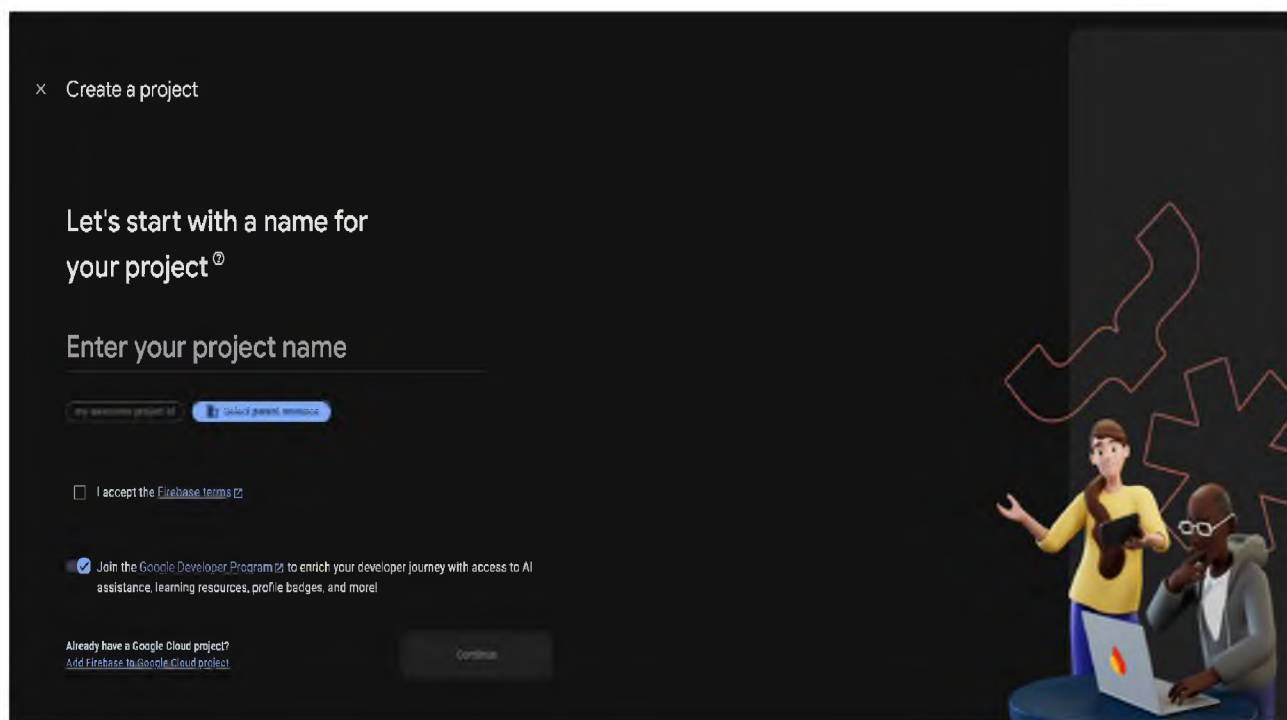


Рисунок 4.2. — Создание проекта в Firebase

Регистрация приложения часто называется «добавлением» приложения в проект:

1. Перейдите в консоль Firebase (рисунок 4.3).
2. В центре страницы обзора проекта нажмите значок Android () или «Добавить приложение», чтобы запустить рабочий процесс настройки (рисунок 4.4).
3. Введите имя пакета вашего приложения в поле «Имя пакета Android» (рисунок 4.5).

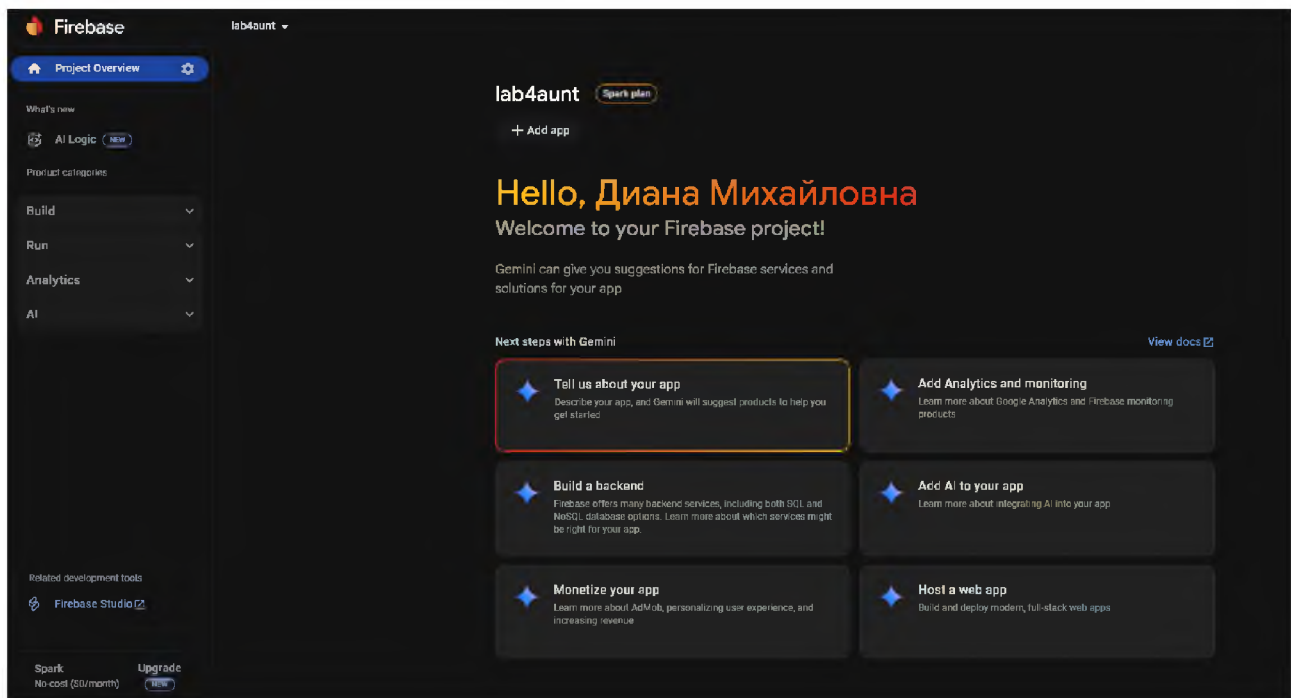


Рисунок 4.3. – Добавление Firebase в приложение

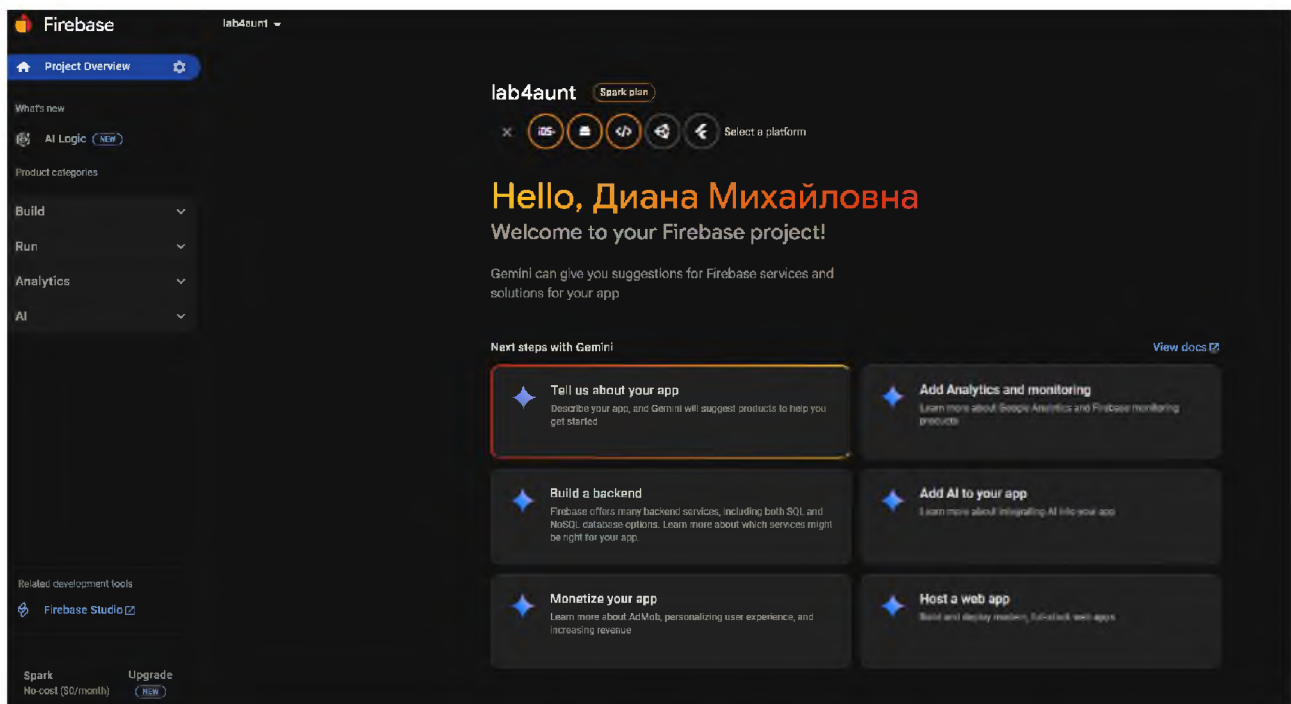


Рисунок 4.4. – Запуск рабочего процесса настройки для приложения Android

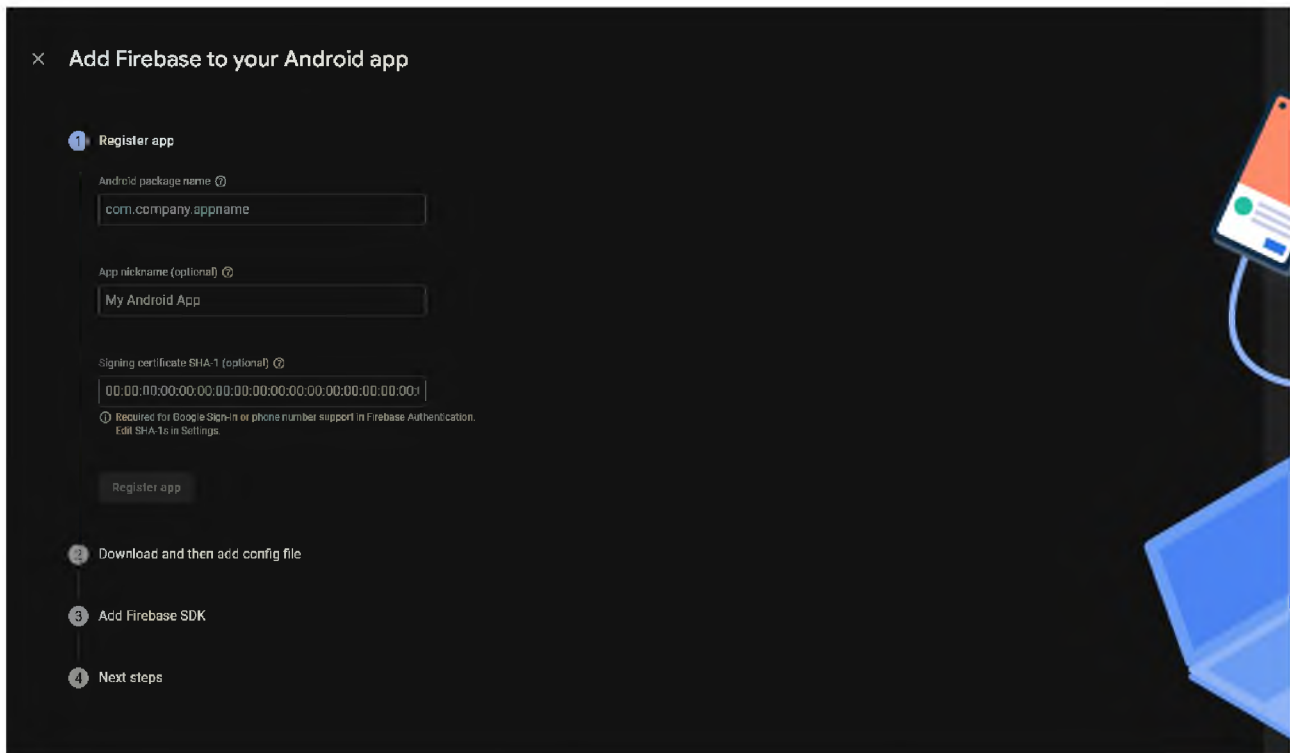


Рисунок 4.5. – Регистрация приложения в Firebase

Что такое имя пакета и где его можно найти?

– Имя пакета уникально идентифицирует ваше приложение на устройстве и в магазине Google Play.

– Имя пакета часто называют *идентификатором приложения*.

– Найдите имя пакета вашего приложения в файле Gradle вашего модуля (уровня приложения), обычно это app/build.gradle (пример имени пакета: com.yourcompany.yourproject) (рисунок 4.6).

– Имейте в виду, что значение имени пакета чувствительно к регистру и его нельзя изменить для этого приложения Firebase Android после его регистрации в вашем проекте Firebase.

4. Нажмите «Зарегистрировать приложение».

Шаг 2: Добавьте файл конфигурации Firebase.

1. Загрузите и добавьте файл конфигурации Firebase вашего приложения (google-services.json) в вашу кодовую базу (рисунок 4.7):

– нажмите «Загрузить google-services.json», чтобы получить файл конфигурации Firebase вашего приложения;

– переместите файл конфигурации в корневой каталог модуля (уровня приложения) вашего приложения.

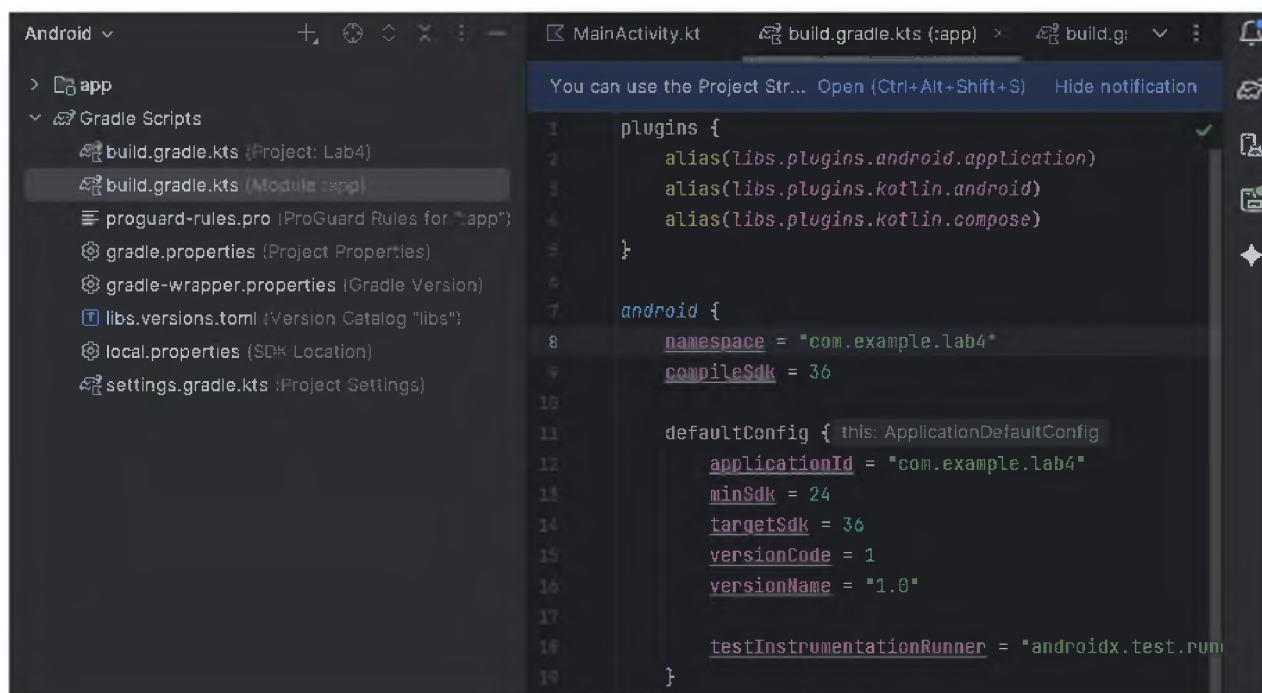


Рисунок 4.6. – Имя пакета приложения

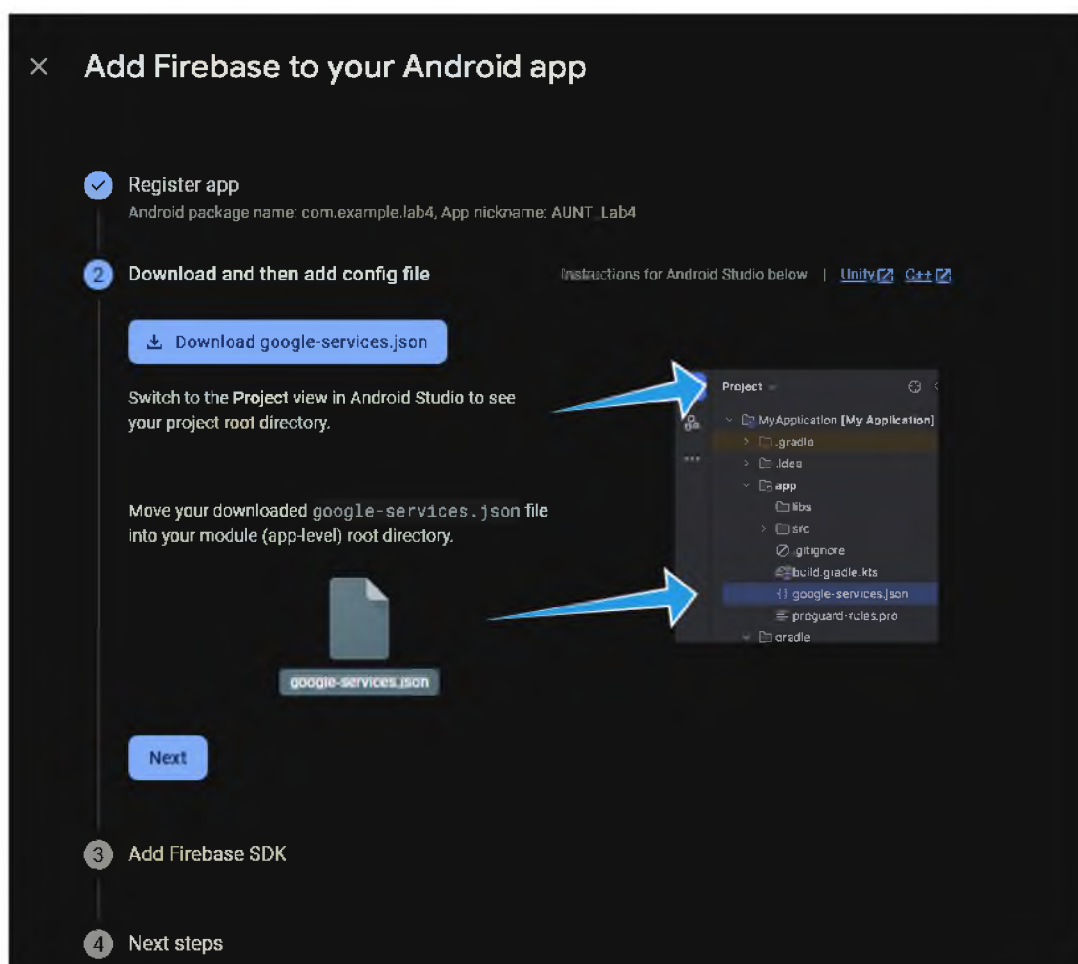


Рисунок 4.7. – скачивание файла конфигурации Firebase

Что нужно знать об файле конфигурации:

- Файл конфигурации Firebase содержит уникальные, но не секретные идентификаторы вашего проекта и приложения.
- Снова загрузить файл конфигурации Firebase можно в любое время.
- К имени файла конфигурации не должны быть добавлены дополнительные символы.

2. Чтобы сделать значения в файле конфигурации google-services.json доступными для Firebase SDK, понадобится плагин Gradle для сервисов Google (google-services):

- в файле Gradle корневого уровня (уровня проекта) (<project>/build.gradle.kts или <project>/build.gradle) добавьте плагин служб Google в качестве зависимости:

```
plugins {  
    id("com.android.application") version "7.3.0" apply false  
    // ...  
  
    // Add the dependency for the Google services Gradle plugin  
    id("com.google.gms.google-services") version "4.4.3" apply false  
}
```

- в файле Gradle вашего модуля (уровня приложения) (обычно <project>/<app-module>/build.gradle.kts или <project>/<app-module>/build.gradle) добавьте плагин служб Google:

```
plugins {  
    id("com.android.application")  
  
    // Add the Google services Gradle plugin  
    id("com.google.gms.google-services")  
    // ...  
}
```

Шаг 3: Добавьте Firebase SDK в свое приложение.

1. В файле Gradle вашего модуля (уровня приложения) (обычно <project>/<app-module>/build.gradle.kts или <project>/<app-module>/build.gradle) добавьте зависимости для продуктов Firebase, которые вы хотите использовать в своем приложении. Мы рекомендуем использовать Firebase Android BoM для управления версиями библиотеки:

```
// ===== FIREBASE =====
implementation(platform("com.google.firebase:firebase-bom:32.7.2")) // ОБНОВЛЕННАЯ
ВЕРСИЯ

// Firebase Authentication
implementation("com.google.firebase:firebase-auth-ktx") // КТХ ВЕРСИЯ

// Firebase Firestore (если нужно хранить данные пользователей)
implementation("com.google.firebase:firebase-firestore-ktx")
```

Задействуя Firebase Android BoM, приложение всегда будет использовать совместимые версии библиотек Firebase Android.

2. После добавления зависимостей для продуктов, которые вы хотите использовать, синхронизируйте свой проект Android с файлами Gradle.

Сборки Gradle с плагином Android Gradle (AGP) версии 4.2 или более ранней должны включать поддержку Java 8. В противном случае при добавлении Firebase SDK в этих проектах Android возникнет ошибка сборки.

Чтобы исправить эту ошибку сборки, вы можете:

- добавить указанные compileOptions из сообщения об ошибке в файл build.gradle.kts или build.gradle уровня приложения или
- увеличить minSdk для вашего Android-проекта до 26 или выше.

III. Работы с аутентификацией FIREBASE на ANDROID.

Для добавления Firebase Authentication в приложение необходимо перейти в файл Gradle (уровня приложения) и обновить зависимости для библиотек, включая Firebase Authentication для Android:

```
dependencies {
// ===== ANDROID X CORE =====
implementation(libs.androidx.core.ktx)
implementation(libs.androidx.lifecycle.runtime.ktx)
implementation(libs.androidx.activity.compose)

// ===== COMPOSE =====
implementation(platform(libs.androidx.compose.bom))
implementation(libs.androidx.compose.ui)
implementation(libs.androidx.compose.ui.graphics)
implementation(libs.androidx.compose.ui.tooling.preview)
implementation(libs.androidx.compose.material3)

// ===== COMPOSE ДОПОЛНИТЕЛЬНО =====
implementation("androidx.navigation:navigation-compose:2.7.5")
```

```

implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")

// ===== FIREBASE =====
implementation(platform("com.google.firebase:firebase-bom:32.7.2")) // ОБНОВЛЕННАЯ
ВЕРСИЯ

// Firebase Authentication
implementation("com.google.firebase:firebase-auth-ktx") // KTX ВЕРСИЯ

// Firebase Firestore (если нужно хранить данные пользователей)
implementation("com.google.firebase:firebase-firestore-ktx")

// ===== COROUTINES =====
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.7.3")

// ===== TESTING =====
testImplementation(libs.junit)
androidTestImplementation(libs.androidx.junit)
androidTestImplementation(libs.androidx.espresso.core)
androidTestImplementation(platform(libs.androidx.compose.bom))
androidTestImplementation(libs.androidx.compose.ui.test.junit4)
debugImplementation(libs.androidx.compose.ui.tooling)
debugImplementation(libs.androidx.compose.ui.test.manifest)
}

```

1. В новом файле создадим класс AuthViewModel для ViewModel:

```

package com.example.lab4.auth

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.google.firebase.auth.FirebaseAuth
import com.google.firebase.auth.FirebaseUser
import com.google.firebase.auth.ktx.auth
import com.google.firebase.ktx.Firebase
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch
import kotlinx.coroutines.tasks.await

// VIEWMODEL ДЛЯ УПРАВЛЕНИЯ АУТЕНТИФИКАЦИЕЙ =====

/**
 * AuthViewModel - ViewModel для управления состоянием аутентификации и пользова-
 * тельскими данными
 * Наследуется от ViewModel для сохранения состояния при изменениях конфигурации

```

```

*/
class AuthViewModel : ViewModel() {

    // FIREBASE AUTHENTICATION =====

    /**
     * Firebase Auth instance - основной объект для работы с аутентификацией Firebase
     * Инициализируется один раз и используется throughout ViewModel
     */
    private val auth: FirebaseAuth = Firebase.auth

    // СОСТОЯНИЯ UI =====

    /**
     * MutableStateFlow для состояния UI аутентификации
     * private - изменяется только внутри ViewModel
     * _uiState - соглашение об именовании для mutable StateFlow
     */
    private val _uiState = MutableStateFlow<AuthUiState>(AuthUiState.Loading)

    /**
     * Public StateFlow для наблюдения за состоянием UI из Compose
     * asStateFlow() - преобразует mutable в immutable StateFlow
     */
    val uiState: StateFlow<AuthUiState> = _uiState.asStateFlow()

    /**
     * MutableStateFlow для текущего пользователя
     * Может быть null если пользователь не авторизован
     */
    private val _currentUser = MutableStateFlow<FirebaseUser?>(null)

    /**
     * Public StateFlow для наблюдения за текущим пользователем из Compose
     */
    val currentUser: StateFlow<FirebaseUser?> = _currentUser.asStateFlow()

    // КОНСТРУКТОР И ИНИЦИАЛИЗАЦИЯ =====

    /**
     * init блок - выполняется при создании ViewModel
     * Проверяет текущего пользователя при инициализации ViewModel
     */
    init {
        // Проверка текущего пользователя при инициализации
        checkCurrentUser()
    }
}

```

```

// ПРОВЕРКА ТЕКУЩЕГО ПОЛЬЗОВАТЕЛЯ =====

/**
 * Проверяет, авторизован ли пользователь в Firebase Auth
 * Вызывается при инициализации ViewModel
 */
private fun checkCurrentUser() {
    // Получение текущего пользователя из Firebase Auth
    val user = auth.currentUser

    // Обновление StateFlow с текущим пользователем
    _currentUser.value = user

    // Обновление состояния UI в зависимости от наличия пользователя
    if (user != null) {
        // Если пользователь существует - состояние Успех
        _uiState.value = AuthUiState.Success(user)
    } else {
        // Если пользователя нет - состояние Не авторизован
        _uiState.value = AuthUiState.NotSignedIn
    }
}

// РЕГИСТРАЦИЯ НОВОГО ПОЛЬЗОВАТЕЛЯ =====

/**
 * Регистрирует нового пользователя с email и паролем
 *
 * @param email Email пользователя для регистрации
 * @param password Пароль пользователя (минимум 6 символов)
 */
fun signUp(email: String, password: String) {
    // Запуск корутины в viewModelScope (автоматически отменяется при очистке View-
    Model)
    viewModelScope.launch {
        // Установка состояния "Загрузка" перед началом операции
        _uiState.value = AuthUiState.Loading

        try {
            // ВЫПОЛНЕНИЕ РЕГИСТРАЦИИ В FIREBASE =====

            // await() - приостанавливает корутину до завершения операции
            // createUserWithEmailAndPassword - метод Firebase для создания пользователя
            val result = auth.createUserWithEmailAndPassword(email, password).await()

            // УСПЕШНАЯ РЕГИСТРАЦИЯ =====

```

```

        // Обновление текущего пользователя
        _currentUser.value = result.user

        // Установка состояния "Успех" с данными пользователя
        // !! - уверенность что user не null при успешной регистрации
        _uiState.value = AuthUiState.Success(result.user!!)

    } catch (e: Exception) {
        // ОШИБКА РЕГИСТРАЦИИ =====

        // Установка состояния "Ошибка" с сообщением
        // e.message ?: "Registration failed" - сообщение об ошибке или дефолтное
        _uiState.value = AuthUiState.Error(e.message ?: "Registration failed")
    }
}

// ВХОД ПОЛЬЗОВАТЕЛЯ =====

/**
 * Выполняет вход существующего пользователя с email и паролем
 *
 * @param email Email пользователя для входа
 * @param password Пароль пользователя
 */
fun signIn(email: String, password: String) {
    // Запуск корутины в viewModelScope
    viewModelScope.launch {
        // Установка состояния "Загрузка" перед началом операции
        _uiState.value = AuthUiState.Loading

        try {
            // ВЫПОЛНЕНИЕ ВХОДА В FIREBASE =====

            // signInWithEmailAndPassword - метод Firebase для входа пользователя
            val result = auth.signInWithEmailAndPassword(email, password).await()

            // УСПЕШНЫЙ ВХОД =====

            // Обновление текущего пользователя
            _currentUser.value = result.user

            // Установка состояния "Успех" с данными пользователя
            _uiState.value = AuthUiState.Success(result.user!!)

        } catch (e: Exception) {

```

```

// ОШИБКА ВХОДА =====

// Установка состояния "Ошибка" с сообщением
_uiState.value = AuthUiState.Error(e.message ?: "Login failed")
    }
}

// ВЫХОД ИЗ СИСТЕМЫ =====

/**
 * Выполняет выход текущего пользователя из системы
 * Очищает состояние пользователя и обновляет UI состояние
 */
fun signOut() {
    // ВЫЗОВ FIREBASE SIGNOUT =====

    // signOut() - метод Firebase для выхода пользователя
    auth.signOut()

    // ОЧИСТКА СОСТОЯНИЯ В VIEWMODEL =====

    // Очистка текущего пользователя
    _currentUser.value = null

    // Установка состояния "Не авторизован"
    _uiState.value = AuthUiState.NotSignedIn
}

// СОСТОЯНИЯ UI ДЛЯ АУТЕНТИФИКАЦИИ =====

/**
 * Sealed class, представляющий различные состояния UI аутентификации
 * Sealed class позволяет создавать ограниченную иерархию классов
 */
sealed class AuthUiState {

    /**
     * Object - состояние загрузки (не требует дополнительных данных)
     * Используется когда выполняется асинхронная операция
     */
    object Loading : AuthUiState()

    /**
     * Object - состояние "не авторизован" (пользователь не вошел)
     * Используется когда пользователь не авторизован

```

```

*/
object NotSignedIn : AuthUiState()

/**
 * Data class - состояние успешной аутентификации
 * Содержит данные авторизованного пользователя
 *
 * @param user Объект FirebaseUser с данными авторизованного пользователя
 */
data class Success(val user: FirebaseUser) : AuthUiState()

/**
 * Data class - состояние ошибки аутентификации
 * Содержит сообщение об ошибке для показа пользователю
 *
 * @param message Текст сообщения об ошибке
 */
data class Error(val message: String) : AuthUiState()
}

```

2. Создадим Composable экраны:

```

package com.example.lab4.ui.screens

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material3.Button
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Scaffold
import androidx.compose.material3.SnackbarHost
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.material3.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue

```

```

import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.text.input.PasswordVisualTransformation
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.lab4.auth.AuthViewModel
import kotlinx.coroutines.flow.collectLatest

// Аннотация для использования экспериментальных функций Material3
@OptIn(ExperimentalMaterial3Api::class)
// Аннотация @Composable указывает, что это функция для построения UI
@Composable
// ОПРЕДЕЛЕНИЕ КОМПОЗАБЛА LOGINSCREEN =====
fun LoginScreen(
    // Функция обратного вызова для навигации на главный экран
    onNavigateToHome: () -> Unit,
    // ViewModel для управления состоянием аутентификации, по умолчанию получается че-
    рез viewModel()
    viewModel: AuthViewModel = viewModel()
){
    // СОСТОЯНИЯ ФОРМЫ =====

    // Состояние для хранения email пользователя
    // remember - сохраняет значение при перекомпозиции
    // mutableStateOf - делает значение наблюдаемым для Compose
    var email by remember { mutableStateOf("") }

    // Состояние для хранения пароля пользователя
    var password by remember { mutableStateOf("") }

    // Состояние загрузки (показывается индикатор прогресса)
    var isLoading by remember { mutableStateOf(false) }

    // SNACKBAR ДЛЯ ПОКАЗА СООБЩЕНИЙ =====

    // SnackbarHostState для управления показом уведомлений
    val snackbarHostState = remember { SnackbarHostState() }

    // НАБЛЮДЕНИЕ ЗА СОСТОЯНИЕМ АУТЕНТИФИКАЦИИ =====

    // LaunchedEffect - запускает блок кода при изменении ключа (viewModel.uiState)
    LaunchedEffect(key1 = viewModel.uiState) {
        // collectLatest - собирает последние значения из Flow
        viewModel.uiState.collectLatest { state ->
            // Обработка различных состояний аутентификации
            when (state) {

```

```

// Если состояние "Загрузка" - показываем индикатор
is AuthUiState.Loading -> isLoading = true

// Если состояние "Успех" - скрываем индикатор и переходим на главный экран
is AuthUiState.Success -> {
    isLoading = false
    onNavigateToHome() // Вызов функции навигации
}

// Если состояние "Ошибка" - скрываем индикатор и показываем сообщение
is AuthUiState.Error -> {
    isLoading = false
    // Показываем сообщение об ошибке в Snackbar
    snackbarHostState.showSnackbar(state.message)
}

// Для всех остальных состояний скрываем индикатор загрузки
else -> isLoading = false
}
}

// ОСНОВНОЙ UI С ИСПОЛЬЗОВАНИЕМ SCAFFOLD =====

// Scaffold - базовый layout для Material Design приложения
Scaffold(
    // Настройка Snackbar для показа уведомлений
    snackbarHost = { SnackbarHost(snackbarHostState) }
) { paddingValues -> // paddingValues - отступы от системных UI элементов
    // COLUMN - вертикальная компоновка элементов
    Column(
        // МОДИФИКАТОРЫ ДЛЯ НАСТРОЙКИ ВНЕШНЕГО ВИДА =====
        modifier = Modifier
            .fillMaxSize() // Занимает всю доступную площадь
            .padding(paddingValues) // Отступы от Scaffold
            .padding(24.dp), // Дополнительные внутренние отступы

        // НАСТРОЙКА ВЫРАВНИВАНИЯ ЭЛЕМЕНТОВ =====
        verticalArrangement = Arrangement.Center, // Вертикальное центрирование
        horizontalAlignment = Alignment.CenterHorizontally // Горизонтальное центрирование
    ) {
        // ЗАГОЛОВОК ЭКРАНА =====
        Text(
            text = "Firebase Auth", // Текст заголовка
            style = MaterialTheme.typography.headlineMedium, // Стиль из темы
            color = MaterialTheme.colorScheme.primary // Цвет из цветовой схемы
        )
    }
}

```

```

)

// РАЗДЕЛИТЕЛЬ МЕЖДУ ЭЛЕМЕНТАМИ =====
Spacer(modifier = Modifier.height(32.dp))

//  ПОЛЕ ДЛЯ ВВОДА EMAIL =====
OutlinedTextField(
    value = email, // Текущее значение из состояния
    onChange = { email = it }, // Обновление состояния при изменении текста
    label = { Text("Email") }, // Подпись поля
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email), //
Настройка клавиатуры
    modifier = Modifier.fillMaxWidth(), // Занимает всю ширину
    singleLine = true // Однострочное поле
)

// РАЗДЕЛИТЕЛЬ МЕЖДУ ПОЛЯМИ ВВОДА =====
Spacer(modifier = Modifier.height(16.dp))

// ПОЛЕ ДЛЯ ВВОДА ПАРОЛЯ =====
OutlinedTextField(
    value = password, // Текущее значение пароля
    onChange = { password = it }, // Обновление состояния пароля
    label = { Text("Password") }, // Подпись поля
    visualTransformation = PasswordVisualTransformation(), // Скрытие символов па-
роля
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password), //
Клавиатура для пароля
    modifier = Modifier.fillMaxWidth(), // Занимает всю ширину
    singleLine = true // Однострочное поле
)

// РАЗДЕЛИТЕЛЬ ПЕРЕД КНОПКАМИ =====
Spacer(modifier = Modifier.height(24.dp))

// КНОПКА ВХОДА В СИСТЕМУ =====
Button(
    // ОБРАБОТЧИК НАЖАТИЯ КНОПКИ
    onClick = {
        // Проверка что оба поля заполнены
        if (email.isNotEmpty() && password.isNotEmpty()) {
            // Вызов метода входа из ViewModel
            viewModel.signIn(email, password)
        }
    },
    // МОДИФИКАТОРЫ КНОПКИ
    modifier = Modifier.fillMaxWidth(), // Кнопка на всю ширину

```

```

        enabled = !isLoading // Блокировка кнопки во время загрузки
    ) {
        // УСЛОВНЫЙ РЕНДЕРИНГ СОДЕРЖИМОГО КНОПКИ
        if (isLoading) {
            // Если идет загрузка - показываем индикатор
            CircularProgressIndicator(
                modifier = Modifier.height(16.dp) // Настройка размера индикатора
            )
        } else {
            // Если загрузки нет - показываем текст
            Text("Sign In")
        }
    }
}

// РАЗДЕЛИТЕЛЬ МЕЖДУ КНОПКАМИ =====
Spacer(modifier = Modifier.height(8.dp))

// КНОПКА РЕГИСТРАЦИИ НОВОГО ПОЛЬЗОВАТЕЛЯ =====
Button(
    // ОБРАБОТЧИК НАЖАТИЯ КНОПКИ
    onClick = {
        // Проверка что оба поля заполнены
        if (email.isNotEmpty() && password.isNotEmpty()) {
            // Вызов метода регистрации из ViewModel
            viewModel.signUp(email, password)
        }
    },
    // МОДИФИКАТОРЫ КНОПКИ
    modifier = Modifier.fillMaxWidth(), // Кнопка на всю ширину
    enabled = !isLoading // Блокировка кнопки во время загрузки
) {
    // УСЛОВНЫЙ РЕНДЕРИНГ СОДЕРЖИМОГО КНОПКИ
    if (isLoading) {
        // Если идет загрузка - показываем индикатор
        CircularProgressIndicator(
            modifier = Modifier.height(16.dp) // Настройка размера индикатора
        )
    } else {
        // Если загрузки нет - показываем текст
        Text("Sign Up")
    }
}
}
}
}
}

```

Итоговый вид экрана авторизации и регистрации представлен на рисунках 4.8 и 4.9.

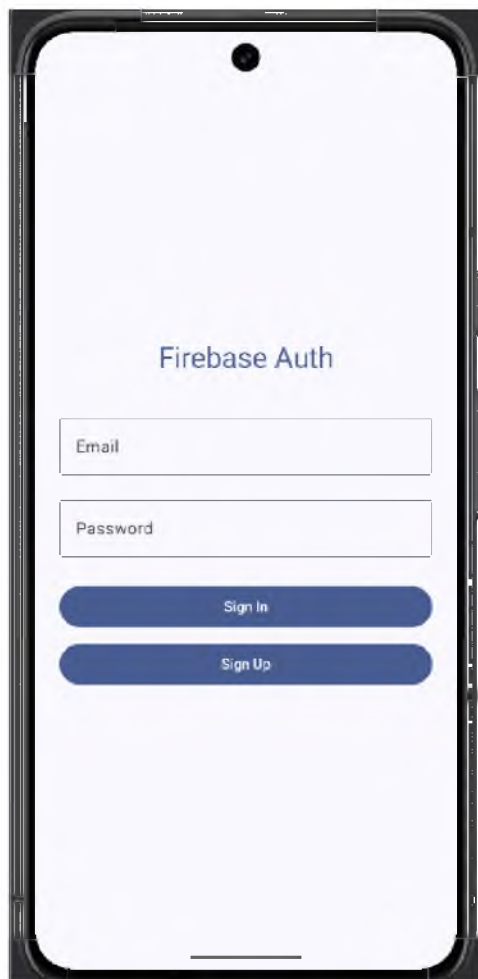


Рисунок 4.8. – Экран регистрации и авторизации

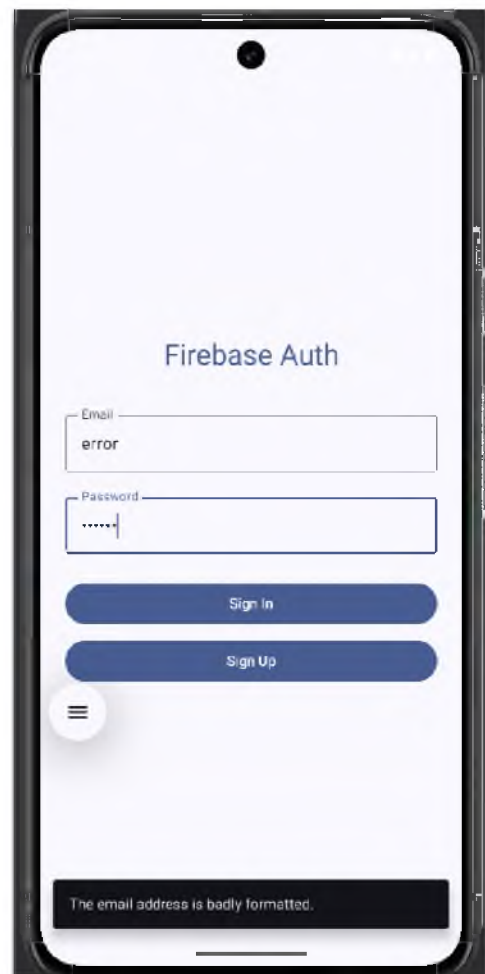


Рисунок 4.9. – Сообщение об ошибке авторизации/регистрации

3. Создадим экран для авторизованных пользователей:

```
package com.example.lab4.ui.screens

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
```

```

import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.lab4.auth.AuthViewModel
import com.google.firebase.auth.FirebaseAuth

// ГЛАВНЫЙ ЭКРАН ПОСЛЕ УСПЕШНОЙ АУТЕНТИФИКАЦИИ =====

/**
 * HomeScreen - главный экран приложения, показываемый после успешного входа
 *
 * @param onSignOut Функция обратного вызова для выхода из системы и навигации
 * @param viewModel ViewModel для управления состоянием аутентификации и пользова-
 * тельскими данными
 */
@Composable
fun HomeScreen(
    // Функция обратного вызова, вызываемая при выходе пользователя
    onSignOut: () -> Unit,
    // ViewModel получается автоматически через viewModel()
    viewModel: AuthViewModel = viewModel()
) {
    // ПОЛУЧЕНИЕ ТЕКУЩЕГО ПОЛЬЗОВАТЕЛЯ =====

    // Получение текущего пользователя из ViewModel
    // .value - доступ к значению StateFlow
    val currentUser = viewModel.currentUser.value

    // ОСНОВНОЙ КОНТЕЙНЕР ЭКРАНА =====

    // Column - вертикальное расположение элементов
    Column(
        // МОДИФИКАТОРЫ ДЛЯ НАСТРОЙКИ ВНЕШНЕГО ВИДА =====
        modifier = Modifier
            .fillMaxSize() // Занимает всю доступную площадь
            .padding(24.dp), // Внутренние отступы по краям

        // 🎯 НАСТРОЙКА ВЫРАВНИВАНИЯ ЭЛЕМЕНТОВ =====
        verticalArrangement = Arrangement.Center, // Вертикальное центрирование
        horizontalAlignment = Alignment.CenterHorizontally // Горизонтальное центрирование
    ) {
        // КАРТОЧКА С ИНФОРМАЦИЕЙ О ПОЛЬЗОВАТЕЛЕ =====

        // UserInfoCard - переиспользуемый компонент для отображения данных пользова-
        теля
    }
}

```

```

UserInfoCard(user = currentUser)

// РАЗДЕЛИТЕЛЬ МЕЖДУ КАРТОЧКОЙ И КНОПКОЙ =====
Spacer(modifier = Modifier.height(32.dp))

// КНОПКА ВЫХОДА ИЗ СИСТЕМЫ =====
Button(
    // ОБРАБОТЧИК НАЖАТИЯ КНОПКИ =====
    onClick = {
        // 1. Вызов метода выхода из ViewModel (очистка состояния)
        viewModel.signOut()
        // 2. Вызов функции обратного вызова для навигации на экран логина
        onSignOut()
    }
) {
    // ТЕКСТ НА КНОПКЕ =====
    Text("Sign Out")
}
}

// КОМПОНЕНТ КАРТОЧКИ С ИНФОРМАЦИЕЙ О ПОЛЬЗОВАТЕЛЕ =====

/**
 * UserInfoCard - переиспользуемый компонент для отображения информации о пользова-
 * теле
 *
 * @param user Объект FirebaseUser с данными текущего пользователя (может быть null)
 */
@Composable
fun UserInfoCard(user: FirebaseUser?) {
    // CARD - MATERIAL DESIGN КАРТОЧКА =====

    // Card - компонент Material Design для группировки связанной информации
    Card(
        // Модификатор для растягивания карточки на всю доступную ширину
        modifier = Modifier.fillMaxWidth()
    ) {
        // КОЛОНКА С ИНФОРМАЦИЕЙ ВНУТРИ КАРТОЧКИ =====
        Column(
            // Внутренние отступы внутри карточки
            modifier = Modifier.padding(16.dp)
        ) {
            // ПРИВЕТСТВЕННЫЙ ЗАГОЛОВОК =====
            Text(
                text = "Welcome!", // Текст приветствия
                style = MaterialTheme.typography.headlineSmall, // Стиль маленького заголовка
            )
        }
    }
}

```

```

        color = MaterialTheme.colorScheme.primary // Цвет из основной палитры темы
    )

    // РАЗДЕЛИТЕЛЬ МЕЖДУ ЗАГОЛОВКОМ И ДАННЫМИ =====
    Spacer(modifier = Modifier.height(8.dp))

    // EMAIL ПОЛЬЗОВАТЕЛЯ =====
    Text(
        // Отображение email или "Unknown" если пользователь null
        text = "Email: ${user?.email ?: "Unknown"}",
        style = MaterialTheme.typography.bodyMedium // Стил ь основного текста
    )

    // УНИКАЛЬНЫЙ ID ПОЛЬЗОВАТЕЛЯ (СОКРАЩЕННЫЙ) =====
    Text(
        // Отображение первых 8 символов UID или "Unknown"
        // take(8) - берет только первые 8 символов для безопасности
        text = "UID: ${user?.uid?.take(8) ?: "Unknown"}...",
        style = MaterialTheme.typography.bodySmall, // Стил ь мелкого текста
        color = MaterialTheme.colorScheme.onSurfaceVariant // Вторичный цвет текста
    )

    // СТАТУС ПОДТВЕРЖДЕНИЯ EMAIL =====
    Text(
        // Отображение статуса подтверждения email
        text = "Email verified: ${user?.isEmailVerified ?: false}",
        style = MaterialTheme.typography.bodySmall, // Стил ь мелкого текста
        color = MaterialTheme.colorScheme.onSurfaceVariant // Вторичный цвет текста
    )
}
}
}

```

Итоговый вид главного экрана после успешной авторизации представлен на рисунке 4.10.

4. Настроим навигацию:

```

package com.example.lab4.ui.navigation

import androidx.compose.runtime.Composable
import androidx.navigation.NavHostController
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
import com.example.lab4.ui.screens.HomeScreen
import com.example.lab4.ui.screens.LoginScreen

```

```

// НАВИГАЦИОННЫЙ ГРАФ ДЛЯ АУТЕНТИФИКАЦИИ =====

/**
 * AuthNavGraph - основной навигационный компонент приложения
 * Управляет переходами между экранами аутентификации и главным экраном
 *
 * @param navController Контроллер навигации для управления переходами между экра-
нами
 * По умолчанию создается новый через rememberNavController()
 */
@Composable
fun AuthNavGraph(
    navController: NavHostController = rememberNavController()
) {
    // СОЗДАНИЕ НАВИГАЦИОННОГО ХОСТА =====

    // NavHost - контейнер, который управляет отображением различных экранов
    NavHost(
        // Контроллер навигации, который управляет стеком экранов и переходами
        navController = navController,

        // ОПРЕДЕЛЕНИЕ НАЧАЛЬНОГО ЭКРАНА =====

        // startDestination - экран, который показывается при запуске приложения
        // Условно выбирается на основе статуса авторизации пользователя
        startDestination = if (isUserLoggedIn()) "home" else "login"
    ) {
        // ОПРЕДЕЛЕНИЕ ЭКРАНА ЛОГИНА =====

        // composable - определяет экран, который можно использовать в навигации
        composable("login") {
            // LoginScreen - экран входа и регистрации пользователя
            LoginScreen(
                // ОБРАБОТЧИК УСПЕШНОГО ВХОДА =====

                // onNavigateToHome - вызывается при успешной аутентификации
                onNavigateToHome = {
                    // Навигация на главный экран с очисткой стека
                    navController.navigate("home") {
                        // ОЧИСТКА СТЕКА НАВИГАЦИИ =====

                        // popUpTo("login") - удаляет все экраны до "login" из стека
                        // inclusive = true - удаляет и сам экран "login" из стека
                        // Это предотвращает возврат на экран логина кнопкой "Назад"
                        popUpTo("login") { inclusive = true }
                    }
                }
            )
        }
    }
}

```

```

    }
  )
}

// ОПРЕДЕЛЕНИЕ ГЛАВНОГО ЭКРАНА =====

composable("home") {
  // HomeScreen - главный экран приложения после успешного входа
  HomeScreen(
    // ОБРАБОТЧИК ВЫХОДА ИЗ СИСТЕМЫ =====

    // onSignOut - вызывается при выходе пользователя из системы
    onSignOut = {
      // Навигация на экран логина с очисткой стека
      navController.navigate("login") {
        // ОЧИСТКА СТЕКА НАВИГАЦИИ =====

        // popUpTo("home") - удаляет все экраны до "home" из стека
        // inclusive = true - удаляет и сам экран "home" из стека
        // Это предотвращает возврат на главный экран после выхода
        popUpTo("home") { inclusive = true }
      }
    }
  )
}
}

// ФУНКЦИЯ ПРОВЕРКИ АВТОРИЗАЦИИ =====

/**
 * Упрощенная функция проверки статуса авторизации пользователя
 *
 * В реальном приложении здесь была бы проверка через:
 * - SharedPreferences (токены)
 * - ViewModel с состоянием аутентификации
 * - Firebase Auth currentUser
 * - Базу данных или локальное хранилище
 *
 * @return Boolean - true если пользователь авторизован, false если нет
 */
fun isUserLoggedIn(): Boolean {
  // ВРЕМЕННАЯ ЗАГЛУШКА - ВСЕГДА ВОЗВРАЩАЕТ FALSE
  // В реальном приложении здесь должна быть настоящая логика проверки:

  // Пример реальной реализации:
  // return Firebase.auth.currentUser != null

```

```
// или
// return sharedPreferences.getBoolean("is_logged_in", false)
// или
// return authViewModel.isUserAuthenticated

return false
}
```

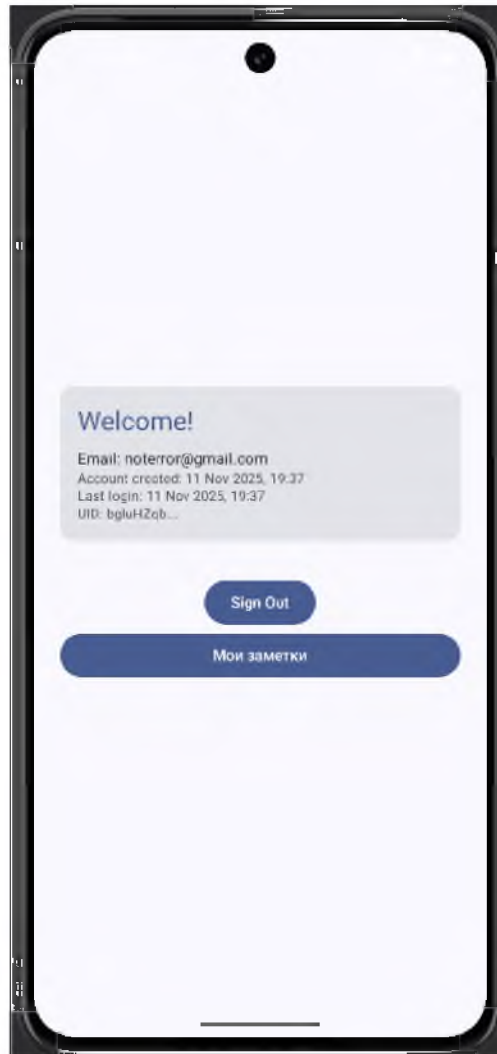


Рисунок 4.10. — Итоговый вид главного экрана

5. Обновим MainActivity для Compose:

```
package com.example.lab4

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxSize
```

```

import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.ui.Modifier
import com.example.lab4.ui.navigation.AuthNavGraph
import com.example.lab4.ui.theme.YourAppTheme

// ГЛАВНАЯ ACTIVITY ПРИЛОЖЕНИЯ =====

/**
 * MainActivity - главная точка входа в Android приложение
 * Наследуется от ComponentActivity для поддержки Jetpack Compose
 */
class MainActivity : ComponentActivity() {

    /**
     * onCreate - основной метод жизненного цикла Activity
     * Вызывается при создании Activity
     *
     * @param savedInstanceState Сохраненное состояние Activity (если пересоздается)
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        // Вызов родительского метода для стандартной инициализации
        super.onCreate(savedInstanceState)

        // НАСТРОЙКА COMPOSE UI =====

        // setContent - устанавливает Compose как корневой UI
        setContent {
            // ПРИМЕНЕНИЕ КАСТОМНОЙ ТЕМЫ ПРИЛОЖЕНИЯ =====

            // YourAppTheme - кастомная тема, определяющая цвета, шрифты и стили
            YourAppTheme {
                // SURFACE - ОСНОВНОЙ КОНТЕЙНЕР UI =====

                // Surface - Material Design контейнер, обеспечивающий:
                // - Фон согласно теме
                // - Elevation (тени)
                // - Кликабельные области
                Surface(
                    // МОДИФИКАТОРЫ ДЛЯ НАСТРОЙКИ ВНЕШНЕГО ВИДА =====

                    // fillMaxSize() - растягивает контейнер на весь доступный размер
                    modifier = Modifier.fillMaxSize(),

```

```

// НАСТРОЙКА ЦВЕТА =====

// color - устанавливает цвет фона из цветовой схемы темы
// MaterialTheme.colorScheme.background - автоматически адаптируется
// под светлую/темную тему
color = MaterialTheme.colorScheme.background
){
// НАВИГАЦИОННЫЙ ГРАФ ПРИЛОЖЕНИЯ =====

// AuthNavGraph - корневой навигационный компонент, который:
// - Управляет переходами между экранами
// - Определяет начальный экран (логин или главный)
// - Обрабатывает состояние аутентификации
AuthNavGraph()
}
}
}
}
}
}
}

```

6. Создадим тему приложения:

```

package com.example.lab4.ui.theme

import android.app.Activity
import android.os.Build
import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.darkColorScheme
import androidx.compose.material3.dynamicDarkColorScheme
import androidx.compose.material3.dynamicLightColorScheme
import androidx.compose.material3.lightColorScheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.SideEffect
import androidx.compose.ui.graphics.toArgb
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalView
import androidx.core.view.WindowCompat

// ОПРЕДЕЛЕНИЕ ЦВЕТОВЫХ ПАЛИТР =====

// ТЕМНАЯ ЦВЕТОВАЯ СХЕМА - используется когда включена темная тема
private val DarkColorScheme = darkColorScheme(
    primary = Purple80, // Основной цвет в темной теме

```

```

    secondary = PurpleGrey80, // Вторичный цвет в темной теме
    tertiary = Pink80        // Третичный/акцентный цвет в темной теме
)

// СВЕТЛАЯ ЦВЕТОВАЯ СХЕМА - используется когда включена светлая тема
private val LightColorScheme = lightColorScheme(
    primary = Purple40,      // Основной цвет в светлой теме
    secondary = PurpleGrey40, // Вторичный цвет в светлой теме
    tertiary = Pink40        // Третичный/акцентный цвет в светлой теме
)

// ОСНОВНАЯ ТЕМА ПРИЛОЖЕНИЯ =====

/**
 * Кастомная тема приложения, которая настраивает цвета, типографику и другие стили
 *
 * @param darkTheme Булево значение, определяющее использование темной темы
 *                По умолчанию определяется автоматически через isSystemInDarkTheme()
 * @param dynamicColor Булево значение, разрешающее использование динамических
цветов
 *                (доступно на Android 12+)
 * @param content Composable лямбда, содержащая UI контент, к которому применяется
тема
 */
@Composable
fun YourAppTheme(
    // Определение темной темы: автоматически или вручную
    darkTheme: Boolean = isSystemInDarkTheme(),
    // Включение динамических цветов (Material You) на Android 12+
    dynamicColor: Boolean = true,
    // Содержимое UI, к которому будет применена тема
    content: @Composable () -> Unit
) {
    // ВЫБОР ЦВЕТОВОЙ СХЕМЫ =====

    val colorScheme = when {
        // УСЛОВИЕ 1: Если динамические цвета разрешены и устройство на Android 12+
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            // Получение контекста для доступа к ресурсам устройства
            val context = LocalContext.current
            // Выбор между динамической темной или светлой схемой
            if (darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(context)
        }

        // УСЛОВИЕ 2: Если включена темная тема (без динамических цветов)
        darkTheme -> DarkColorScheme
    }
}

```

```

// УСЛОВИЕ 3: По умолчанию - светлая тема
else -> LightColorScheme
}

// НАСТРОЙКА СИСТЕМНОГО UI =====

// Получение текущего View для доступа к системным элементам
val view = LocalView.current

// Проверка что мы не в режиме предпросмотра Android Studio
if (!view.isEditMode) {
    // SideEffect - выполняет побочные эффекты при композиции
    SideEffect {
        // Получение Window текущей Activity
        val window = (view.context as Activity).window

        // НАСТРОЙКА ЦВЕТА СТАТУС БАРА (верхняя панель уведомлений)
        // Установка цвета статус бара в соответствии с primary цветом темы
        window.statusBarColor = colorScheme.primary.toArgb()

        // НАСТРОЙКА ВНЕШНЕГО ВИДА СТАТУС БАРА
        // Для темной темы - светлый текст, для светлой - темный текст
        WindowCompat.getInsetsController(window, view).isAppearanceLightStatusBars =
darkTheme
    }
}

// ПРИМЕНЕНИЕ ТЕМЫ MATERIAL DESIGN 3 =====

MaterialTheme(
    // Установка выбранной цветовой схемы
    colorScheme = colorScheme,
    // Установка кастомной типографики (должна быть определена в этом же пакете)
    typography = Typography,
    // Применение темы к переданному контенту
    content = content
)
}

```

6. После того как все будет готово, откройте Firebase Console и перейдите в раздел Authentication во вкладке Build. Активируйте Аутентификацию в вашем приложении нажав на кнопку «Get started» (рисунок 4.11).

Выберите вид авторизации, для приложения выше это Email/Password (рисунок 4.12).

Активируйте и сохраните изменения (рисунок 4.13).

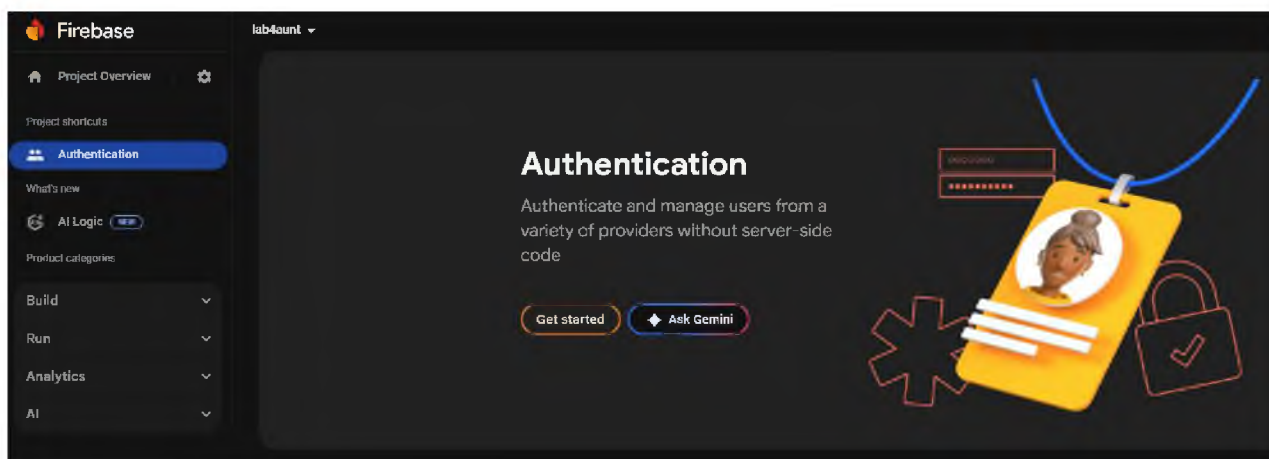


Рисунок 4.11. – Добавление Аутентификации в приложение

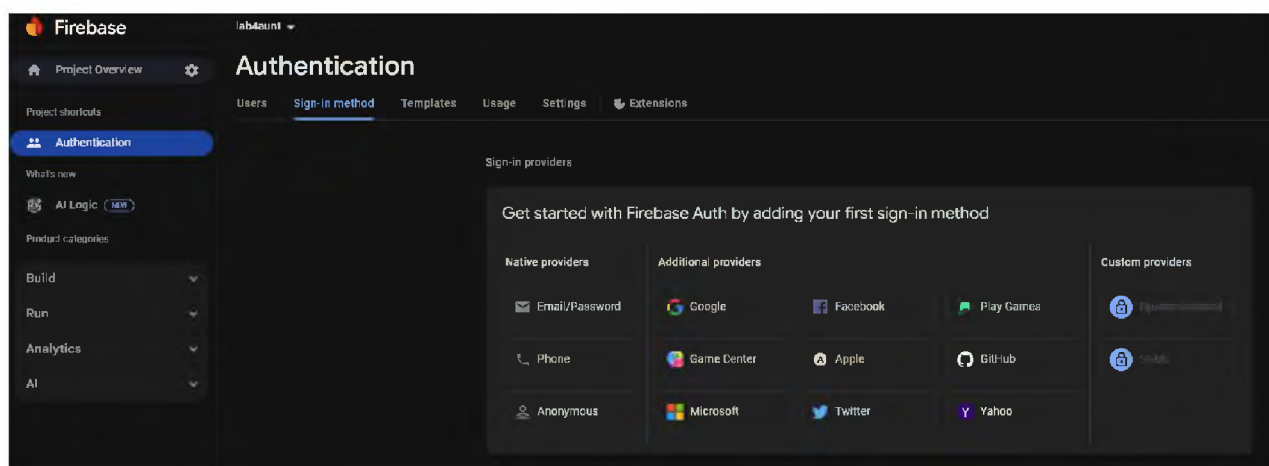


Рисунок 4.12. – Выбор вида авторизации

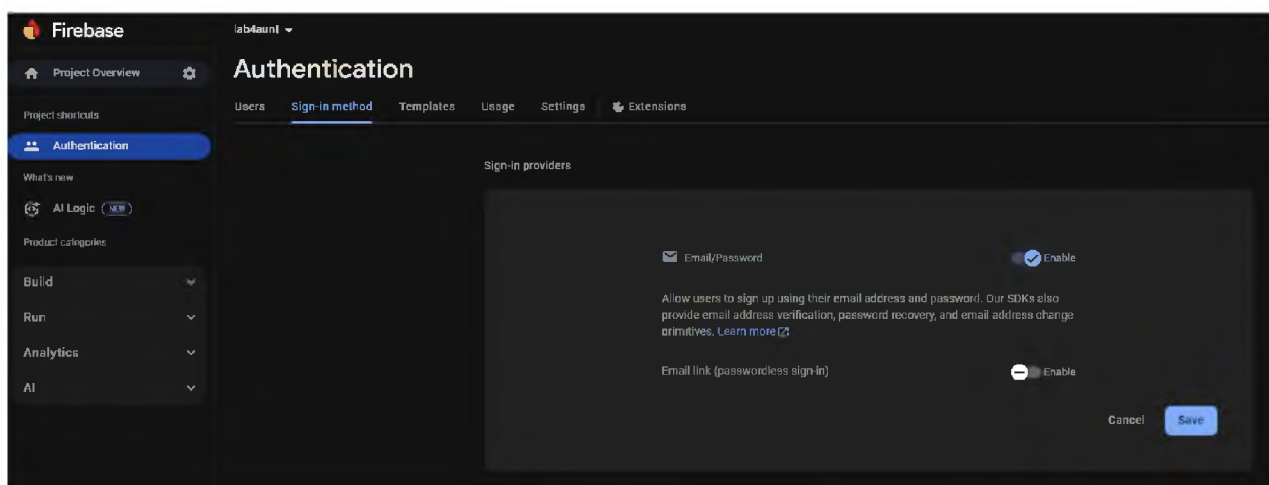


Рисунок 4.13. — Активация Email/Password

7. После запуска приложения и регистрации пользователя, в Firebase можно увидеть его данные (рисунок 4.14).

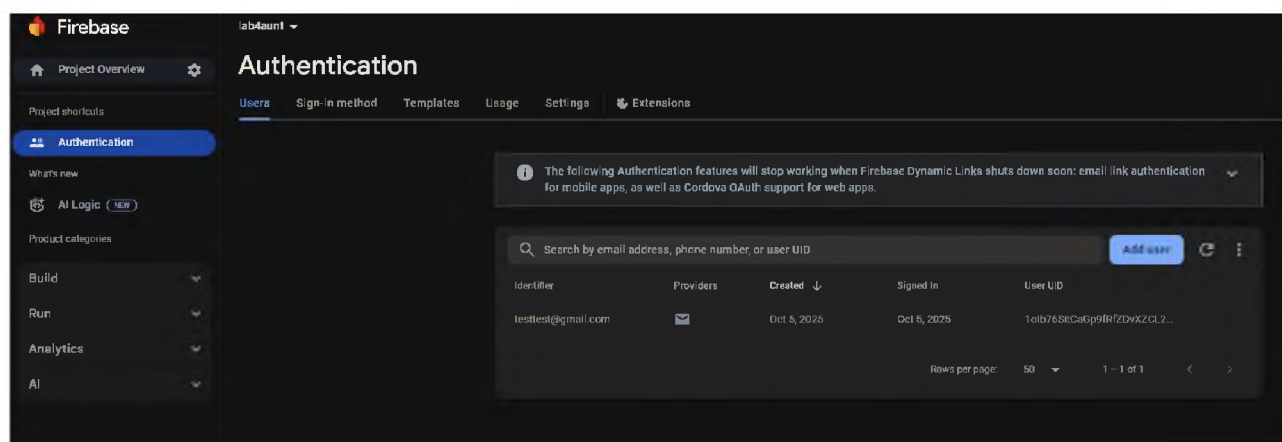


Рисунок 4.14. – Зарегистрированные пользователи приложения

Ход работы

1. Изучить теоретический материал.
2. Выполнить задания.
3. Оформить отчет.
4. Ответить на контрольные вопросы.
5. Загрузить отчет в Classroom.
6. Защитить работу у преподавателя.

Задание

1. Создать новый проект согласно теме варианта.
2. Зарегистрировать приложение в Firebase Console.
3. Добавить аутентификацию в приложении. Реализовать экран регистрации и входа.
4. Проверить сохранение пользователя в Firebase Auth.

Вариант	Тема
1	Путеводитель по музеям. Создать приложение с аутентификацией и возможностью создания личных заметок о посещенных музеях с прикреплением фотографий
2	Садоводческий помощник. Создать приложение с аутентификацией и возможностью создания личных заметок о растениях с прикреплением фотографий

3	Кулинарная книга. Создать приложение с аутентификацией и возможностью создания личных рецептов с прикреплением фотографий
4	Личный финансовый помощник. Создать приложение с аутентификацией и возможностью создания и отображения списка финансовых операций
5	Тренировочный план для домашних питомцев. Создать приложение с аутентификацией и возможностью добавления и удаления личных тренировок
6	Архив личных воспоминаний. Создать приложение с аутентификацией и возможностью добавления воспоминаний с прикреплением фотографий
7	Каталог фильмов и сериалов. Создать приложение с аутентификацией и возможностью добавления фильмов и сериалов с прикреплением фотографий.
8	Организатор задач для студентов. Создать приложение с аутентификацией и возможностью добавления и удаления задач для студентов.
9	Дневник настроения. Создать приложение с аутентификацией и возможностью добавления и удаления пометок о настроении.
10	Организатор мероприятий. Создать приложение с аутентификацией и возможностью добавления мероприятий для студентов с прикреплением фотограф

Контрольные вопросы

1. Каковы минимальные требования к проекту Android для начала работы с Firebase?
2. Какие два основных способа подключения Android приложения к Firebase описаны в работе?
3. Что такое google-services.json и на каком этапе настройки Firebase он добавляется в проект?
4. Какую роль в проекте выполняет плагин com.google.gms.google-services? В каких файлах Gradle его необходимо прописать?
5. Что такое Firebase Android BoM (Bill of Materials)? Какую проблему он решает?
6. Какие четыре состояния определены в sealed class AuthUiState? Для чего используется каждое из них?
7. Как в AuthViewModel организована обработка асинхронных операций (регистрации и входа)? Почему для этого используется viewModelScope?
8. Что относится к функциям сборки в Firebase?

9. Что относится к выпуску и мониторингу в Firebase?
10. Что относится к инструментам вовлечения в Firebase?

Содержание отчета

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

ЛАБОРАТОРНАЯ РАБОТА 5

Работы с сообщениями через Firebase Cloud Messaging в мобильных приложениях

Цель работы: ознакомиться с этапами создания и отправки push-уведомлений и сообщений напрямую на устройства Android с использованием готовых backend-решений Firebase.

Теоретическая часть

Firebase Cloud Messaging (FCM) – бесплатный облачный сервис от Google, который позволяет отправлять push-уведомления и сообщения напрямую на устройства Android, iOS и веб-приложения. FCM широко используется для реализации мгновенных уведомлений, оповещений о событиях, маркетинговых рассылок и других сценариев, где требуется быстрая доставка сообщений пользователям.

Рассмотрим работу с push-уведомлениями FCM. Добавим приветственное сообщение пользователю после успешной авторизации в системе. Для этого необходимо:

1. В ваш проект на Firebase добавить Messaging (рисунок 5.1). Его можно найти во вкладке Run, или напрямую через URL:

<https://console.firebase.google.com/project/БАШ-PROJECT-ID/notification>

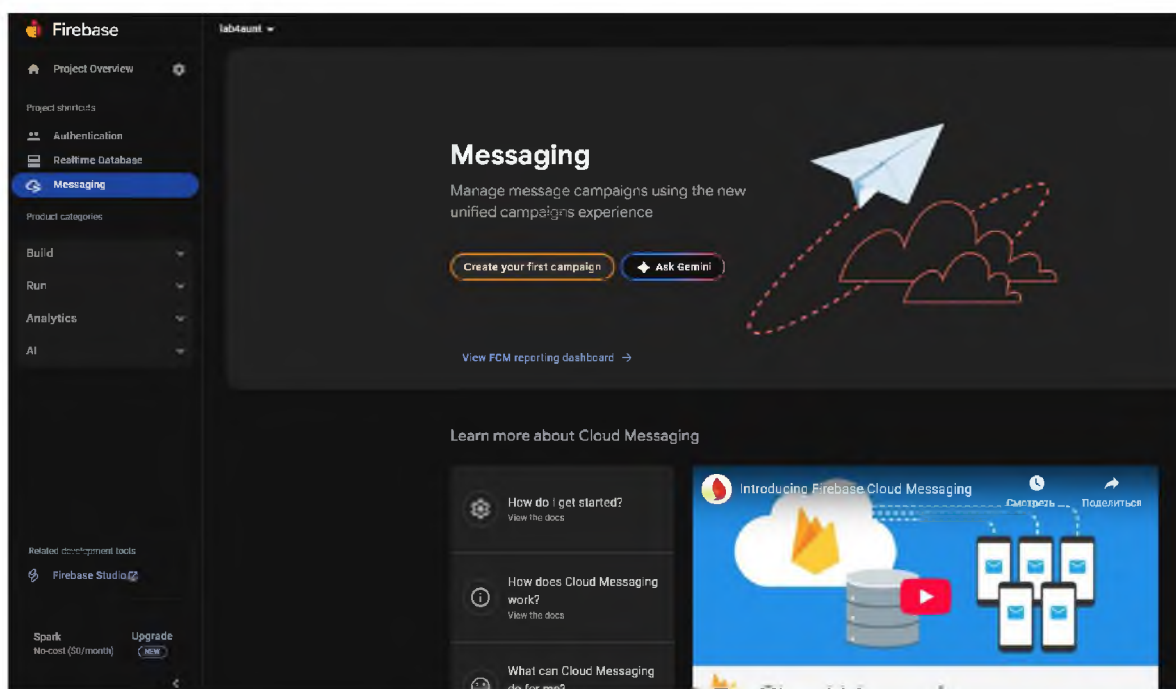


Рисунок 5.1. – Добавление Messaging в проект

2. Теперь необходимо настроить уведомление внутри проекта. Для начала добавим FCM зависимости в файл build.gradle (уровень приложения):

```
dependencies {  
    // ... существующие зависимости  
  
    // FCM (Firebase Cloud Messaging)  
    implementation("com.google.firebase:firebase-messaging-ktx")  
  
    // Дополнительно для работы с корутинами в FCM  
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.7.3")  
}
```

3. Затем создадим Cloud Function (functions/index.js) – серверный код, который выполняется в облаке Firebase. Она нужна для безопасной отправки push-уведомлений, т.к. клиентские приложения не должны иметь прямой доступ к серверным ключам FCM. Функция sendWelcome отправляет сообщение всем устройствам, подписанным на топик "all_users", что позволяет массово уведомлять пользователей.

```
const functions = require('firebase-functions');  
const admin = require('firebase-admin');  
admin.initializeApp();  
  
// Cloud Function для отправки приветственного уведомления  
exports.sendWelcome = functions.https.onCall((data, context) => {  
    const message = {  
        notification: {  
            title: 'Добро пожаловать! 🎉', // Заголовок уведомления  
            body: 'Рады видеть вас в нашем приложении!' // Текст уведомления  
        },  
        topic: 'all_users' // Отправка всем подписанным на топик устройствам  
    };  
  
    return admin.messaging().send(message); // Отправка сообщения через FCM  
});
```

4. Далее реализуем ViewModel для создания и отображения нашего уведомления. ViewModel – это архитектурный компонент Android, который отвечает за хранение и управление данными, связанными с UI. Она переживает изменения конфигурации (например, поворот экрана) и отделяет бизнес-логику от пользовательского интерфейса. В данном случае ViewModel управляет процессом аутентификации и связанными с ним действиями – подпиской на топик уведомлений и отправкой приветственного сообщения.

```

class AuthViewModel : ViewModel() {

    private val functions = Firebase.functions // Клиент для вызова Cloud Functions

    fun signIn(email: String, password: String) {
        viewModelScope.launch { // Запуск корутины в scope ViewModel
            try {
                val result = auth.signInWithEmailAndPassword(email, password).await()

                // После успешной аутентификации подписываем и отправляем приветствие
                subscribeAndSendWelcome()

            } catch (e: Exception) {
                // Обработка ошибок аутентификации
            }
        }
    }

    private fun subscribeAndSendWelcome() {
        // 1. Подписка устройства на топик для получения уведомлений
        FirebaseMessaging.getInstance().subscribeToTopic("all_users")

        // 2. Вызов Cloud Function для отправки приветствия
        viewModelScope.launch {
            try {
                functions.getHttpsCallable("sendWelcome").call()
            } catch (e: Exception) {
                // Игнорируем ошибки отправки приветствия
            }
        }
    }
}

```

5. Теперь необходимо получить и обработать сообщение уведомления. Для этого создадим FCM Service – сервис, который обрабатывает входящие push-уведомления. Когда приложение получает сообщение через FCM, этот сервис перехватывает его и может кастомизировать отображение. Метод `onMessageReceived` вызывается, когда приложение находится на переднем плане (foreground), а для фоновых уведомлений система показывает стандартное уведомление автоматически.

```

class MyFCMService : FirebaseMessagingService() {

    // Обработка входящих FCM сообщений
    override fun onMessageReceived(message: RemoteMessage) {

```

```

message.notification?.let {
    // Показываем уведомление с полученными данными
    showNotification(it.title ?: "", it.body ?: "")
}
}

private fun showNotification(title: String, body: String) {
    val manager = getSystemService(NOTIFICATION_SERVICE) as NotificationManager

    // Создание канала уведомлений для Android 8.0+
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val channel = NotificationChannel(
            "default", // ID канала
            "Уведомления", // Отображаемое имя
            NotificationManager.IMPORTANCE_DEFAULT // Уровень важности
        )
        manager.createNotificationChannel(channel)
    }

    // Создание и отображение уведомления
    val notification = NotificationCompat.Builder(this, "default")
        .setContentTitle(title) // Заголовок уведомления
        .setContentText(body) // Текст уведомления
        .setSmallIcon(android.R.drawable.ic_dialog_info) // Иконка
        .setAutoCancel(true) // Автоматическое закрытие при тапе
        .build()

    manager.notify(1, notification) // Показ уведомления
}
}

```

6. Далее регистрируем FCM сервис в AndroidManifest, чтобы система знала, какой компонент должен обрабатывать входящие push-уведомления.

```

<service android:name=".MyFCMService">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>

```

7. Завершающим шагом будет создание самого уведомления в Firebase. Для этого, вернемся в Firebase, перейдем в раздел нашего проекта, нажмем кнопку создания уведомления «Create your first campaign». В открывшемся окне на выбор предлагается два вида уведомления (рисунок 5.2):

– уведомления Firebase. «Отправляйте уведомления, чтобы мгновенно информировать пользователей о специальных предложениях или новых функциях.

Пользователи будут получать уведомления, даже если они находятся за пределами вашего приложения;

– сообщения Firebase In-App. «Отправляйте сообщения In-App, чтобы побудить активных пользователей вашего приложения подписаться, посмотреть видео, пройти уровень или купить товар. Пользователи смогут получать сообщения только внутри вашего приложения.

Для нашего приложения подходит первый вариант.

После нажатия кнопки Create откроется настройка уведомления, где необходимо заполнить следующие поля (рисунок 5.3): заголовок уведомления (Notification title), текст уведомления (Notification text), изображение уведомления (Notification image), название уведомления (Notification name – идентификатор уведомления, не отображается пользователю), предпросмотр (Preview Section – показывает отображение уведомления, также можно выбрать устройство для предпросмотра на нем «Send test message»). После заполнения нажимаем кнопку «Next».

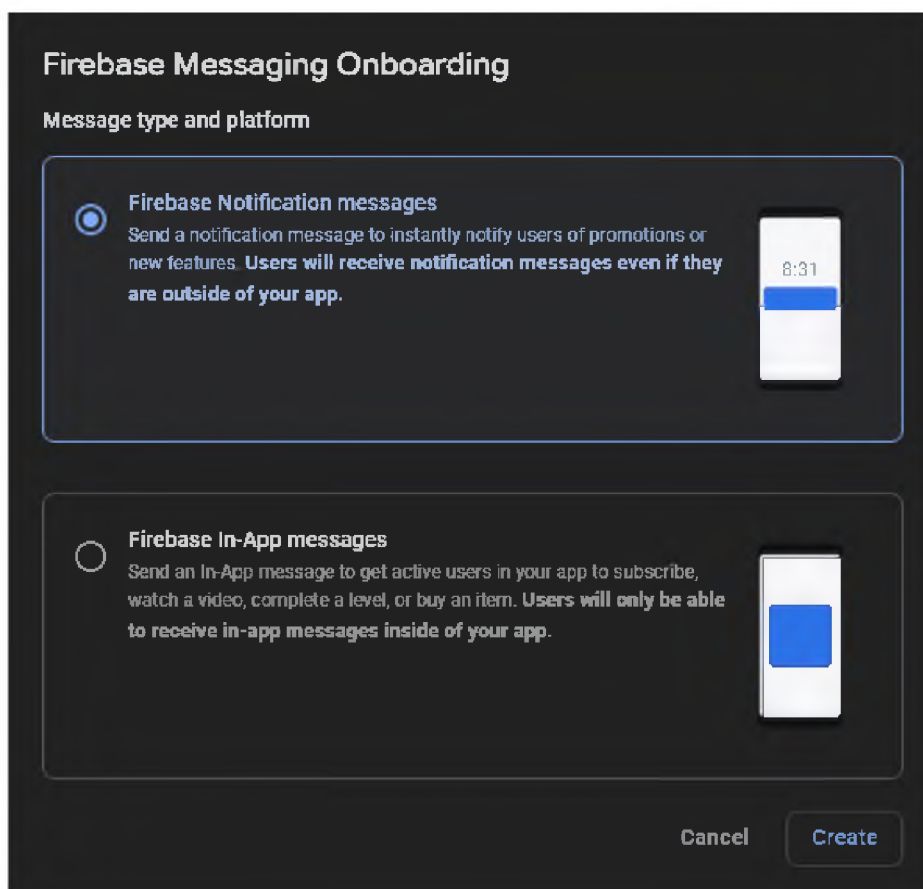


Рисунок 5.2. – Выбор типа и платформы уведомлений

1 Notification

Notification title ⓘ
Добро пожаловать! 📄

Notification text
Рады видеть вас в нашем приложении!

Notification image (optional) ⓘ
Example: <https://yourapp.com/image.png>

Notification name (optional) ⓘ
Приветственное сообщение

Send test message

Initial state Expanded view

Android

Apple

Next

Рисунок 5.3 – Первый шаг настройки уведомления

Вторым шагом для заполнения будет Target (Целевая аудитория) – определяет, кому будет отправлено уведомление (рисунок 5.4). Пользовательский сегмент (User segment) отвечает за пользователей, которым будет отправлено уведомление, это могут быть: все пользователи, пользователям по id и пользователям с заранее созданным сегментам. Топик/тема (Topic): Message topic – отправка по подписке на тему. Планирование отправки (Scheduling) – когда уведомление будет отправлено: отправить сейчас, запланировать на потом. Ключевые события (Key events) – привязка уведомлений к событиям в аналитике. Дополнительные опции: платформа для отправки, приоритет доставки, время жизни сообщения, группировка сообщений, добавление пользовательских данных, ключ-значение для обработки в приложении.

Основными шагами для заполнения являются 1–3, пример заполнения можно посмотреть на рисунках 5.3 и 5.4. После заполнения и нажатия кнопки Review откроется окно предпросмотра заполненных данных (рисунок 5.5).

The screenshot shows a dark-themed interface for configuring a notification. A vertical progress bar on the left indicates five steps: 1. Notification, 2. Target (active), 3. Scheduling, 4. Key events (optional), and 5. Additional options (optional). The 'Notification' step shows the text 'Рады видеть вас в нашем приложении!'. The 'Target' step has two tabs: 'User segment' and 'Topic', with 'Topic' selected. Below the tabs, the 'Message topic' field contains the text 'welcome'. A 'Next' button is located below the text field. At the bottom right, there are two buttons: 'Save as draft' and 'Review'.

Notification
Рады видеть вас в нашем приложении!

2 Target

User segment Topic

Message topic ⓘ

welcome

Next

3 Scheduling
Send now

4 Key events (optional)

5 Additional options (optional)

Save as draft Review

Рисунок 5.4. – Настройка целевой аудитории

The screenshot shows a dark-themed interface for reviewing a notification. The title 'Review message' is at the top. Below it, the 'Notification content' section shows a list icon and the text 'Рады видеть вас в нашем приложении!'. The 'Target' section shows a target icon and the text 'Subscribers of welcome topic'. The 'Scheduling' section shows a calendar icon and the text 'Send now'. At the bottom right, there are two buttons: 'Cancel' and 'Publish'.

Review message

Notification content

Рады видеть вас в нашем приложении!

Target

Subscribers of welcome topic

Scheduling

Send now

Cancel Publish

Рисунок 5.5. – Предпросмотр

После предпросмотра и нажатия кнопки Publish наше уведомление будет создано и отобразится в уже запущенном приложении.

Итоговый результат выполнения программы представлен на рисунке 5.6.

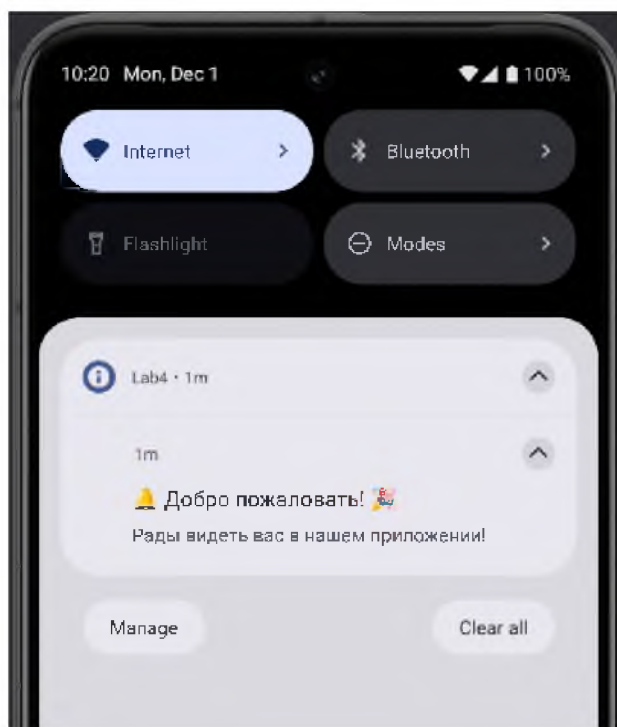


Рисунок 5.6. – Полученное уведомление

Ход работы

1. Изучить теоретический материал.
2. Выполнить задания.
3. Оформить отчет.
4. Ответить на контрольные вопросы.
5. Загрузить отчет в Classroom.
6. Защитить работу у преподавателя.

Задание

1. Продолжить разработку проекта из лабораторной работы 4.
2. Добавить отправку уведомления всем пользователям через FCM, согласно варианту.

Контрольные вопросы

1. Что такое Firebase Cloud Messaging (FCM)?
2. В чем заключается разница между обработкой push-уведомлений, когда приложение находится на переднем плане (foreground) и в фоне (background)?

3. Какие два основных типа сообщений можно создавать через консоль Firebase?

4. Опишите последовательность действий для добавления push-уведомлений через FCM в готовый проект.

Вариант	Тема
1	Путеводитель по музеям. Добавить уведомление об успешно созданной заметке о музеях.
2	Садоводческий помощник. Добавить уведомление об успешном добавлении нового растения.
3	Кулинарная книга. Реализовать получение уведомления об успешном добавлении нового рецепта.
4	Личный финансовый помощник. Добавить уведомление о получении или списании средств.
5	Тренировочный план для домашних питомцев. Добавить уведомление об завершенной тренировке.
6	Архив личных воспоминаний. Добавить уведомление об успешно созданном воспоминании.
7	Каталог фильмов и сериалов. Добавить уведомление об успешном добавлении фильма/сериала.
8	Организатор задач для студентов. Добавить уведомление об успешно завершенных задачах.
9	Дневник настроения. Добавить уведомление об настроении пользователя, после его авторизации в приложении.
10	Организатор мероприятий. Добавить уведомление об успешно созданном мероприятии.

Содержание отчета

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Установка среды разработки

Первый шаг в разработке приложений для мобильных устройств является установка необходимого программного обеспечения, в данном случае Android Studio Ladybug Feature Drop 2024.2.2. Для разработки на мобильные платформы необходимо установить не только саму среду разработки, но и SDK (Software Development Kit) – инструменты для создания приложений на одну определенную платформу, в данном случае – ОС Android.

Приложения для операционной системы Android могут быть разработаны на разных языках. Android Studio поддерживает такие языки, как Java, C++, Kotlin. Java доступен без какой-либо предварительной подготовки, Kotlin – после подключения одноименного модуля в самой Android Studio. Для сборки проектов с использованием C/C++ необходим NDK – средства нативной разработки, позволяющие компилировать код на C/C++ под ОС Android.

Минимальный пакет для начала разработки – среда и набор SDK, который Android Studio поставит самостоятельно.

Процесс установки Android Studio не отличается от установки любого программного обеспечения.

1. Загрузите файл установщика можно с официального сайта: <https://developer.android.com/studio> (рисунок П.1).

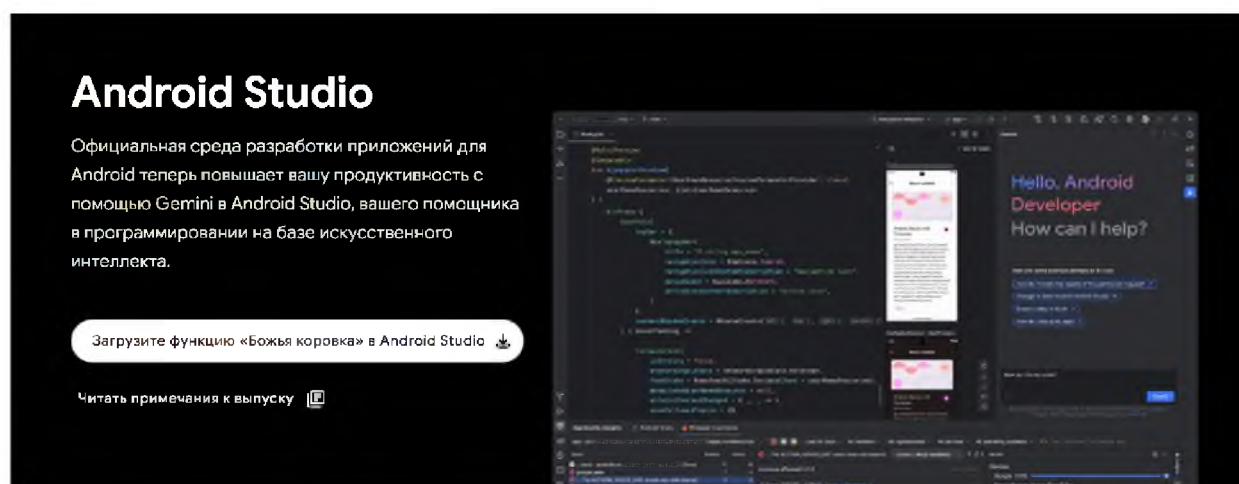


Рисунок П.1 – Загрузка файла установщика

2. Затем ознакомьтесь с условиями лицензионного соглашения, отметьте соответствующий пункт внизу и нажмите на кнопку для старта скачивания (рисунок П.2).

14. General Legal Terms

14.1 The License Agreement constitutes the whole legal agreement between you and Google and governs your use of the SDK (excluding any services which Google may provide to you under a separate written agreement), and completely replaces any prior agreements between you and Google in relation to the SDK. 14.2 You agree that if Google does not exercise or enforce any legal right or remedy which is contained in the License Agreement (or which Google has the benefit of under any applicable law), this will not be taken to be a formal waiver of Google's rights and that those rights or remedies will still be available to Google. 14.3 If any court of law, having the jurisdiction to decide on this matter, rules that any provision of the License Agreement is invalid, then that provision will be removed from the License Agreement without affecting the rest of the License Agreement. The remaining provisions of the License Agreement will continue to be valid and enforceable. 14.4 You acknowledge and agree that each member of the group of companies of which Google is the parent shall be third party beneficiaries to the License Agreement and that such other companies shall be entitled to directly enforce, and rely upon, any provision of the License Agreement that confers a benefit on (or rights in favor of) them. Other than this, no other person or company shall be third party beneficiaries to the License Agreement. 14.5 EXPORT RESTRICTIONS. THE SDK IS SUBJECT TO UNITED STATES EXPORT LAWS AND REGULATIONS. YOU MUST COMPLY WITH ALL DOMESTIC AND INTERNATIONAL EXPORT LAWS AND REGULATIONS THAT APPLY TO THE SDK. THESE LAWS INCLUDE RESTRICTIONS ON DESTINATIONS, END USERS AND END USE. 14.6 The rights granted in the License Agreement may not be assigned or transferred by either you or Google without the prior written approval of the other party. 14.7 The License Agreement, and your relationship with Google under the License Agreement, shall be governed by the laws of the State of California without regard to its conflict of laws provisions. You and Google agree to submit to the exclusive jurisdiction of the courts located within the county of Santa Clara, California to resolve any legal matter arising from the License Agreement. Notwithstanding this, you agree that Google shall still be allowed to apply for injunctive remedies (or an equivalent type of urgent legal relief) in any jurisdiction. July 27, 2021

☒ Я прочитал и согласен с вышеуказанными условиями

Скачать Android Studio Ladybug Feature Drop | 2024.2.2 для Windows

android-studio-2024.2.2.13-windows.exe

Рисунок П.2 – Ознакомление и согласие с условиями лицензионного соглашения

3. Дождитесь окончания загрузки файла, после чего запустите его двойным нажатием левой кнопки мыши. Установите программное обеспечение, следуя подсказкам пошагового мастера (рисунок П.3).

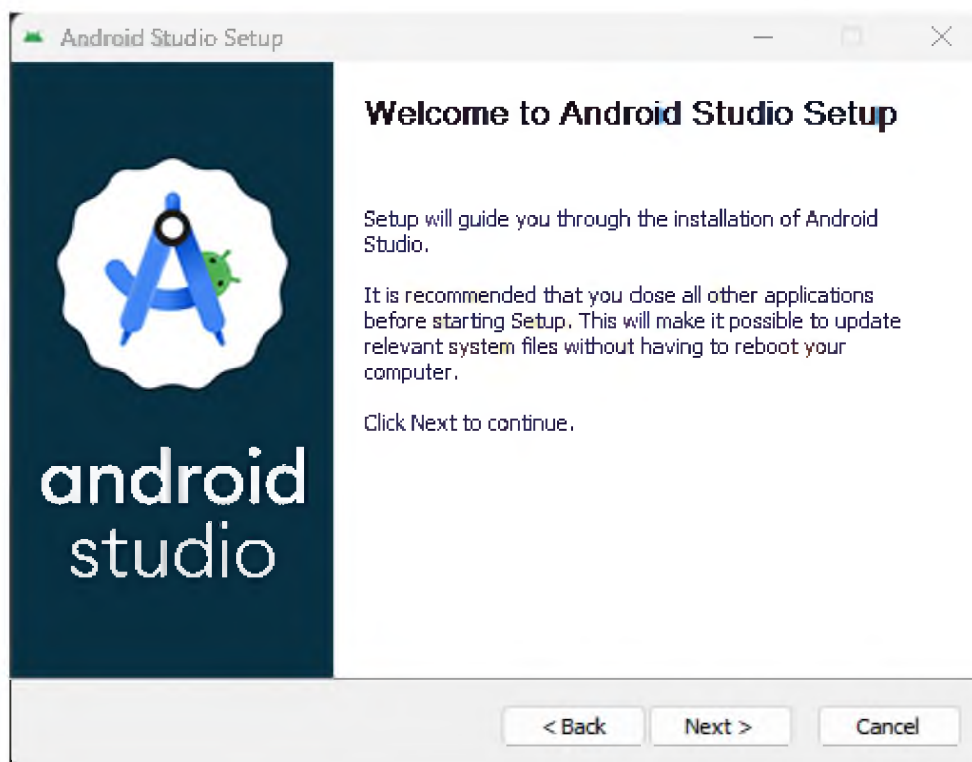


Рисунок П.3. – Установка программного обеспечения

4. После успешной установки запустите софт. Должно отобразиться приветственное окно с предварительной настройкой. Кликните по «Next» (рисунок П.4).

5. Отметьте пункт «Standart» на следующем шаге – это позволит установить общие настройки (рисунок П.5).

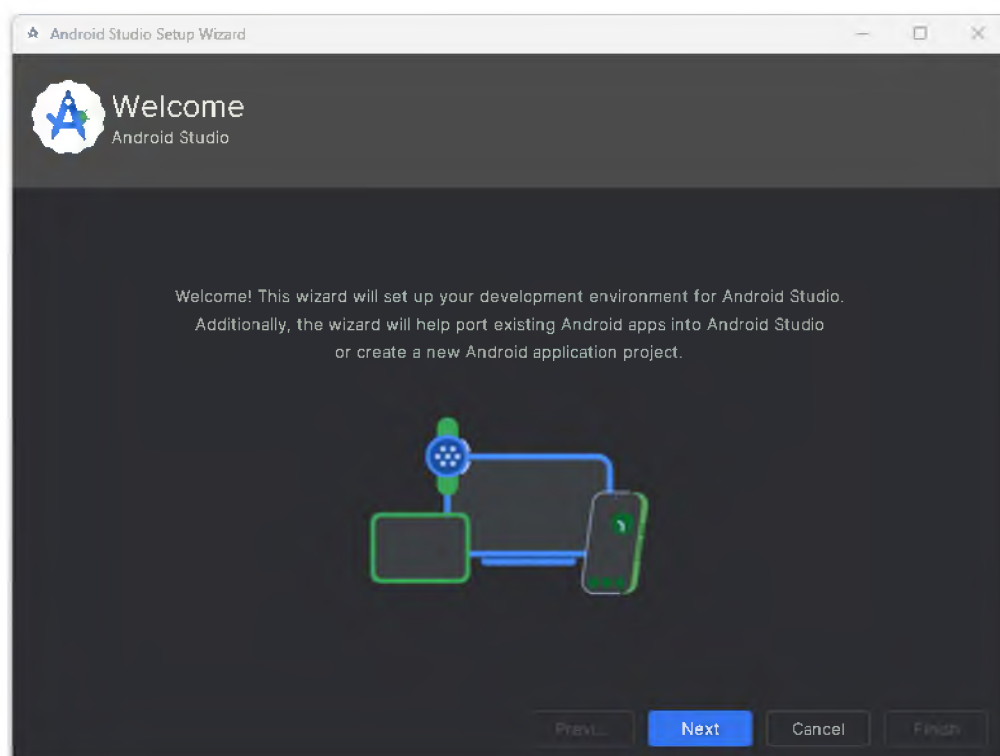


Рисунок П.4. – Запуск Android Studio

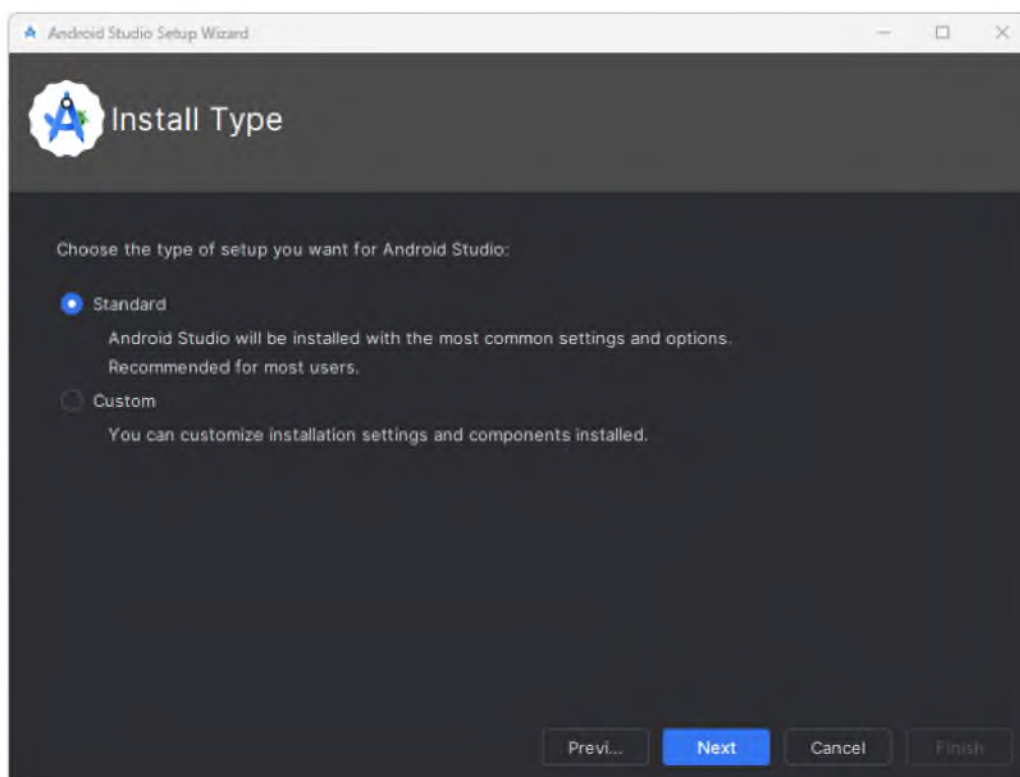


Рисунок П.5. – Установка общих настроек

6. Отобразится окно со списком компонентов и инструментов, которые будут установлены. Проверьте, все ли так, и нажмите на «Next» (рисунок П.6).

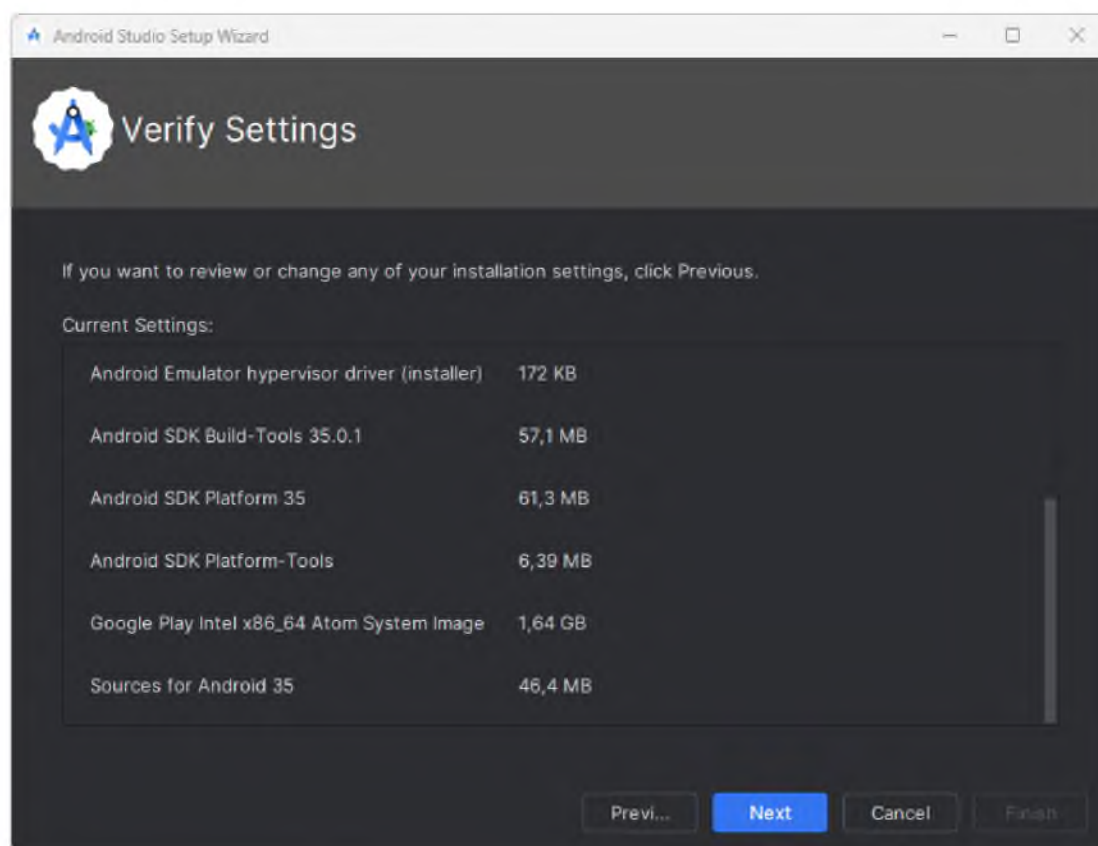


Рисунок П.6. – Список компонентов и инструментов

7. В следующем окне отметьте «Accept», таким образом согласившись с условиями лицензионного использования, затем щелкните по «Finish» внизу (рисунок П.7). Останется дождаться окончания скачивания всех требующихся программных компонентов. При необходимости можно проследить за процессом, нажав на кнопку «Show Details»

После успешной установки среды разработки может потребоваться ее настройка – настройка (загрузка) необходимых пакетов SDK. Посмотреть/установить/удалить их можно перейдя во вкладку Tools->SDK Manager после открытия/создания проекта в меню. Откроется окно, в котором показан список установленных/доступных для установки пакетов, имеющий три вкладки: SDK Platforms, SDK Tools, SDK Update Sites. SDK Platforms (рисунок П.8) – содержит необходимые файлы для разработки для какой-либо версии API. SDK Update Sites – откуда будут скачиваться SDK. SDK Tools – включают в себя необходимые инструменты для разработки и тестирования приложения.

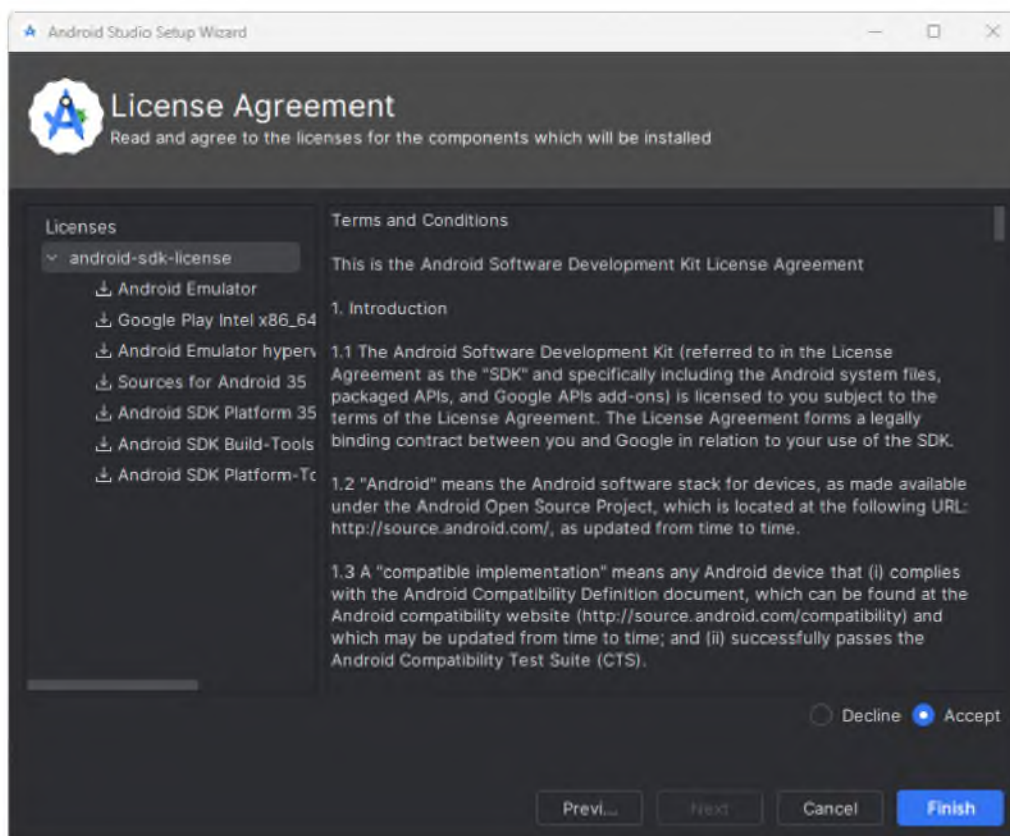


Рисунок П.7. – Соглашение с условиями лицензионного использования

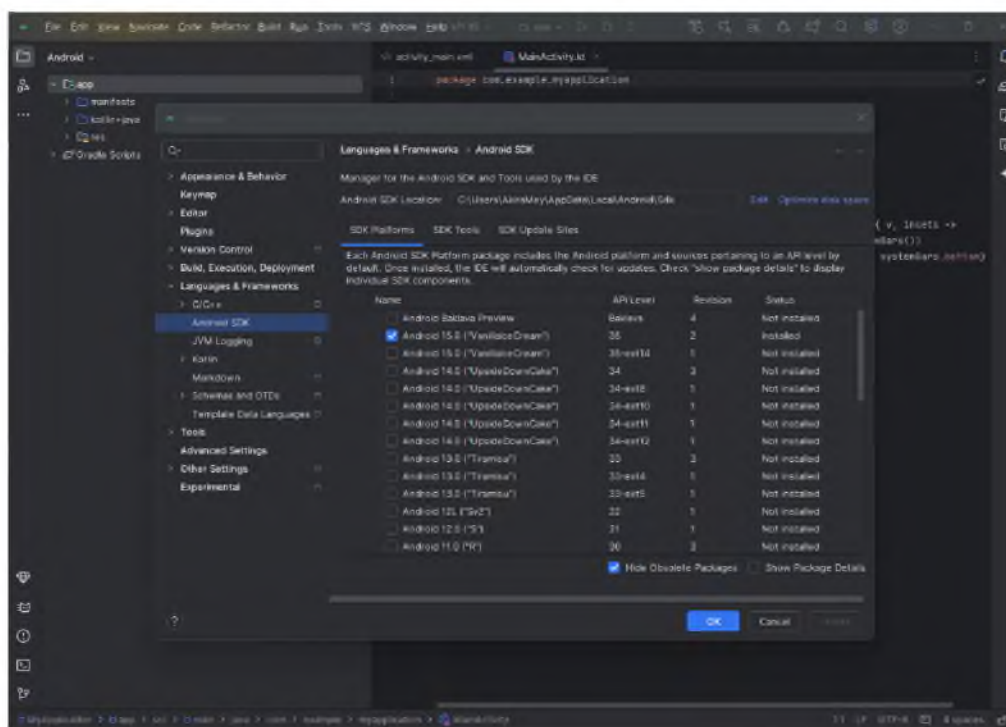


Рисунок П.8. – Окно SDK Manager

Для первого запуска проекта в большинстве случаев достаточно набора пакетов, которые среда разработки ставит самостоятельно. Это основные SDK максимального API на момент установки среды и основные инструменты. Если при сборке проекта среде понадобится неустановленный пакет, то в информации об ошибке среда разработки укажет ссылку на установку необходимого компонента. При переходе по ссылке откроется SDK Manager.

Литература

1. Пирская, Л. В. Разработка мобильных приложений в среде Android Studio : учеб. пособие / Л. В. Пирская. – М. : ЛитРес, 2022. – 125 с.
2. Разработка приложений для ОС Android : учеб. пособие / И. В. Кузнецов, М. С. Исаев, Ю. В. Пономарчук и др. – Хабаровск : ДВГУПС, 2023. – 112 с. – URL: <https://e.lanbook.com/book/433625> (дата обращения: 24.02.2025).
3. Петросян, Л. Э. Разработка мобильных приложений на Kotlin : учебное пособие / Л. Э. Петросян, Н. А. Приходько. – М. : РТУ МИРЭА, 2024. – 101 с. – URL: <https://e.lanbook.com/book/421091> (дата обращения: 22.04.2025).
4. Android для разработчиков // Официальная документация Android Developers. – 2024. – URL: <https://developer.android.com/> (дата обращения: 15.12.2025).
5. Документация по языку Kotlin // JetBrains. – 2024. – URL: <https://kotlinlang.org/docs/home.html> (дата обращения: 15.12.2025).
6. Jetpack Compose: современный UI-инструментарий для Android // Android Developers. – 2024. – URL: <https://developer.android.com/jetpack/compose> (дата обращения: 15.12.2025).
7. Room: библиотека для работы с базой данных SQLite // Android Developers. – 2024. – URL: <https://developer.android.com/training/data-storage/room> (дата обращения: 15.12.2025).
8. Интеграция Firebase в приложения для Android // Firebase Documentation. – 2024. – URL: <https://firebase.google.com/docs/android> (дата обращения: 15.12.2025).
9. Material Design 3 (Material You) для Android // Material Design. – 2024. – URL: <https://m3.material.io/develop/android> (дата обращения: 15.12.2025).
10. Руководство по компонентам Android Jetpack // Android Developers. – 2024. – URL: <https://developer.android.com/jetpack> (дата обращения: 15.12.2025).
11. Навигация в приложении с использованием Navigation Component // Android Developers. – 2024. – URL: <https://developer.android.com/guide/navigation> (дата обращения: 15.12.2025).
12. Руководство пользователя Android Studio // Android Developers. – 2024. – URL: <https://developer.android.com/studio/intro> (дата обращения: 15.12.2025).
13. Настройка системы сборки Android с помощью Gradle // Android Developers. – 2024. – URL: <https://developer.android.com/studio/build> (дата обращения: 15.12.2025).
14. Google Codelabs для Android-разработчиков: практические руководства // Google Developers. – 2024. – URL: <https://codelabs.developers.google.com/?cat=Android> (дата обращения: 15.12.2025).
15. Официальный блог для Android-разработчиков // Google. – 2024. – URL: <https://android-developers.googleblog.com/> (дата обращения: 15.12.2025).
16. Android Developers на русском языке (частичный перевод) // Android Developers. – 2024. – URL: <https://developer.android.com/ru> (дата обращения: 15.12.2025).
17. Документация Kotlin на русском языке // Kotlin. – 2024. – URL: <https://kotlinlang.ru/docs/> (дата обращения: 15.12.2025).
18. Хаб «Разработка под Android» на платформе Хабрахабр // Habr. – 2024. – URL: https://habr.com/ru/hub/android_dev/ (дата обращения: 15.12.2025).