**Структура типичного мобильного приложения: активности и интенты, архитектура MVVM.**

**1. Активности (Activities) и Интенты (Intents)**

Как и в предыдущих объяснениях, Активности - это основные строительные блоки пользовательского интерфейса. Интенты используются для навигации между Активностями и выполнения других действий (например, запуск сервиса). В контексте MVVM, Активности выполняют роль View (частично) и связаны с ViewModel.

**2. Архитектура MVVM (Model-View-ViewModel)**

MVVM – это архитектурный шаблон проектирования, призванный упростить разработку и тестирование пользовательских интерфейсов, чётко разделяя ответственность между компонентами. Он является эволюцией шаблона MVC, устраняя некоторые недостатки последнего.

Model: Представляет данные приложения и логику работы с ними. Это может быть:

Локальная база данных (Room)

Удалённый API (Retrofit)

Репозиторий, который объединяет источники данных.

View: Отвечает за отображение данных и взаимодействие с пользователем. В Android, View обычно состоит из:

XML-разметки для компоновки интерфейса.

Activity или Fragment, которые отвечают за отображение XML-разметки и обработку событий пользовательского интерфейса. Важно: View должна быть максимально пассивной и делегировать всю логику ViewModel.

ViewModel: Действует как посредник между Model и View.

Содержит логику представления (presentation logic). Преобразует данные из Model в формат, удобный для отображения во View.

Предоставляет View данные через LiveData (или StateFlow, Flow). Это позволяет View автоматически обновляться при изменении данных.

Обрабатывает пользовательские события (например, нажатия кнопок). ViewModel не имеет прямой ссылки на View, но может запрашивать у него информацию через callback-и или LiveData.

Не содержит ссылок на View (Activity/Fragment). Это критически важно для тестируемости и устойчивости к изменениям конфигурации устройства (например, поворот экрана). ViewModel переживает повороты экрана и другие изменения конфигурации.

Управляется жизненным циклом View с помощью ViewModelProvider.

**Пример реализации MVVM (Список задач):**

**Model:**

// Task.kt

data class Task(val id: Int, val title: String, val description: String, val completed: Boolean)

// TaskDao.kt (Room Data Access Object)

@Dao

interface TaskDao {

@Query("SELECT \* FROM tasks")

fun getAllTasks(): LiveData<List<Task>>

@Insert

suspend fun insertTask(task: Task)

@Update

suspend fun updateTask(task: Task)

@Delete

suspend fun deleteTask(task: Task)

}

// TaskDatabase.kt (Room Database)

@Database(entities = [Task::class], version = 1)

abstract class TaskDatabase : RoomDatabase() {

abstract fun taskDao(): TaskDao

}

// TaskRepository.kt (Репозиторий)

class TaskRepository(private val taskDao: TaskDao) {

val allTasks: LiveData<List<Task>> = taskDao.getAllTasks()

suspend fun insertTask(task: Task) {

taskDao.insertTask(task)

}

suspend fun updateTask(task: Task) {

taskDao.updateTask(task)

}

suspend fun deleteTask(task: Task) {

taskDao.deleteTask(task)

}

}

**View:**

// MainActivity.kt

class MainActivity : AppCompatActivity() {

private lateinit var binding: ActivityMainBinding // Используем View Binding

private lateinit var taskViewModel: TaskViewModel

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

binding = ActivityMainBinding.inflate(layoutInflater)

setContentView(binding.root)

// Получаем экземпляр ViewModel

taskViewModel = ViewModelProvider(this).get(TaskViewModel::class.java)

// Настраиваем RecyclerView

val adapter = TaskAdapter()

binding.recyclerView.adapter = adapter

binding.recyclerView.layoutManager = LinearLayoutManager(this)

// Наблюдаем за LiveData из ViewModel

taskViewModel.allTasks.observe(this, Observer { tasks ->

tasks?.let { adapter.submitList(it) } // Используем DiffUtil для эффективного обновления списка

})

// Обработчик нажатия кнопки добавления задачи

binding.fabAddTask.setOnClickListener {

val intent = Intent(this, AddTaskActivity::class.java)

startActivity(intent)

}

}

}

// activity\_main.xml

<androidx.constraintlayout.widget.ConstraintLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<androidx.recyclerview.widget.RecyclerView

android:id="@+id/recyclerView"

android:layout\_width="0dp"

android:layout\_height="0dp"

app:layout\_constraintTop\_toTopOf="parent"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintStart\_toStartOf="parent"

app:layout\_constraintEnd\_toEndOf="parent" />

<com.google.android.material.floatingactionbutton.FloatingActionButton

android:id="@+id/fabAddTask"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_margin="16dp"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintEnd\_toEndOf="parent"

android:src="@android:drawable/ic\_input\_add" />

</androidx.constraintlayout.widget.ConstraintLayout>

**ViewModel:**

// TaskViewModel.kt

class TaskViewModel(application: Application) : AndroidViewModel(application) {

private val repository: TaskRepository

val allTasks: LiveData<List<Task>>

init {

val taskDao = TaskDatabase.getDatabase(application).taskDao()

repository = TaskRepository(taskDao)

allTasks = repository.allTasks

}

fun insertTask(task: Task) = viewModelScope.launch(Dispatchers.IO) {

repository.insertTask(task)

}

fun updateTask(task: Task) = viewModelScope.launch(Dispatchers.IO) {

repository.updateTask(task)

}

fun deleteTask(task: Task) = viewModelScope.launch(Dispatchers.IO) {

repository.deleteTask(task)

}

}

**Пояснения:**

LiveData: LiveData - это наблюдаемый контейнер данных, который знает о жизненном цикле View. ViewModel предоставляет данные View через LiveData. Когда данные в LiveData изменяются, View автоматически обновляется.

ViewModelProvider: ViewModelProvider - это механизм для получения экземпляров ViewModel, связанных с жизненным циклом Activity/Fragment. ViewModel сохраняется при изменении конфигурации (например, поворот экрана).

viewModelScope: viewModelScope - это корутин Scope, связанный с жизненным циклом ViewModel. Когда ViewModel уничтожается, все запущенные в этом скоупе корутины отменяются. Это предотвращает утечки памяти.

Dispatchers.IO: Используем Dispatchers.IO для выполнения операций ввода/вывода (например, работа с базой данных) в фоновом потоке, чтобы не блокировать основной поток UI.

AndroidViewModel: Наследуемся от AndroidViewModel вместо ViewModel, чтобы иметь доступ к Application context. Это может быть необходимо для доступа к системным сервисам.

**Преимущества MVVM:**

Тестируемость: ViewModel легко протестировать, так как она не зависит от View и Android SDK.

Разделение ответственности: Четкое разделение ответственности между UI (View), логикой представления (ViewModel) и данными (Model).

Удобство поддержки и масштабирования: Архитектура способствует более структурированному коду, что облегчает его понимание и поддержку.

Управляемость жизненным циклом: ViewModel переживает изменения конфигурации (например, повороты экрана), что предотвращает потерю данных.

Data Binding (опционально): MVVM хорошо сочетается с Data Binding Library, что позволяет связать View и ViewModel декларативно, уменьшая количество boilerplate кода.

\_\_\_\_\_

**Написать в мобильном приложении получение массива данных с API используя паттерн MVVM. Ссылка на API расположена в каталоге «Экзамен».**