Работа с базой данных

SQLite и Room

База данных является наиболее распространенным местом, где мобильное приложение хранит свои данные. Наиболее популярной с системой управления базами данных в мобильном приложении является SQLite. И Android SDK предоставляет необходимый функционал для работы с SQLite. Конечно, прежде чем углубляться в особенности SQLite в контексте разработки под Android, необходимо иметь базовое понимание запросов SQL. Для этого можно обратиться к соответствующему руководству на данном сайте - Руководство по SQLite. Здесь же мы будет рассматривать работу с SQLite непосредственно в контексте приложения на Jetpack Compose на Android без отсылки к основам SQLite.

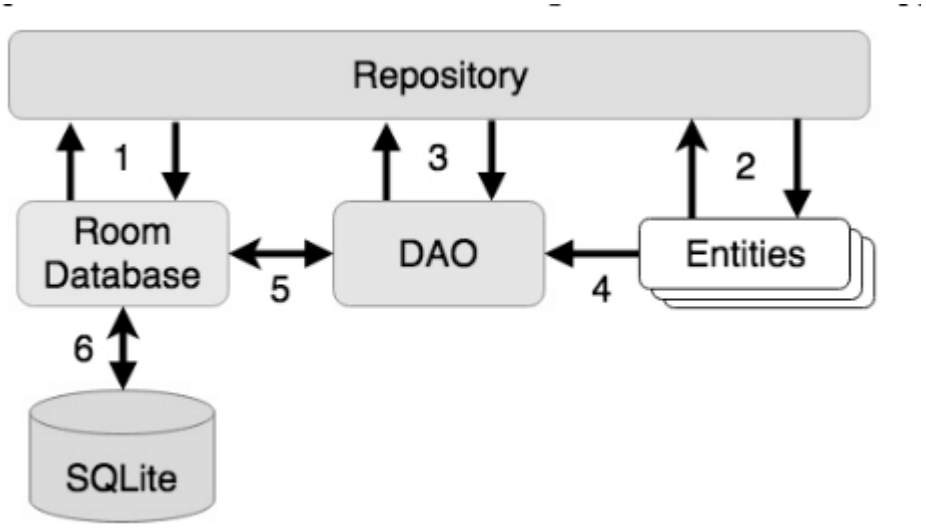
Базы данных SQLite размещаются в файлах (обычно с раширением *.db), которые зачастую расположены по следующему пути:

```
/data/data/<имя_пакета>/databases/<имя_базы_данных>.db
```

Например, если пакет приложения называется "com.example.helloapp", и оно использует базу данных "mydatabase.db", то путь к файлу базы данных будет следующим:

```
/data/data/com.example.helloapp/databases/mydatabase.db
```

Для работы с SQLite Jetpack Compose имееет специальную библиотеку Room, которая предоставляет высокоуровневый интерфейс поверх системы базы данных SQLite и которая значительно упрощает взаимодействие с базами данными. Вся схема взаимодействия с базой данных выглядит следующим образом:



Прежде всего для получения данных из базы данных обычно определяется отдельный модуль репозитория. Однако код приложения, в частности, репозиторий не обращается напрямую к базе данных. Вместо этого все операции с базой данных выполняются с использованием комбинации базы

данных Room, DAO и сущностей. DAO (Data Access Object - объект доступа к данным) содержит операторы SQL, необходимые репозиторию для добавления, извлечения и удаления данных в базе данных SQLite. Эти инструкции SQL сопоставляются с методами, которые затем вызываются из репозитория для выполнения соответствующего запроса.

Объект базы данных Room предоставляет интерфейс к базовой базе данных SQLite. Он также предоставляет репозиторию доступ к объекту DAO. Приложение должно иметь только один экземпляр базы данных Room, который можуj использовать для доступа к нескольким таблицам базы данных.

Для определения данных, которые хранятся в отдельной таблице, применяются специальные классы - сущности (entity). Сущности определяют схему таблицы в базе данных, имя таблицы, ее столбцы и сопутствующюю информацию. Когда репозиторий получает данные базы данных через DAO, то эти данные как раз представляют объекты классов сущностей. Аналогично, когда репозиторию необходимо добавить в базу данных новые данные, он создает объект сущности и затем вызывает методы добавления, определенные в DAO.

В итоге все взаимодействие с базой данных выглядит следующим образом:

1. Репозиторий взаимодействует с базой данных Room для получения объекта базы данных, который, в свою очередь, используется для получения ссылок на объект DAO
2. Репозиторий создает объекты сущностей и настраивает их с данными перед передачей их в DAO
3. Репозиторий вызывает методы DAO, передавая в них объекты сущностей, которые надо добавить и т.д.
4. Когда объект DAO получает результат запроса для передачи в репозиторий, он упаковывает эти результаты в объекты сущностей
5. Объект DAO взаимодействует с базой данных Room для выполнения операций с базой данных и обработки результатов
6. База данных Room обрабатывает все низкоуровневые взаимодействия с базой данных SQLite, отправляя запросы и получая результаты

Добавление Room в проект

Перед началом работы с Room, необходимо настроить конфигурацию проекта и добавить все необходимые зависимости. Первым шагом является добавление плагина ksp и дополнительных библиотек в конфигурацию сборки Gradle. Для этого перейдем в преокте к файлу libs.versions.toml и изменим его следующим образом:

```
[versions]
...................
roomRuntime = "2.6.1"
runtimeLivedata = "1.6.5"
ksp = "1.9.0-1.0.13"

[libraries]
............................
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-
```

```
viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
androidx-room-ktx = { module = "androidx.room:room-ktx", version.ref =
"roomRuntime" }
androidx-room-room-compiler = { module = "androidx.room:room-compiler",
version.ref = "roomRuntime" }
androidx-room-runtime = { module = "androidx.room:room-runtime", version.ref =
"roomRuntime" }
androidx-runtime-livedata = { module = "androidx.compose.runtime:runtime-
livedata", version.ref = "runtimeLivedata" }


[plugins]
....................................
devtoolsKsp = { id = "com.google.devtools.ksp", version.ref = "ksp"}
```

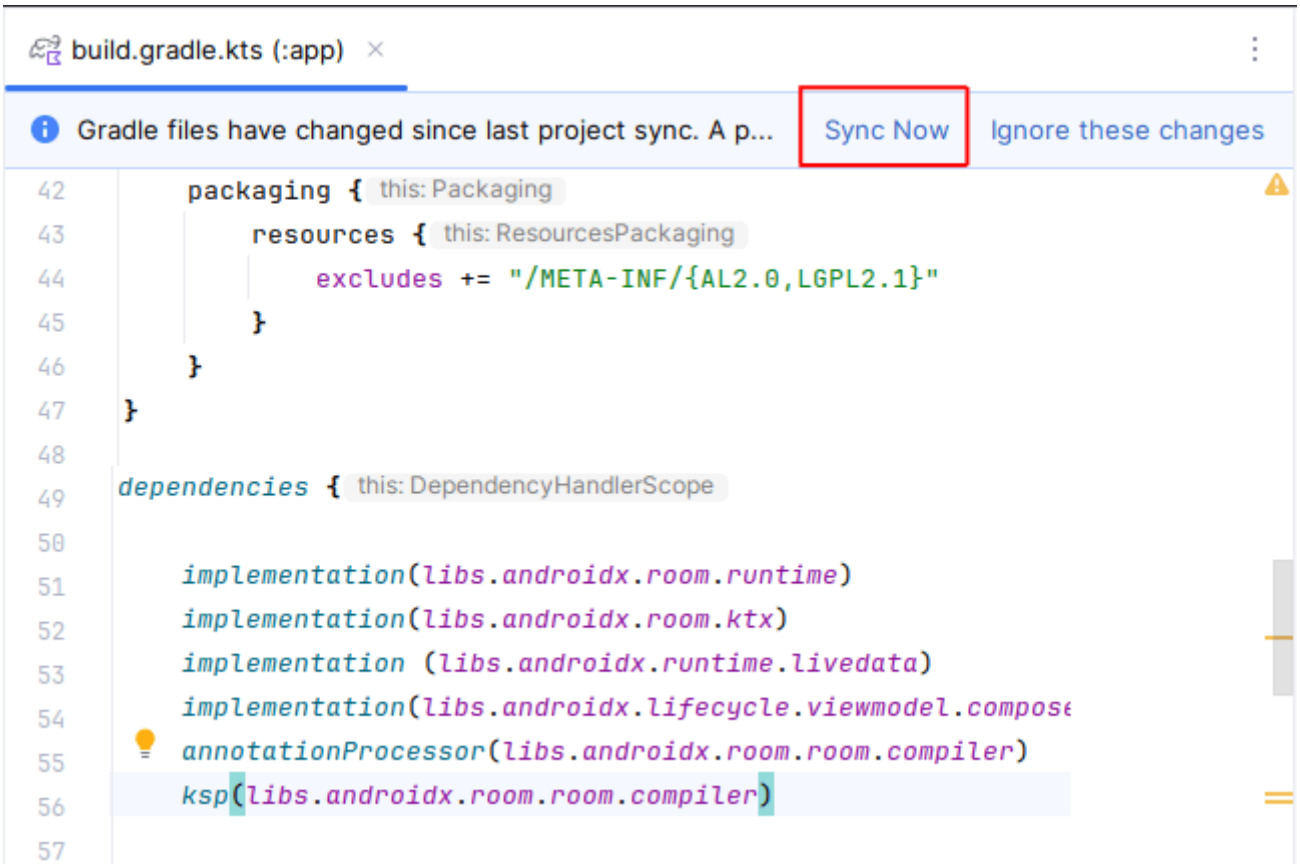
Далее для добавления плагина ksp изменим файл build.gradle.kts уровня проекта ("build.gradle.kts (Project: HelloApp)") следующим образом:

```
// Top-level build file where you can add configuration options common to all sub-
projects/modules.
plugins {
    alias(libs.plugins.androidApplication) apply false
    alias(libs.plugins.jetbrainsKotlinAndroid) apply false
    alias(libs.plugins.devtoolsKsp)
}
```

И далее отредактируем файл build.gradle.kts уровня модуля ("build.gradle.kts (Module :app)") следующим образом:

```
plugins {
    alias(libs.plugins.androidApplication)
    alias(libs.plugins.jetbrainsKotlinAndroid)
    alias(libs.plugins.devtoolsKsp)
}

.....................

dependencies {

    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx)
    implementation (libs.androidx.runtime.livedata)
    implementation(libs.androidx.lifecycle.viewmodel.compose)
    annotationProcessor(libs.androidx.room.room.compiler)
    ksp(libs.androidx.room.room.compiler)

    ....................................
}
```

Затем синхронизируем проект, нажав на кнопку Sync Now



Основные элементы Room

Определение сущностей

Центральноым звеном в организации взаимодействия с базой данных через Room является сущность. С каждой таблицей базы данных связан определенный класс сущности, который определяет схему таблицы. Синтаксически сущность представляет стандартный класса Kotlin с некоторыми специальными аннотациями Room. Например:

```
@Entity(tableName = "users")
class User {
    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "userId")
    var id: Int = 0
    @ColumnInfo(name = "userName")
    var name: String? = null
    var age: Int? = null

    constructor() {}

    constructor(id: Int, name: String, age: Int) {
        this.id = id
        this.name = name
        this.age = age
```

```
    }
    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
}
```

Определение сущности начинается с аннотации @Entity, которая настраивает ряд метаданных для сущности. В частности, параметр tableName указывает, с какой таблицей в базе данных будет сопоставляться эта сущность:

```
@Entity(tableName = "users")
class User {
```

И здесь мы видим, что класс сущности User будет сопоставляться в базе данных с таблицей "users".

Свойства класса сопоставляются с определенными столбцами. По умолчанию сопоставление между свойствами классов и столбцами в таблице выполняется по имени. Например, свойство age не имеет никаких аннотаций:

```
var age: Int? = null
```

Поэтому данные этого свойства будут храниться в таблице базы данных в одноименном столбце "age". Однако с помощью аннотации @ColumnInfo это можно переопределить с помощью параметра name:

```
@ColumnInfo(name = "userName")
var name: String? = null
```

В данном случае мы указываем, что для хранения свойства name в базе данных будет использоваться столбец "userName".

Каждой таблице базы данных необходим столбец, который будет выступать в качестве первичного ключа. Для установки свойства как первичного ключа применяется аннотация @PrimaryKey. В примере выше в качестве такого столбца выступает userId. И с этим столбцом будет сопоставляться свойство id:

```
@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "userId")
var id: Int = 0
```

Кроме того, с помощью параметра autoGenerate = true аннотации @PrimaryKey значение id настроено на автоматическую генерацию. Это значит, что система автоматически сгенерирует идентификатор, присваиваемый новым добавляемым объектам, чтобы избежать дублирования ключей.

Кроме того, к свойству id применяется аннотация @NonNull, которая указывает, что свойство (и соответственно столбец в таблице) не может хранить значение null/NULL.

Если мы не хотим, чтобы данные какого-то столбца хранились в базе данных, то мы можем к соответствующему свойству применить аннотацию @Ignore:

```
@Ignore
var age: Int? = null
```

Определение DAO

Объект доступа к данным или DAO (Data Access Object) предоставляет способ доступа к данным, хранящимся в базе данных SQLite. DAO объявляется как стандартный интерфейс Kotlin с помощью аннотации @Dao:

```
@Dao
interface UserDao {

}
```

Интерфейс DAO может содержать функции, которые с помощью дополнительных аннотаций сопоставляются с конкретными инструкциями SQL. Подобные функции затем может вызывать репозиторий. Например, определим метод getUsers() для получения всех объектов User из таблицы:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getUsers(): LiveData<List<User>>
}
```

В данном случае вызов метода getUsers() приведет к выполнению SQL-выражения "SELECT * FROM users". Этот метод возвращает объект List, который содержит объекты сущности User для каждой строки таблицы. DAO также использует LiveData, чтобы репозиторий мог отслеживать изменения в базе данных.

Методы интерфейсов также могут принимать параметры, на которые затем можно ссылаться в выражениях SQL в качестве аргументов. Например:

```
@Query("SELECT * FROM users WHERE name = :userName")
fun getUser(userName: String): List<User>
```

Данный метод получает через параметр userName имя пользователя и ищет по этому имени всех пользователей. А для вставки значения параметра в SQL-выражение применяется двоеточие :.

Для сопоставления метода интерфейса с SQL-выражением добавления данных может применяться специальная аннотация @Insert:

```
@Insert
fun addUser(user: User)
```

В подобном случае библиотека Room может определить, что сущность User, переданная методу addUser(), должна быть добавлена в базу данных. Причем эта аннотация позволяет добавлять сразу несколько объектов в рамках одной транзакции:

```
@Insert
fun insertUsers(User... userList)
```

Для удаления данных можно определить SQL-инструкцию:

```
@Query("DELETE FROM users WHERE userName = :name")
fun deleteUser(name: String)
```

В качестве альтернативы также можно использовать аннотацию @Delete, в том числе для удаления набора данных:

```
@Delete
fun deleteUsers(User... users)
```

Аналогично для обновления можно использовать аннотацию @Update:

```
@Update
fun updateUsers(User... users)
```

Методы DAO для вышерассмотренных операций также могут возвращать значение типа Int, которое хранит количество строк, затронутых транзакцией, например:

```
@Delete
fun deleteUsers(User... users): int
```

База данных Room

База данных Room представляет слой поверх фактической базы данных SQLite, который отвечает за предоставление доступа к экземплярам DAO, связанным с базой данных. Каждое приложение Android

должно иметь только один экземпляр базы данных Room.

Синтаксически база данных Room представляет класс, который наследуется от RoomDatabase. Например:

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.example.helloapp.User
import com.example.helloapp.UserDao


@Database(entities = [(User::class)], version = 1)
abstract class UserRoomDatabase: RoomDatabase() {

    abstract fun UserDao(): UserDao

    // реализуем синглтон
    companion object {
        private var INSTANCE: UserRoomDatabase? = null
        fun getInstance(context: Context): UserRoomDatabase {

            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        UserRoomDatabase::class.java,
                        "User_database"

                    ).fallbackToDestructiveMigration().build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

К классу базы данных Room применяется аннотация @Database, которая объявляет сущности, с которыми должна работать база данных. В остальном класс содержит фактически реализацию паттерна синглтон, который гарантирует, что одномоментно может быть только один объект этого класса.

Репозиторий

Репозиторий содержит код, который вызывает методы DAO для выполнения операций с базой данных. Пример репозитория:

```
import androidx.lifecycle.LiveData
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class UserRepository(private val userDao: UserDao) {
    private val coroutineScope = CoroutineScope(Dispatchers.Main)

    val userList: LiveData<List<User>>? = userDao.getUsers()

    fun addUser(User: User) {
        coroutineScope.launch(Dispatchers.IO) {
            userDao.addUser(User)
        }
    }

    fun deleteUser(id:Int) {
        coroutineScope.launch(Dispatchers.IO) {
            userDao.deleteUser(id)
        }
    }
}
```

При вызове методов DAO важно отметить, что если метод не возвращает экземпляр LiveData (который автоматически выполняет запросы в отдельном потоке), операцию нельзя выполнить в основном потоке приложения. А при попытке сделать это мы получим следующую ошибку:

```
Cannot access database on the main thread since it may potentially lock the UI for
a long period of time
```

Кроме того, поскольку выполнение некоторых транзакций базы данных может занять много времени, выполнение операций в отдельном потоке позволяет избежать блокировки приложения.

После объявления всех классов необходимо создать и инициализировать экземпляры базы данных, DAO и репозитория, код которых может выглядеть следующим образом:

val userDb = UserRoomDatabase.getInstance(application) val userDao = userDb.UserDao() private val repository: UserRepository = UserRepository(userDao)

База данных в памяти

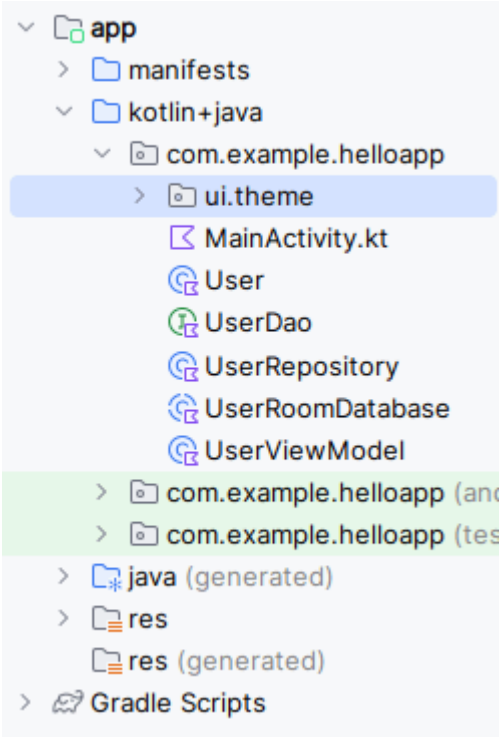
Стоит отметить, что библиотека Room также поддерживает базы данных в памяти. Эти базы данных полностью находятся в памяти и уничтожаются при завершении работы приложения. Единственное изменение, необходимое для работы с базой данных в памяти, — это вызов метода Room.inMemoryDatabaseBuilder() класса Room вместо Room.databaseBuilder(). Например, сравнение двух подходов:

```
// Создание базы данных SQLite
val instance = Room.databaseBuilder(
                   context.applicationContext,
                   UserRoomDatabase::class.java,
                   "users_db"
               ).fallbackToDestructiveMigration()
               .build()

// Создание базы данных в памяти
val instance2 = Room.inMemoryDatabaseBuilder(
                   context.applicationContext,
                   UserRoomDatabase::class.java,
               ).fallbackToDestructiveMigration()
               .build()
```

Обратите внимание, что для базы данных в памяти не требуется имя базы данных.

Пример работы с SQLite и Room

Рассмотрим примитивный практически пример работы с базой данных SQLite через библиотеку Room с использованием ViewModel. Финальный проект будет выглядеть следующим образом:



Пусть в файле User.kt будет располагаться класс User - сущность, с которой мы будем работать:

```
package com.example.helloapp

import androidx.annotation.NonNull
import androidx.room.ColumnInfo
import androidx.room.Entity
```

```
import androidx.room.PrimaryKey

@Entity(tableName = "users")
class User {
    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "userId")
    var id: Int = 0
    @ColumnInfo(name = "userName")
    var name: String=""
    var age: Int = 0

    constructor() {}

    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
}
```

Класс User будет сопоставляться с таблицей "users".

В файле UserDao.kt располагается интерфейс UserDao, который определяет методы для взаимодействия с базой данных:

```
package com.example.helloapp

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query

@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getUsers(): LiveData<List<User>>

    @Insert
    fun addUser(user: User)

    @Query("DELETE FROM users WHERE userId = :id")
    fun deleteUser(id:Int)
}
```

Здесь определены три метода. Метод getUsers() будет выполнять SELECT-запрос и возвращает список всех объектов User из базы данных в виде объекта `LiveData<List<User>>`. Метод addUser() принимает добавляемый объект User и выполняет INSERT-запрос. И метод deleteUser() удаляет объект User из базы данных по id.

В файле UserRoomDatabase.kt располагается класс базы данных Room:

```
package com.example.helloapp

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [(User::class)], version = 1)
abstract class UserRoomDatabase: RoomDatabase() {

    abstract fun userDao(): UserDao

    // реализуем синглтон
    companion object {
        private var INSTANCE: UserRoomDatabase? = null
        fun getInstance(context: Context): UserRoomDatabase {

            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        UserRoomDatabase::class.java,
                        "usersdb"
                    ).fallbackToDestructiveMigration().build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

В данном случае UserRoomDatabase определяет объект-синглтон, поскольку в приложении база данных Room должна существовать в одном в единственном виде. В качестве имени базы данных устанавливается "usersdb".

В файле UserRepository.kt расположен класс репозитория, через который приложение будет взаимодействовать с базой данных через объект UserDao:

```
package com.example.helloapp

import androidx.lifecycle.LiveData
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class UserRepository(private val userDao: UserDao) {
    private val coroutineScope = CoroutineScope(Dispatchers.Main)
```

```
    val userList: LiveData<List<User>> = userDao.getUsers()

    fun addUser(User: User) {
        coroutineScope.launch(Dispatchers.IO) {
            userDao.addUser(User)
        }
    }

    fun deleteUser(id:Int) {
        coroutineScope.launch(Dispatchers.IO) {
            userDao.deleteUser(id)
        }
    }
}
```

Класс репозитория определяет ряд методов, которые обращаются к методам объекта UserDao, который передается через конструктор. Для этого используются корутины, чтобы избежать выполнения операций с базой данных в основном потоке.

Отдельно стоит сказать про свойство userList, которое хранит список всех объектов из БД. Для его получения вызывается метод userDao.getUsers(). Причем репозиторию необходимо вызвать этот метод один раз при инициализации и сохранить результат в объекте LiveData, который может наблюдаться ViewModel и, в свою очередь, объектом Activity. После этого каждый раз, когда в таблице базы данных будет происходить изменение, компонент, который отслеживает изменения, получит уведомление об изменениях и будет перекомпонован с использованием обновленного списка userList.

В файле UserViewModel.kt расположен класс UserViewModel - модель представления, который будет выполнять роль посредника между репозиторием и графическим интерфейсом:

```
package com.example.helloapp

import android.app.Application
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.lifecycle.LiveData
import androidx.lifecycle.ViewModel

class UserViewModel(application: Application) : ViewModel() {

    val userList: LiveData<List<User>>
    private val repository: UserRepository
    var userName by mutableStateOf("")
    var userAge by mutableStateOf(0)

    init {
        val userDb = UserRoomDatabase.getInstance(application)
        val userDao = userDb.userDao()
        repository = UserRepository(userDao)
        userList = repository.userList
```

```
    }
    fun changeName(value: String){
        userName = value
    }
    fun changeAge(value: String){
        userAge = value.toIntOrNull()?:userAge
    }
    fun addUser() {
        repository.addUser(User(userName, userAge))
    }
    fun deleteUser(id: Int) {
        repository.deleteUser(id)
    }
}
```

Класс ViewModel принимает экземпляр контекста приложения, который представлен классом Android Context и который используется в коде приложения для получения доступа к ресурсам приложения во время выполнения. Кроме того, в контексте приложения можно вызывать широкий спектр методов для сбора информации и внесения изменений в среду приложения. В нашем случае контекст приложения необходим при создании базы данных.

ViewModel определяет ряд переменных.

```
val userList: LiveData<List<User>>
private val repository: UserRepository
var userName by mutableStateOf("")
var userAge by mutableStateOf(0)
```

userList прелставляет список пользователей, полученный из базы данных. для взаимодействия с БД определяется переменная репозитория - repository. И для управления вводом новых данных для имени и возраста пользователя определяются две переменных состояния - userName и userAge.

Блок инициализатора создает базу данных, которая используется для создания объекта UserDao. Затем мы используем UserDao для инициализации репозитория и получения данных в userList:

```
init {
    val userDb = UserRoomDatabase.getInstance(application)
    val userDao = userDb.userDao()
    repository = UserRepository(userDao)
    userList = repository.userList
}
```

И также UserViewModel определяет ряд методов, которые будут вызываться при изменении ввода в текстовом поле или при нажатии на кнопку.

И наконец определим сам интерфейс в MainActivity.kt:

```
package com.example.helloapp

import android.app.Application
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material3.Button
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.getValue
import androidx.compose.ui.unit.sp
import androidx.lifecycle.ViewModel

import androidx.compose.runtime.Composable
import androidx.compose.runtime.livedata.observeAsState
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.tooling.preview.Preview
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.viewmodel.compose.LocalViewModelStoreOwner
import androidx.lifecycle.viewmodel.compose.viewModel


class UserViewModelFactory(val application: Application) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return UserViewModel(application) as T
    }
}

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val owner = LocalViewModelStoreOwner.current

            owner?.let {
                val viewModel: UserViewModel = viewModel(
                    it,
```

```
                    "UserViewModel",
                    UserViewModelFactory(LocalContext.current.applicationContext
as Application)
                )
                Main(viewModel)
            }
        }
    }
}


@Composable
fun Main(vm: UserViewModel = viewModel()) {
    val userList by vm.userList.observeAsState(listOf())
    Column {
        OutlinedTextField(vm.userName, modifier= Modifier.padding(8.dp), label = {
Text("Name") }, onValueChange = {vm.changeName(it)})
        OutlinedTextField(vm.userAge.toString(), modifier= Modifier.padding(8.dp),
label = { Text("Age") },
            onValueChange = {vm.changeAge(it)},
            keyboardOptions = KeyboardOptions(keyboardType= KeyboardType.Number)
        )
        Button({ vm.addUser() }, Modifier.padding(8.dp)) {Text("Add", fontSize =
22.sp)}
        UserList(users = userList, delete = {vm.deleteUser(it)})
    }
}
@Composable
fun UserList(users:List<User>, delete:(Int)->Unit) {
    LazyColumn(Modifier.fillMaxWidth()) {
        item{ UserTitleRow()}
        items(users) {u -> UserRow(u, {delete(u.id)})  }
    }
}
@Composable
fun UserRow(user:User, delete:(Int)->Unit) {
    Row(Modifier .fillMaxWidth().padding(5.dp)) {
        Text(user.id.toString(), Modifier.weight(0.1f), fontSize = 22.sp)
        Text(user.name, Modifier.weight(0.2f), fontSize = 22.sp)
        Text(user.age.toString(), Modifier.weight(0.2f), fontSize = 22.sp)
        Text("Delete", Modifier.weight(0.2f).clickable { delete(user.id) },
color=Color(0xFF6650a4), fontSize = 22.sp)
    }
}
@Composable
fun UserTitleRow() {
    Row(Modifier.background(Color.LightGray).fillMaxWidth().padding(5.dp)) {
        Text("Id", color = Color.White,modifier = Modifier.weight(0.1f), fontSize
= 22.sp)
        Text("Name", color = Color.White,modifier = Modifier.weight(0.2f),
fontSize = 22.sp)
        Text("Age", color = Color.White, modifier = Modifier.weight(0.2f),
fontSize = 22.sp)
        Spacer(Modifier.weight(0.2f))
```

```
    }
}
```

Для создания модели представления UserViewModel ей необходимо передать ссылку на объект Application (контекст приложения). Стандартная функция viewModel(), которая обычно применяется для создания моделей представления, не позволяет этого сделать. Поэтому вместо использования viewModel() мы создаем свой собственный класс ViewModelProvider.Factory, предназначенный для передачи ссылки на объект Application и возврата объекта UserViewModel.

```
class UserViewModelFactory(val application: Application) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return UserViewModel(application) as T
    }
}
```

Кроме фабрики типа ViewModelProvider.Factory функция viewModel() также требует ссылки на текущий объект ViewModelStoreOwner. Этот объект представляет своего рода контейнер, в котором хранятся все активные в данный момент модели представления ViewModel. Используя название ViewModel (в нашем случае "UserViewModel"), можно получить нужную модель представления:

```
setContent {
    val owner = LocalViewModelStoreOwner.current

    owner?.let {
    val viewModel: UserViewModel = viewModel(
        it,
        "UserViewModel",
        UserViewModelFactory(LocalContext.current.applicationContext as
Application)
        )
    Main(viewModel)
    }
}
```

Здесь вначале получаем текущий объект ViewModelStoreOwner, используя свойство LocalViewModelStoreOwner.current. Провереяем полученный объект на null и вызываем функцию viewModel(), в которую передаем объект ViewModelStoreOwner (it), идентифицирующую строку модели представления ("UserViewModel") и фабрику модели представления UserViewModelFactory (которой передается ссылка на контекст приложения).

Получив объект ViewModel, передаем его в компонент Main, который определяет интерфейс приложения:

```
@Composable
fun Main(vm: UserViewModel = viewModel()) {
    val userList by vm.userList.observeAsState(listOf())
    Column {
        OutlinedTextField(vm.userName, modifier= Modifier.padding(8.dp), label = {
Text("Name") }, onValueChange = {vm.changeName(it)})
        OutlinedTextField(vm.userAge.toString(), modifier= Modifier.padding(8.dp),
label = { Text("Age") },
            onValueChange = {vm.changeAge(it)},
            keyboardOptions = KeyboardOptions(keyboardType= KeyboardType.Number)
        )
        Button({ vm.addUser() }, Modifier.padding(8.dp)) {Text("Add", fontSize =
22.sp)}
        UserList(users = userList, delete = {vm.deleteUser(it)})
    }
}
```

Компонент получает из модули представления список пользователей в переменную userList. Причем благодаря получению с помощью функции vm.userList.observeAsState() компонент будет отслеживать все изменения в списке, и если в список будет добавлены новые объекты или из него будут удалены ранее существовавшие, то компонент будет обновлен, чтобы отразить эти изменения.

Внутри компонента определен простейший интерфейс. Прежде всего это два текстовых поля для ввода данных для имени и возраста пользователя, при изменении которых срабатывают функции changeName и changeAge из UserViewModel. Также определена кнопка, при нажатии на которую вызывается функция addUser, которая добавляет нового пользователя.

И также вызывается кастомный компонент UserList, который выводит список пользователей:

```
@Composable
fun UserList(users:List<User>, delete:(Int)->Unit) {
    LazyColumn(Modifier.fillMaxWidth()) {
        item{ UserTitleRow()}
        items(users) {u -> UserRow(u, {delete(u.id)})  }
    }
}
```

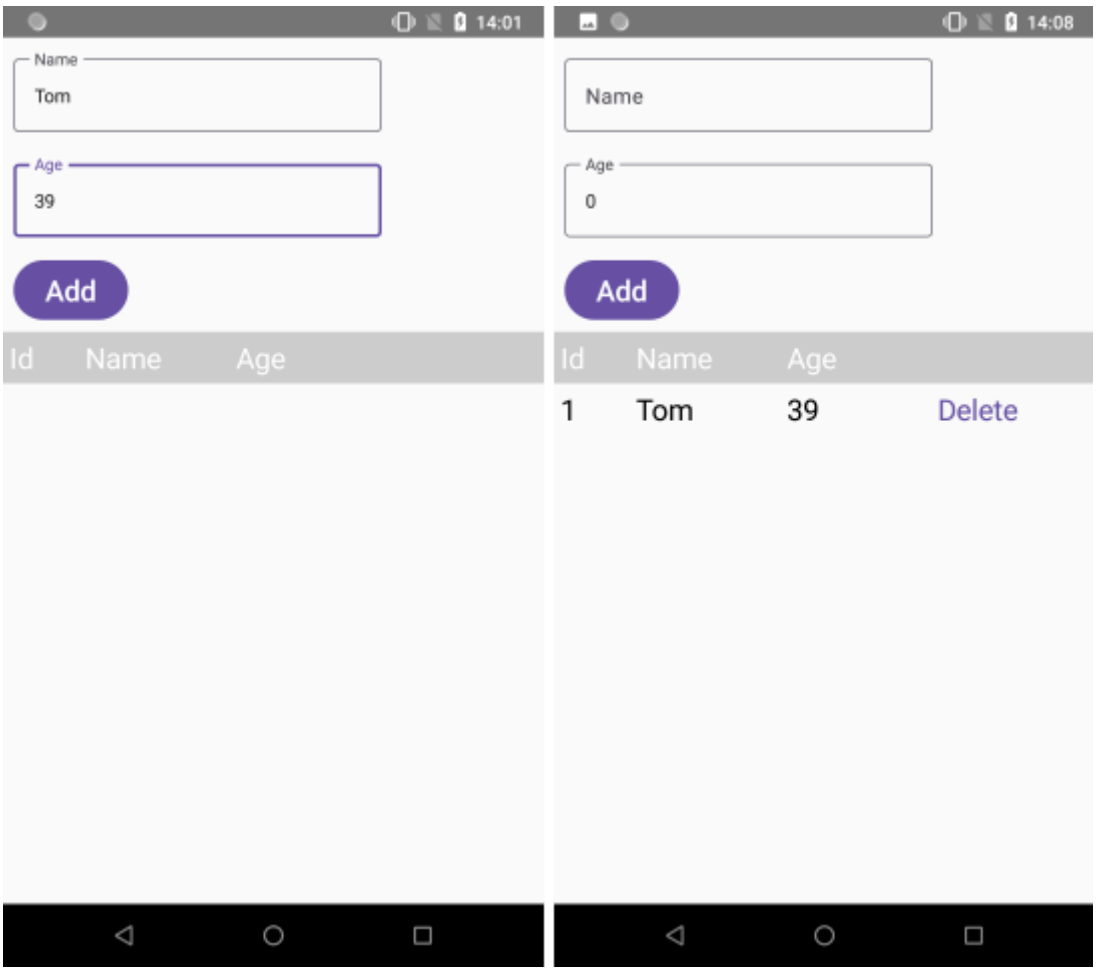
Для вывода списка применяется LazyColumn, который для отображения заголовка использует кастомный компонент UserTitleRow, а для вывода данных каждого пользователя - UserRow.

```
@Composable
fun UserRow(user:User, delete:(Int)->Unit) {
    Row(Modifier .fillMaxWidth().padding(5.dp)) {
        Text(user.id.toString(), Modifier.weight(0.1f), fontSize = 22.sp)
        Text(user.name, Modifier.weight(0.2f), fontSize = 22.sp)
        Text(user.age.toString(), Modifier.weight(0.2f), fontSize = 22.sp)
        Text("Delete", Modifier.weight(0.2f).clickable { delete(user.id) },
color=Color(0xFF6650a4), fontSize = 22.sp)
```

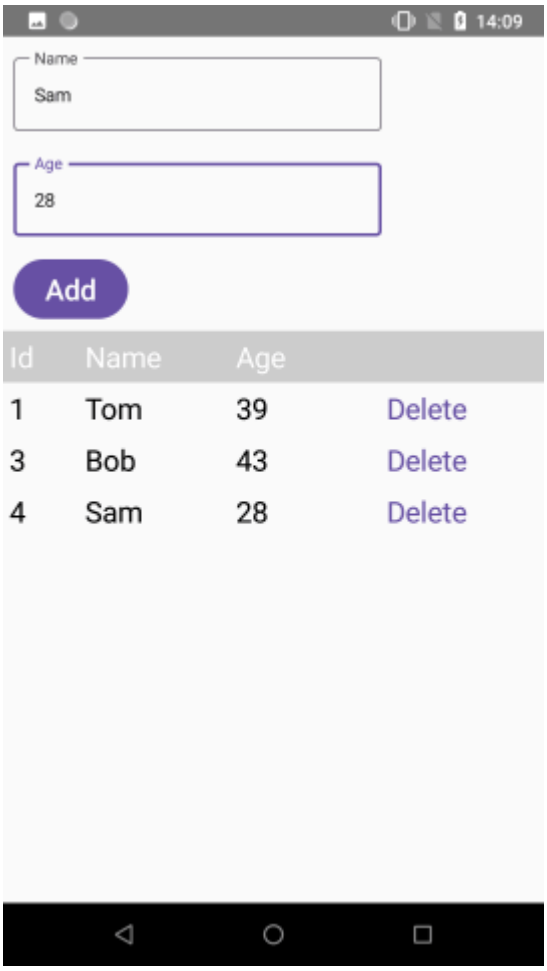
```
        }
    }
```

Здесь выводятся все свойства объекта User, а последнbq компонент Text выстпает в качестве кнопки, по нажатию на которую срабаывает метод deleteUser в UserViewModel.

Запустим приложение и добавим объект в базу данных:



Причем данные автоматически отобразяться в списке объектов. Аналогичным образом можно добавить больше объектов или удалять их из базы данных:


14:09
Name
Sam
Age
28
Add
Id
Name
Age
1
Tom
39
Delete
3
Bob
43
Delete
4
Sam
28
Delete