МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Лабораторная работа № 3**

*по дисциплине «Программирование мобильных устройств»*

Выполнил студен группы ПИбд-32

Емельянов А. С.

Проверил доцент кафедры

«Информационные системы»

Филиппов А.А.

Ульяновск, 2023

**Задание** Хранение данных (персистентность). Необходимо: 1. Создать необходимые для работы приложения классы-сущности (@Entity). Минимум 2 сущности. Между сущностями должны быть связи (один-к-одному, один-комногим или многие-ко-многим). Обязательно должна быть сущность User для последующей реализации функции аутентификации. 2. Разработать DAO-интерфейсы (@Dao) с необходимым набором методов для создания, изменения, удаления и получения сущностей. 3. Разработать класс для работы с БД с помощью Room (RoomDatabase). Получение экземпляра данного класса должно быть реализовано с помощью шаблона проектирования Singleton. 4. Добавить callback для заполнения БД тестовым набором данных. Данные можно представить в виде db-файла, json-файла или написать программный код для заполнения БД. 5. Android не позволяет запускать методы, которые обращаются к БД, в основном (mainUI) потоке. Необходимо использовать методы асинхронного выполнения запросов к БД (rxjava, coroutines). 6. Во все экранные формы (см. ЛР №2) добавить логику для получения тестовых данных из БД вместо обращения к подготовленным спискам, массивам и т. д. Необходимо учитывать асинхронную природу соответствующих методов работы с БД. 7. Отчет и изменения проекта загрузить в репозиторий по адресу http://student.git.athene.tech

**Решение**

1. Создадим необходимые модели для работы
2. @Entity  
   data class Sneaker(  
    @PrimaryKey(autoGenerate = true)  
    val sneakerId: Int? = null,  
    @ColumnInfo(name = "Brand")  
    val brand: String,  
    @ColumnInfo(name = "Model")  
    val model: String,  
    @ColumnInfo(name = "Description")  
    val description: String,  
    @ColumnInfo(name = "Price")  
    val price: Double,  
    @ColumnInfo(name = "Photo")  
    val photo: Int  
   )

@Entity(tableName = "users")  
data class User(  
 @PrimaryKey(autoGenerate = true)  
 val userId: Int? = null,  
 @ColumnInfo(name = "Name")  
 val name: String,  
 @ColumnInfo(name = "Surname")  
 val surname: String,  
 @ColumnInfo(name = "Email")  
 val email: String,  
 @ColumnInfo(name = "Password")  
 val password: String,  
 @ColumnInfo(name = "Role")  
 val role: RoleEnum,  
 @ColumnInfo(name = "Photo")  
 val photo: Int? = null,  
)

@Entity  
data class Order(  
 @PrimaryKey(autoGenerate = true)  
 val orderId: Int? = null,  
 @ColumnInfo(name = "Date")  
 val date: Long,  
 @ColumnInfo(name = "City")  
 val city: String,  
 @ColumnInfo(name = "Street")  
 val street: String,  
 @ColumnInfo(name = "House")  
 val house: String,  
 @ColumnInfo(name = "Sub Total")  
 val subtotal: Double,  
 @ColumnInfo(name = "Taxes")  
 val taxes: Double,  
 @ColumnInfo(name = "Total")  
 val total: Double,  
 @ColumnInfo(name = "CreatorUserId")  
 val creatorUserId: Int  
)

Сделаем связи между сущностями

@Entity(primaryKeys = ["orderId", "sneakerId"])  
data class OrderSneaker(  
 val orderId: Int,  
 val sneakerId: Int  
)

data class OrderWithSneakers(  
 @Embedded val order: Order,  
 @Relation(  
 parentColumn = "orderId",  
 entityColumn = "sneakerId",  
 associateBy = Junction(OrderSneaker::class)  
 )  
 val sneakers: List<Sneaker>  
)

data class UserWithOrder(  
 @Embedded val user: User,  
 @Relation(  
 parentColumn = "userId",  
 entityColumn = "CreatorUserId"  
 )  
 val orders: List<Order>  
)

2. Создадим Dao интерфейсы для взаимодействия с сущностями

@Dao  
interface SneakerDao {  
 @Insert  
 suspend fun insert(sneaker: Sneaker)  
  
 @Update  
 suspend fun update(sneaker: Sneaker)  
  
 @Delete  
 suspend fun delete(sneaker: Sneaker)  
  
 @Query("SELECT\*FROM Sneaker")  
 fun getAllSneakers(): Flow<List<Sneaker>>  
  
 @Query("SELECT \* FROM Sneaker WHERE sneakerId = :id")  
 suspend fun getSneakerById(id: Int): Sneaker  
}

@Dao  
interface OrderDao {  
  
 @Insert  
 suspend fun createOrder(order: Order): Long  
  
 @Insert  
 suspend fun insertOrderSneaker(orderSneaker: OrderSneaker)  
  
 @Query("SELECT \* FROM 'Order' WHERE orderId = :id")  
 fun getOrderWithSneakers(id: Int): Flow<OrderWithSneakers>  
  
 @Query("SELECT \* FROM `Order`")  
 fun getAllOrder(): Flow<List<Order>>  
  
 @Delete  
 suspend fun delete(order: Order)  
}

@Dao  
interface UserDao {  
 @Insert  
 suspend fun createUser(user: User)  
  
 @Update  
 suspend fun updateUser(user: User)  
  
 @Delete  
 suspend fun deleteUser(user: User)  
  
 @Query("SELECT \* FROM users WHERE userId = :id")  
 suspend fun getUserById(id: Int): User  
  
 @Query("SELECT \* FROM users WHERE email = :email")  
 suspend fun getUserByEmail(email: String): User  
  
 @Query("SELECT \* FROM users WHERE userId =:id")  
 fun getUserOrders(id: Int) : Flow<UserWithOrder>  
}

1. Разработаем класс для работы с БД
2. @Database(entities = [Sneaker::class, User::class, Order::class, OrderSneaker::class], version = 5)  
   abstract class AppDatabase : RoomDatabase() {  
    abstract fun sneakerDao(): SneakerDao  
    abstract fun userDao(): UserDao  
    abstract fun orderDao(): OrderDao  
     
    companion object {  
    private const val DB\_NAME: String = "my-db"  
     
    @Volatile  
    private var INSTANCE: AppDatabase? = null  
     
    suspend fun populateDatabase() {  
    INSTANCE?.*let* **{** database **->** // User  
    val userDao = database.userDao()  
    val user1 = User(null, "Artem", "Emelyanov", "artem@mail.ru", "123", RoleEnum.*Admin*)  
    val user2 = User(null, "Danil", "Markov", "danil@mail.ru", "123", RoleEnum.*User*)  
    val user3 = User(null, "Viktoria", "Presnyakova", "vika@mail.ru", "123", RoleEnum.*User*)  
    userDao.createUser(user1)  
    userDao.createUser(user2)  
    userDao.createUser(user3)  
    // Sneaker  
    val sneakerDao = database.sneakerDao()  
    val sneaker1 = Sneaker(null, "Nike", "Air Force 1", "nice", 159.99, R.drawable.*img\_1*)  
    val sneaker2 = Sneaker(null, "Adidas", "ZX 750", "beautiful", 169.99, R.drawable.*img\_2*)  
    val sneaker3 = Sneaker(null, "Reebok", "Classic", "amazing", 179.99, R.drawable.*img\_3*)  
    val sneaker4 = Sneaker(null, "Puma", "Classic", "normal", 189.99, R.drawable.*img\_4*)  
    sneakerDao.insert(sneaker1)  
    sneakerDao.insert(sneaker2)  
    sneakerDao.insert(sneaker3)  
    sneakerDao.insert(sneaker4)  
    // Order  
    **}** }  
     
    fun getInstance(appContext: Context): AppDatabase {  
    return INSTANCE ?: *synchronized*(this) **{** Room.databaseBuilder(  
    appContext,  
    AppDatabase::class.*java*,  
    DB\_NAME  
    )  
    .addCallback(object : Callback() {  
    override fun onCreate(db: SupportSQLiteDatabase) {  
    super.onCreate(db)  
    *CoroutineScope*(Dispatchers.IO).*launch* **{** populateDatabase()  
    **}** }  
    })  
    .fallbackToDestructiveMigration()  
    .build()  
    .*also* **{** INSTANCE = **it }  
    }** }  
    }  
   }
3. Так же тут присутсвует метод populateDatabase заполнения тестовыми данными
4. Для вызова методов получения подготовленных списков используем функции, которые позволяют работать с асинхронными операциями

val basketList = remember { mutableStateMapOf<Sneaker, Int>() }

val context = LocalContext.current

LaunchedEffect(Unit) {

withContext(Dispatchers.IO) {

AppDatabase.getInstance(context).basketDao().getBasketWithSneakers(1).collect { data ->

basketList.clear()

for (item in data.services){

basketList[item] = (basketList[item] ?: 0) + 1

}

}

}

}