Практическая работа 14. Room

В большинстве приложений используются данные. Но если вы не позаботитесь о том, чтобы эти данные были где-то сохранены, данные будут навсегда потеряны при закрытии приложения. В мире приложений Android информация обычно хранится в базах данных, поэтому в этой практической работе мы представим библиотеку хранения данных Room. Вы научитесь строить базы данных, создавать таблицы и определять методы доступа к данным с использованием аннотаций классов и интерфейсов. Вы узнаете, как использовать сопрограммы для выполнения кода баз данных в фоновом режиме. Заодно вы научитесь преобразовывать данные `Live Data` при их изменении с помощью `Transformations.map()`.

Многим приложениям требуется хранить данные Почти во всех приложениях, написанных нами ранее, использовались небольшие объемы статических данных. Например, в приложении `Guessing Game`, в модели представления хранился статический массив строк, из которого игра случайным образом выбирала загаданное слово. Однако в реальном мире приложения обычно не ограничиваются статическими данными; они должны иметь возможность сохранять данные, которые могут изменяться, чтобы изменения не терялись при закрытии приложения. Например, приложение-проигрыватель может хранить списки воспроизведения, а игра может хранить информацию о текущем состоянии игрока, чтобы при очередном запуске он мог продолжить игру с того места, где она была остановлена. Приложения могут хранить информацию в базе данных Как правило, данные пользователя удобнее всего хранить в базе данных, поэтому в этой практической работе вы научитесь работать с базой данных на примере приложения `Tasks`. В этом приложении пользователь может добавлять задачи в базу данных и выводить список всех уже введенных задач. Приложение будет выглядеть так:

Прежде чем начать строить приложения, стоит разобраться в его структуре.

Структура приложения

Приложение содержит одну активность (с именем `MainActivity`), которая будет использоваться для отображения фрагмента с именем `TasksFragment`.

`TasksFragment` представляет главный экран приложения. Его файл макета (`fragment_tasks.xml`) включает текстовое поле и кнопку, которая позволяет пользователю ввести имя задачи и вставить ее в базу данных. В нем также присутствует текстовое представление, в котором выводятся все задачи, сохраненные в базе данных:

Мы также добавим в приложение модель представления (с именем `TasksViewModel`), которая будет использоваться фрагментом `TasksFragment` для бизнес-логики. Она включает свойства и методы, которые используются фрагментом для взаимодействия с базой данных приложения. Мы также включим связывание данных, чтобы макет `TasksFragment` мог напрямую обращаться к модели представления.

Эти компоненты будут взаимодействовать по следующей схеме:

Для создания базы данных будет использоваться библиотека `Android Room`. Что же собой представляет `Room`?

Room — библиотека баз данных, работающая на базе SQLite

Во внутренней реализации многие базы данных Android используют `SQLite`. `SQLite` — упрощенная, стабильная, быстрая и оптимизированная для однопользовательского доступа система, и все эти особенности делают ее хорошим кандидатом для приложений Android. Тем не менее написание кода для создания, управления и взаимодействия с базами данных `SQLite` может быть непростым делом. Для упрощения задачи в `Android Jetpack` включена библиотека хранения данных Room, которая работает на базе `SQLite`. С `Room` вы пользуетесь всеми преимуществами `SQLite`, но с более простым кодом. Например, библиотека поддерживает удобные аннотации, которые позволяют быстро писать код баз данных, менее однообразный и более надежный.

Приложения `Room` обычно используют структуру `MVVM` Приложения, использующие Room, включая приложение `Tasks`, обычно используют структуру архитектурного паттерна проектирования, который называется `MVVM` (сокращение от `Model-View-ViewModel`). Эта структура выглядит так:

Представление Включает весь код, отвечающий за пользовательский интерфейс: код активностей, фрагментов и макетов.

Модель представления Объекты моделей представлений отвечают за бизнес-логику и данные каждого представления.

Модель Данные, лежащие в основе приложения. В данном случае это база данных, но в других приложениях это может быть удаленный источник данных.

Эта структура похожа на ту, которая используется в построенном нами приложении `Guessing Game`, не считая того, что в структуре присутствует дополнительный уровень модели для базы данных. Это означает, что UI-код активностей и фрагментов четко отделяется от бизнес-логики, содержащейся в модели представления, а модель представления отделяется от остального кода, обеспечивающего работу базы данных. О том, как использовать структуру MVVM, вы узнаете в процессе построения приложения `Tasks`.

Что мы собираемся сделать

Основные этапы построения приложения `Tasks`:

1. Подготовка базового приложения. Мы создадим файлы приложения, обновим его файлы `build.gradle`, чтобы они использовали необходимые библиотеки, а также напишем базовый код активности, фрагмента и макета.
2. Написание кода базы данных. На этом этапе мы добавим код создания базы данных с таблицей, а также методы доступа к данным, необходимые для взаимодействия с данными таблицы.
3. Вставка записей задач. Мы создадим модель представления и обновим фрагмент приложения, чтобы приложение могло использоваться для вставки записей.
4. Вывод списка записей с описаниями задач. Наконец, мы обновим код модели представления и фрагмента, чтобы приложение выводило список всех записей с описаниями задач, хранящимися в базе данных.

Создание проекта Tasks

Для приложения `Tasks` будет создан новый проект по схеме. Выберите вариант `Empty Activity`, введите имя «Tasks» и имя пакета «com.hfad.tasks», подтвердите папку сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 29, чтобы приложение работало на большинстве устройств Android. Затем мы обновим файлы `build.gradle` проекта, чтобы они включали все возможности и зависимости, необходимые для приложения.

Добавление переменной в файл build.gradle проекта...

В этой главе будут использоваться две библиотеки `Room`, поэтому в файл `build.gradle` проекта будут добавлены новые переменные для используемой версии. Откройте файл *Tasks/build.gradle* и добавьте в раздел `buildscript`:

```
buildscript {
 ext.lifecycle_version = "2.3.1"
 ext.room_version = "2.3.0"
 ...
}
```

В файле `build.gradle` приложения необходимо включить связывание данных и добавить зависимости для модели представления, `Live Data` и библиотек `Room`.

Откройте файл *Tasks/app/build.gradle* и включите следующие строки в соответствующие разделы:

```
plugins {
 ...
 id 'kotlin-kapt'
}

android {
 ...
 buildFeatures {
 viewBinding true
 }
}

dependencies {
 ...
 implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
 implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
 implementation "androidx.room:room-runtime:$room_version"
 implementation "androidx.room:room-ktx:$room_version"
 kapt "androidx.room:room-compiler:$room_version"
 ...
}
```

Затем щелкните на ссылке `Sync Now`, чтобы синхронизировать внесенные изменения с остальными частями проекта.

Создание TasksFragment

Приложение включает один фрагмент с именем `TasksFragment`, который будет использоваться для вывода списка всех задач из базы данных и вставки новых задач.

Чтобы создать `TasksFragment`, выделите файл *com.hfad.tasks* в папке *app/src/main/java* и выберите команду `File→New→Fragment→Fragment (Blank)`. Введите имя фрагмента «TasksFragment» и имя макета «fragment_tasks» и убедитесь в том, что выбран язык Kotlin.

Обновление TasksFragment.kt

После того как фрагмент `TasksFragment` будет добавлен в проект, убедитесь в том, что содержимое `TasksFragment.kt` совпадает с приведенным ниже

```
package com.example.android.roomapp

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.example.android.roomapp.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState:
Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Обновление fragment_tasks.xml

Также необходимо обновить макет `TasksFragment`, чтобы он использовал связывание данных, и включить представления для ввода новых и вывода списка существующих задач.

Откройте файл `fragment_tasks.xml` и включите в него следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" tools:context=".TasksFragment">
    <data>
    </data>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name" />
        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task" />
        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</layout>
```

Отображение TasksFragment в макете MainActivity's...

Чтобы использовать фрагмент `TasksFragment`, необходимо включить его в макет `MainActivity` в `FragmentContainerView`.

Обновите файл `activity_main.xml`, чтобы он выглядел так:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:id="@+id/fragment_container_view"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:padding="16dp"
 android:name="com.hfad.tasks.TasksFragment"
 tools:context=".MainActivity" />
```

и проверка кода `MainActivity.kt` После обновления макета откройте `MainActivity.kt` и убедитесь в том, что он содержит следующий код:

```
package com.hfad.tasks
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
class MainActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)
 }
}
```

Код фрагмента и активности обновлен, можно переходить к работе с базой данных `Room`.

Как создаются базы данных Room

`Room` использует набор интерфейсов и классов с аннотациями для создания баз данных `SQLite` для вашего приложения.

Для этого необходимы три условия:

1. Класс базы данных. Класс определяет базу данных, включая ее имя и номер версии. Он используется для получения экземпляра базы данных.

2. Вся информация в базе данных хранится в таблицах. Каждая таблица определяется при помощи класса данных, который включает аннотации для имени таблицы и ее столбцов.

3. Интерфейсы доступа к данным. Для взаимодействия с таблицей используется интерфейс, который определяет методы доступа к данным, необходимые для вашего приложения. Например, если вам потребуется выполнять вставку записей, в интерфейс включен метод `insert()`, а для получения всех записей добавляется метод `getAll()`.

`Room` использует эти три составляющие для генерирования всего кода, необходимого приложению для создания базы данных SQLite, ее таблиц и методов доступа к данным.

На нескольких ближайших страницах мы покажем, как написать код этих трех компонентов, на примере определения базы данных для приложения `Tasks`. Начнем с определения таблиц.

Данные задач будут храниться в таблице

Как говорилось ранее, вся информация базы данных хранится в одной или нескольких таблицах. Каждая таблица состоит из строк и столбцов; каждая строка соответствует записи, а каждый столбец содержит один блок данных (например, число или текст).

Для каждого типа данных, который должен храниться в базе данных, создается отдельная таблица. Например, в календарном приложении может присутствовать таблица для хранения событий, а в приложении для прогноза погоды — таблица для хранения географических областей. В приложении

Tasks должны храниться записи с описаниями задач; мы создадим таблицу с именем «task_table». Таблица будет выглядеть примерно так:

taskId	task_name	task_name
1	Walk dog	false
2	Book vacation	false
3	Arrange picnic	false

Таблицы определяются классом данных с аннотациями

Для каждой таблицы, которая должна быть включена в базу данных, определяется класс данных. Этот класс данных должен включать свойство для каждого столбца таблицы, а аннотации сообщают Room, как должна быть настроена конфигурация таблицы. Чтобы вы поняли, как это делается, мы определим класс данных с именем Task, который будет использоваться для создания таблицы в базе данных приложения Tasks.

Создание класса данных Task

Начнем с создания класса данных. Выделите пакет *com.hfad.tasks* в папке *app/src/main/java*, после чего выберите команду File→New→Kotlin Class/File. Введите имя файла «Task» и выберите вариант Class.

После того как файл будет создан, включите в него следующий код:

```
package com.hfad.tasks
data class Task(
 var taskId: Long = 0L,
 var taskName: String = "",
 var taskDone: Boolean = false
)
```

Определение имени таблицы аннотацией @Entity

После создания класса данных Task необходимо добавить аннотации, которые сообщают Room, как должна определяться конфигурация таблицы. Начнем с имени таблицы. Чтобы задать имя таблицы, добавьте в класс данных аннотацию @Entity с именем таблицы. Таблице из приложения Tasks будет присвоено имя «task_table», а соответствующий код выглядит так:

```
@Entity(tableName = "task_table")
data class Task(
```

Первичный ключ задается аннотацией @PrimaryKey

Затем необходимо задать первичный ключ таблицы. Он используется для однозначной идентификации отдельной записи и не может принимать повторяющиеся значения. В приложении Tasks для

первичного ключа таблицы `task_table` будет использоваться свойство `taskId`, а таблица будет автоматически генерировать его значения, чтобы они были уникальными. Для этого используется аннотация @PrimaryKey:

```
@PrimaryKey(autoGenerate = true)
 var taskId: Long = 0L,
```

Определение имен столбцов аннотацией @ColumnInfo

Остается задать имена столбцов для свойств `taskName` и `taskDone`. Для этого используется аннотация `@ColumnInfo`:

```
@ColumnInfo(name = "task_name")
 var taskName: String = "",
 @ColumnInfo(name = "task_done")
 var taskDone: Boolean = false
```

Учтите, что аннотация `@ColumnInfo` необходима только в том случае, если имя столбца должно отличаться от имени свойства. Если опустить эту аннотацию, `Room` присвоит столбцу имя, совпадающее с именем столбца. И это все, что необходимо знать для создания класса данных `Task`. Полный код приводится на следующей странице.

Полный код Task.kt

Ниже приведен полный код класса данных `Task`; обновите файл `Task.kt`:

```
package com.hfad.tasks
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey
@Entity(tableName = "task_table")
data class Task(
 @PrimaryKey(autoGenerate = true)
 var taskId: Long = 0L,
 @ColumnInfo(name = "task_name")
 var taskName: String = "",
 @ColumnInfo(name = "task_done")
 var taskDone: Boolean = false
)
```

`Room` использует этот файл для создания таблицы с именем `task_table` с автоматически генерируемым первичным ключом `taskId` и двумя дополнительными столбцами с именами `task_name` и `task_done`. Таблица выглядит примерно так:

taskId	task_name	task_done

taskId	task_name	task_done

Вы увидите, как Room добавляет эту таблицу в базу данных, через несколько страниц, когда мы напишем класс базы данных. Но сначала мы определим методы доступа к базе данных, чтобы приложение могло вставлять, читать, обновлять и удалять данные из таблицы.

Использование интерфейса для определения операций с данными

Чтобы определить, как приложение должно обращаться к данным таблицы, следует создать интерфейс с аннотациями. Интерфейс определяет объект DAO (Data Access Object, то есть «объект доступа к данным») со всеми методами, необходимыми приложению для вставки, чтения, обновления и удаления данных. Чтобы вы лучше поняли, как это делается, мы создадим новый интерфейс с именем TaskDao, который используется приложением Tasks для взаимодействия с данными task_table.

Создание интерфейса TaskDao

Начнем с создания интерфейса. Выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя файла «TaskDao» и выберите вариант Interface. После того как файл будет создан, убедитесь в том, что код выглядит так:

```
package com.hfad.tasks
interface TaskDao {

}
```

Использование @Dao для пометки интерфейса для доступа к данным

На следующем шаге необходимо сообщить Room, что интерфейс TaskDao определяет методы доступа к данным Для этого интерфейс помечается аннотацией @Dao:

```
@Dao
interface TaskDao {
}
```

После того как интерфейс будет помечен аннотацией @Dao, следует добавить методы с аннотациями, которые будут использоваться приложением для взаимодействия с данными. Например, если вы хотите, чтобы в приложении выполнялась вставка записей, необходимо добавить соответствующий метод в интерфейс; чтобы выполнять чтение, вы добавляете другой метод, и т. д. К счастью, Room предоставляет четыре аннотации @Insert, @Update, @Delete и @Query, с которыми эти методы добавляются проще простого. Давайте добавим несколько методов доступа к данным в TaskDao.

Использование аннотации @Insert для вставки записи Начнем с определения метода доступа к данным insert(), который будет использоваться приложением для вставки задачи в task_table. Метод имеет один параметр Task с данными вставляемой задачи. Метод помечается аннотацией @Insert, которая сообщает Room, что метод используется для вставки записей.

Полный код метода `insert()`:

```
@Insert
fun insert(task: Task)
```

Встречая аннотацию `@Insert`, `Room` автоматически генерирует весь код, необходимый приложению для вставки записи в таблицу, чтобы вам не пришлось писать его самостоятельно. Например, для приведенного выше метода `insert()` будет сгенерирован весь код, необходимый для вставки данных объекта `Task` в таблицу `task_table`:

Любые методы, помеченные аннотацией `@Insert`, могут получать в аргументах один или несколько объектов сущностей, то есть объектов, тип которых помечен аннотацией `@Entity`. Методы `@Insert` также могут получать коллекции объектов сущностей. Например, следующий метод вставляет все записи, включенные в аргумент `List<Task>`:

```
@Insert
fun insertAll(tasks: List<Task>)
```

Использование @Update для обновления записей

`Room` также может генерировать весь код, необходимый для обновления одной или нескольких существующих записей в таблице. Для этого в интерфейс `DAO` добавляется метод, помеченный аннотацией `@Update`. Например, следующий метод `update()` генерирует весь код, необходимый для обновления существующей записи с описанием задачи:

```
@Update
fun update(task: Task)
```

При вызове этот метод обновляет запись с соответствующим значением `taskId`, чтобы ее данные совпадали со значениями свойств объекта `Task`.

Использование @Delete для удаления записей

Также существует аннотация `@Delete`, предназначенная для пометки любых методов, которые должны удалять из таблицы конкретные записи. Например, для удаления одной записи можно воспользоваться следующим методом:

```
@Delete
fun delete(task: Task)
```

Код, генерируемый `Room` для этого метода, удаляет запись с соответствующим значением `taskId`:

Использование @Query для всех остальных операций

Все остальные методы доступа к данным помечаются аннотацией `@Query`. Эта аннотация позволяет определить команду SQL (с командами SELECT, INSERT, UPDATE или DELETE), которая будет выполняться при вызове метода. Например, в приложении `Tasks` следующий код может использоваться для определения метода с именем `get()`, возвращающего объект `Live Data Task` для записи с соответствующим значением `taskId`:

```
@Query("SELECT * FROM task_table WHERE taskId = :taskId")
fun get(taskId: Long): LiveData<Task>
```

Также определим метод `getAll()`, который возвращает объект `Live Data` List со всеми записями, хранящимися в таблице:

```
@Query("SELECT * FROM task_table ORDER BY taskId DESC")
fun getAll(): LiveData<List<Task>>
```

Все эти методы возвращают объекты `Live Data`, и приложение может использовать их для оповещения об изменениях данных. Мы воспользуемся этой возможностью позднее в этой главе, чтобы поддерживать актуальность списка записей, отображаемого в `TasksFragment`. Теперь вы знаете все, что необходимо знать для завершения кода `TaskDao`. Полный код приводится на следующей странице.

Полный код TaskDao.kt

Ниже приведен полный код интерфейса `TaskDao`; обновите код `TaskDao.kt`:

```
package com.hfad.tasks
import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
@Dao
interface TaskDao {
 @Insert
 fun insert(task: Task)
 @Update
 fun update(task: Task)
 @Delete
 fun delete(task: Task)
 @Query("SELECT * FROM task_table WHERE taskId = :taskId")
 fun get(taskId: Long): LiveData<Task>
 @Query("SELECT * FROM task_table ORDER BY taskId DESC")
 fun getAll(): LiveData<List<Task>>
}
```

Мы написали полный код класса данных `Task` (который определяет таблицу) и интерфейса `TaskDao` (который определяет методы доступа к данным). А сейчас вы узнаете, как определяются базы данных.

Создание абстрактного класса TaskDatabase

Чтобы определить базу данных приложения, следует создать абстрактный класс. Абстрактный класс определяет имя и номер версии базы данных, а также любые классы или интерфейсы, определяющие таблицы и методы доступа к базе данных. В приложении `Tasks` для определения базы данных будет использован абстрактный класс с именем `TaskDatabase`. Выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду `File→New→Kotlin Class/File`. Введите имя файла «TaskDatabase» и выберите вариант `Class`.

Класс `TaskDatabase` должен расширять `RoomDatabase`, поэтому обновите код `TaskDatabase.kt`:

```
package com.hfad.tasks
import androidx.room.RoomDatabase
abstract class TaskDatabase : RoomDatabase() {
}
```

Пометка класса аннотацией @Database

Затем класс необходимо пометить аннотацией `@Database`, которая сообщает `Room`, что он определяет базу данных. Эта задача решается следующим кодом:

```
package com.hfad.tasks
import androidx.room.Database
import androidx.room.RoomDatabase
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
}
```

Как видно из этого примера, аннотация `@Database` имеет три атрибута: `entities`, `version` и `exportSchema`.

- `entities` задает любые классы (помеченные атрибутом `@Entity`), определяющие таблицы, которые библиотека Room должна добавить в базу данных. В приложении `Tasks` это класс данных `Task`.
- `version` — целое число, определяющее номер версии базы данных. В данном примере он равен 1, так как создается первая версия базы данных.
- `exportSchema` сообщает `Room`, нужно ли экспортировать схему базы данных в папку, чтобы сохранить историю ее версий. В данном примере этому атрибуту присваивается `false`.

Добавление свойств для интерфейсов DAO

Затем необходимо определить интерфейсы (помеченные аннотацией `@Dao`), которые будут использоваться для доступа к данным. Для этого следует добавить свойство для каждого интерфейса. В приложении `Tasks` определяется один интерфейс `DAO` с именем `TaskDao`, поэтому в код `TaskDatabase` добавляется новое свойство `taskDao`:

```
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
 abstract val taskDao: TaskDao
}
```

Создание и возвращение экземпляра базы данных

Наконец, нам понадобится метод `getInstance()`, который создает базу данных и возвращает ее экземпляр. Код выглядит так:

```
abstract class TaskDatabase : RoomDatabase() {
 ...
 companion object {
 @Volatile
 private var INSTANCE: TaskDatabase? = null
 fun getInstance(context: Context): TaskDatabase {
 synchronized(this) {
 var instance = INSTANCE
 if (instance == null) {
 instance = Room.databaseBuilder(
 context.applicationContext,
 TaskDatabase::class.java,
 "tasks_database"
 ).build()
 INSTANCE = instance
 }
 return instance
 }
 }
 }
}
```

И это все, что нам понадобится для класса `TaskDatabase`. Полный код будет приведен ниже.

Полный код TaskDatabase.kt

Ниже приведен полный код абстрактного класса `TaskDatabase`; обновите код `TaskDatabase.kt`:

```
package com.hfad.tasks
import android.content.Context
import androidx.room.Database
```

```
import androidx.room.Room
import androidx.room.RoomDatabase
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
 abstract val taskDao: TaskDao
 companion object {
 @Volatile
 private var INSTANCE: TaskDatabase? = null
 fun getInstance(context: Context): TaskDatabase {
 synchronized(this) {
 var instance = INSTANCE
 if (instance == null) {
 instance = Room.databaseBuilder(
 context.applicationContext,
 TaskDatabase::class.java,
 "tasks_database"
 ).build()
 INSTANCE = instance
 }
 return instance
 }
 }
 }
}
```

Мы написали весь код базы данных, необходимый для приложения `Tasks`.

Снова о MVVM

Ранее мы упоминали о том, что структура приложения `Tasks` будет основана на архитектурном паттерне проектирования MVVM («модель–представление–модель представления»). Кратко напомним, как выглядят такие структуры:

Представление Весь код, отвечающий за пользовательский интерфейс: код активностей, фрагментов и макетов. **Модель представления** Объекты моделей представлений отвечают за бизнес-логику и данные каждого представления. **Модель** Код базы данных приложения, включая ее сущности и объекты DAO.

К настоящему моменту мы написали весь код базы данных, необходимый приложению; для этого были созданы файлы сущностей, DAO и файлы определения базы данных (`Task`, `TaskDao` и `TaskDatabase`). Написание всего кода базы данных означает, что мы завершили часть архитектуры приложения, относящуюся к модели.

Модель представления

На следующем этапе мы будем работать над моделью представления. Для этого будет создана модель представления (`TasksViewModel`), в которой будет храниться бизнес-логика `TasksFragment`. Модель приложения будет включать методы, использующие `TaskDao` для вставки записей в базу данных. Итак, займемся созданием `TasksViewModel`.

Создание `TasksViewModel`

Чтобы создать класс `TasksViewModel`, выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду `File→New→Kotlin Class/File`. Введите имя «TasksViewModel» и выберите вариант `Class`. Мы обновим код модели представления, чтобы `TasksFragment` мог использовать его для вставки новых записей с описаниями задач. Для этого коду потребуются три составляющие:

1. Ссылка на объект `TaskDao`. `TasksViewModel` будет использовать этот объект для взаимодействия с базой данных, поэтому он будет передаваться модели представления в ее конструкторе.
2. Строковое свойство для хранения имени новой задачи. Когда пользователь вводит новое имя задачи, `TasksFragment` обновляет это свойство его значением.
3. Метод `addTask()`, который будет вызываться из `TasksFragment`. Этот метод будет создавать новый объект `Task`, присваивать ему имя и вставлять его в базу данных вызовом метода `insert()` объекта `TaskDao`.
4.

Ниже приведен базовый код модели представления для этих трех составляющих; обновите код `TasksViewModel.kt`, чтобы он выглядел так:

```
package com.hfad.tasks
import androidx.lifecycle.ViewModel
class TasksViewModel(val dao: TaskDao) : ViewModel() {
 var newTaskName = ""
 fun addTask() {
 val task = Task()
 task.taskName = newTaskName
 dao.insert(task)
 }
}
```

Но прежде чем `TasksFragment` сможет вызывать метод `addTask()`, необходимо внести еще одно изменение.

Операции баз данных могут работать меедлееенноооо...

Некоторые задачи в мире Android, такие как вставка записей в базу данных, потенциально могут занимать много времени. По этой причине библиотека хранения данных `Room` настаивает на том, что все операции доступа к данным должны выполняться в фоновом потоке, чтобы они не блокировали главный поток Android и не тормозили пользовательский интерфейс. Так как мы разрабатываем приложения Android на языке Kotlin, для того чтобы все методы доступа к данным выполнялись в фоновом режиме, в нашем приложении будут применяться сопрограммы.

Для фонового выполнения кода доступа к данным будут использоваться сопрограммы

Сопрограмма представляет собой упрощенный программный поток, который позволяет выполнять блоки кода в асинхронном режиме. Использование сопрограмм означает, что фоновое задание (например, вставка записей в базу данных) запускается так, что остальным частям кода не приходится

дожидаться его завершения. Взаимодействие пользователя с приложением становится более плавным. Перевод кода доступа к данным на использование сопрограмм происходит относительно прямолинейно. Вам придется внести всего два изменения:

1. Все методы доступа к данным в DAO помечаются ключевым словом `suspend`. Тем самым каждый метод преобразуется в сопрограму, которая выполняется в фоновом режиме и может приостанавливаться, например:

```
@Insert
suspend fun insert(task: Task)
```

2. Сопрограммы DAO запускаются в фоновом режиме. Например, для вызова метода `insert()` объекта `TaskDao` из `TasksViewModel` используется следующий код:

```
viewModelScope.launch {
 ...
 dao.insert(task)
}
```

Эти изменения необходимы для всех методов доступа к данным, кроме тех, которые возвращают данные `Live Data`. `Room` уже использует фоновый поток для методов, возвращающих объекты `Live Data`, а это означает, что вам не придется вносить дополнительные изменения в ваш код. Обновим код `TaskDao` и `TasksViewModel`, чтобы в нем использовались сопрограммы.

1. Пометка методов `TaskDao` ключевым словом `suspend` Прежде всего необходимо пометить все методы доступа к данным `TaskDao`, не использующие `Live Data`, ключевым словом `suspend`. Это означает, что изменение необходимо применить ко всем методам, кроме `get()` и `getAll()`. Ниже приведен полный код `TaskDao.kt`; обновите код и внесите изменения:

```
package com.hfad.tasks
import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
@Dao
interface TaskDao {
 @Insert
 suspend fun insert(task: Task)
 @Update
 suspend fun update(task: Task)
 @Delete
 suspend fun delete(task: Task)
 @Query("SELECT * FROM task_table WHERE taskId = :key")
 fun get(key: Long): LiveData<Task>
 @Query("SELECT * FROM task_table ORDER BY taskId DESC")
```

```
 fun getAll(): LiveData<List<Task>>
}
```

И это весь код, необходимый для преобразования методов `TaskDao` в сопрограммы, которые могут выполняться в фоновом режиме.

2. Метод `insert()` запускается в фоновом режиме Далее необходимо обновить метод `addTask()` объекта `TasksViewModel`, чтобы он запускал метод `insert()` объекта `TaskDao` как сопрограмму. Этот код приведен ниже; обновите файл `TasksViewModel.kt`:

```
package com.hfad.tasks
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
class TasksViewModel(val dao: TaskDao) : ViewModel() {
 var newTaskName = ""
 fun addTask() {
 viewModelScope.launch {
 val task = Task()
 task.taskName = newTaskName
 dao.insert(task)
 }
 }
}
```

Это изменение означает, что при каждом вызове метода `addTask()` будет вызываться метод `insert()` класса `TaskDao` (сопрограмма) для вставки записей в фоновом режиме. Мы написали весь код, необходимый для модели представления приложения `Task`. Затем объект `TasksViewModel` будет добавлен в `TasksFragment`, чтобы он мог обращаться к свойствам и методам модели представления, а пользователь получил возможность вставлять записи с описаниями задач.

TasksViewModel необходима фабрика модели представления

Как вы узнали, чтобы добавить модель представления в код фрагмента, следует запросить ее у провайдера модели представления. Провайдер модели представления возвращает текущий объект модели представления фрагмента, если он существует, или создает его в противном случае. Если модель представления имеет конструктор без аргументов, провайдер модели представления может создать экземпляр без дополнительной помощи. Но если у конструктора есть аргументы, ему потребуется поддержка фабрики модели представления. В приложении `Tasks` для получения объекта `TasksViewModel` понадобится провайдер модели представления. Так как конструктор `TasksViewModel` получает аргумент `TaskDao`, сначала необходимо определить класс `TasksViewModelFactory`.

Создание TasksViewModelFactory

Чтобы создать фабрику, выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду `File→New→Kotlin Class/File`. Введите имя файла «TasksViewModelFactory» и выберите вариант `Class`.

Код `TasksViewModelFactory` почти не отличается от кода фабрики модели представления. Обновите файл `TasksViewModelFactory.kt`, чтобы он выглядел так:

```
package com.hfad.tasks
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
class TasksViewModelFactory(private val dao: TaskDao)
 : ViewModelProvider.Factory {
 override fun <T: ViewModel?> create(modelClass: Class<T>): T {
 if (modelClass.isAssignableFrom(TasksViewModel::class.java)) {
 return TasksViewModel(dao) as T
 }
 throw IllegalArgumentException("Unknown ViewModel")
 }
}
```

Фабрика модели представления готова. Давайте воспользуемся ею для включения объекта `TasksViewModel` в `TasksFragment`.

TasksViewModelFactory необходим объект TaskDao

Для включения `TasksViewModel` в `TasksFragment` необходимо создать объект `TasksViewModelFactory` и передать его провайдеру модели представления. Провайдер использует фабрику для создания модели представления. Но тут возникает проблема: конструктору `TasksViewModelFactory` требуется аргумент `TaskDao`, а значит, такой объект необходимо получить, чтобы иметь возможность создать объект. Однако `TaskDao` — интерфейс, а не конкретный класс. Как же получить объект `TaskDao`?

Код TaskDatabase содержит свойство TaskDao

Когда мы писали код `TaskDatabase`, в него были включены две ключевые составляющие: свойство `TaskDao` с именем `taskDao` и метод `getInstance()`, возвращающий экземпляр базы данных. Кратко напомню, как выглядит этот код:

```
abstract class TaskDatabase : RoomDatabase() {
 abstract val taskDao: TaskDao
 companion object {
 ...
 fun getInstance(context: Context): TaskDatabase {
 ...
 }
 }
}
```

Чтобы получить ссылку на объект `TaskDao` в коде `TasksFragment`, можно вызвать метод `getInstance()` объекта `TaskDatabase` и обратиться к его свойству `taskDao`. Соответствующий код выглядит так:

```
val application = requireNotNull(this.activity).application
val dao = TaskDatabase.getInstance(application).taskDao
```

Этот код получает ссылку на текущее приложение, строит базу данных, если она еще не существует, и возвращает ее экземпляр. Затем ее объект `TaskDao` присваивается локальной переменной с именем `dao`.

Теперь вы знаете, как получить ссылку на объект `TaskDao`, и мы можем обновить код `TasksFragment` для создания объекта `TasksViewModelFactory`, который будет использоваться для получения `TasksViewModel`. Давайте рассмотрим этот код

Обновленный код TasksFragment.kt

Ниже приведен обновленный код `TasksViewModel`. Как видите, теперь в нем присутствует код получения объекта `TaskDao` и создания `TasksViewModelFactory`. Фабрика передается провайдеру модели представления, который использует ее для получения экземпляра `TasksViewModel`.

Обновите код `TasksFragment.kt`:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.tasks.databinding.FragmentTasksBinding
import androidx.lifecycle.ViewModelProvider
class TasksFragment : Fragment() {
 private var _binding: FragmentTasksBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentTasksBinding.inflate(inflater, container, false)
 val view = binding.root
 val application = requireNotNull(this.activity).application
 val dao = TaskDatabase.getInstance(application).taskDao
 val viewModelFactory = TasksViewModelFactory(dao)
 val viewModel = ViewModelProvider(
 this, viewModelFactory).get(TasksViewModel::class.java)
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

TasksFragment может использовать связывание представлений

Объект `TasksViewModel` был добавлен в `TasksFragment`, и фрагмент может использовать его свойства и методы для вставки записи в базу данных. Для этого будет использоваться связывание данных, которое предоставляет макету прямой доступ к свойствам и методам модели представления. Чтобы включить связывание данных, необходимо сначала добавить переменную связывания данных в макет фрагмента; включите приведенный ниже код в раздел `<data>` файла `fragment_tasks.xml`:

```
<layout
 ...>
 <data>
 <variable
 name="viewModel"
 type="com.hfad.tasks.TasksViewModel" />
 </data>
 ...
</layout>
```

Затем свойство `viewModel` фрагмента присваивается переменной связывания данных, для чего в метод `onCreateView()` объекта `TasksFragment` включается следующая строка:

```
override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 ...
 val viewModel = ViewModelProvider(
 this, viewModelFactory).get(TasksViewModel::class.java)
 binding.viewModel = viewModel
 return view
 }
```

Полный код обоих файлов будет приведен через несколько страниц. А пока обновим файл `fragment_tasks.xml`, чтобы переменная `viewModel` использовалась в нем для вставки записей в базу данных.

Для вставки данных будет использоваться связывание данных

Чтобы вставить в базу данных новую запись с описанием задачи, необходимо: (1) задать свойству `newTaskName` объекта `TasksViewModel` имя новой задачи и (2) вызвать его метод `addTask()`. И то и другое можно сделать в макете `TasksFragment` с использованием связывания данных.

Присваивание свойству newTaskName объекта TasksViewModel

Чтобы задать значение свойства `newTaskName` модели представления, мы свяжем его с текстовым полем `task_name` в макете фрагмента. Это делается с помощью кода:

```
<EditText
 android:id="@+id/task_name"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:inputType="text"
 android:hint="Enter a task name"
 android:text="@={viewModel.newTaskName}" />
```

Обратите внимание: для связывания свойства с текстовым полем используется конструкция `@=` вместо `@`. `@=` означает, что текстовое поле может обновлять свойство, с которым оно связано, — в данном случае `newTaskName`.

Вызов метода `addTask()` объекта `TasksViewModel` Для добавления записи с описанием задачи, мы воспользуемся связыванием данных, чтобы по щелчку на кнопке `Save Task` макета вызывался метод `addTask()` модели представления. Вы уже знаете, как это делается, поэтому мы просто приведем код:

```
<Button
 android:id="@+id/save_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Save Task"
 android:onClick="@{() -> viewModel.addTask()}" />
```

И это все изменения, которые необходимо внести в `fragment_tasks.xml` для вставки записи в базу данных. Давайте посмотрим, как выглядит полный код.

Полный код fragment_tasks.xml

Ниже приведен полный код макета `TasksFragment`; обновите файл `fragment_tasks.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".TasksFragment">
 <data>
 <variable
 name="viewModel"
 type="com.hfad.tasks.TasksViewModel" />
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
 <EditText
 android:id="@+id/task_name"
 android:layout_width="match_parent"
```

```
 android:layout_height="wrap_content"
 android:inputType="text"
 android:hint="Enter a task name"
 android:text="@={viewModel.newTaskName}" />
 <Button
 android:id="@+id/save_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Save Task"
 android:onClick="@{() -> viewModel.addTask()}" />
 <TextView
 android:id="@+id/tasks"
 android:layout_width="match_parent"
 android:layout_height="wrap_content" />
 </LinearLayout>
</layout>
```

Полный код TasksFragment.kt

Прежде чем запускать приложение, также необходимо убедиться в том, что `TasksFragment` включает весь код, необходимый для присваивания значения переменной связывания данных макета. Полный код приведен ниже; обновите файл `TasksFragment.kt`, если это не было сделано ранее:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding
class TasksFragment : Fragment() {
 private var _binding: FragmentTasksBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentTasksBinding.inflate(inflater, container, false)
 val view = binding.root
 val application = requireNotNull(this.activity).application
 val dao = TaskDatabase.getInstance(application).taskDao
 val viewModelFactory = TasksViewModelFactory(dao)
 val viewModel = ViewModelProvider(
 this, viewModelFactory).get(TasksViewModel::class.java)
 binding.viewModel = viewModel
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
```

```
    }
}
```

Давайте разберемся, что происходит при выполнении кода.

Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. `TasksFragment` вызывает метод `TaskDatabase.getInstance()`, который строит базу данных, если она еще не существует. При этом базе данных присваивается имя `tasks_database`, а класс данных `Task` используется для создания таблицы с именем `task_table`. После этого метод возвращает экземпляр базы данных.

2. `TasksFragment` получает объект `TaskDao` экземпляра базы данных и использует его для создания фабрики `TasksViewModelFactory`. Провайдер модели представления использует только что созданный объект `TasksViewModelFactory` для создания `TasksViewModel`.

3. `TasksFragment` присваивает переменной связывания данных `viewModel` макета объект `TasksViewModel`.

4. Пользователь вводит имя задачи в текстовом поле. Макет использует связывание данных, чтобы задать свойству `newTaskName` модели представления введенное значение.

5. Пользователь щелкает на кнопке `Save Task`. Связывание данных используется для вызова метода `addTask()` объекта `TaskViewModel`.

6. Метод `addTask()` создает новый объект `Task` и задает его свойству `taskName` значение `newTaskName`.

7. Метод `addTask()` вызывает метод `insert()` объекта `TaskDao`. Это приводит к вставке записи данных объекта `Task` в таблицу базы данных. Таблица автоматически генерирует значение первичного ключа `taskId`.

При запуске приложения в `MainActivity` отображается фрагмент `TasksFragment`. Если ввести пару новых задач, по щелчку на кнопке `Save Task` вроде бы ничего не происходит.

На самом деле кнопка добавляет новые записи в базу данных, но мы их еще не видим. Чтобы увидеть эти записи в приложении, необходимо внести дополнительные изменения.

В TasksFragment должны выводиться записи

Вы узнали, как построить приложение, которое вставляет записи в базу данных `Room`. На следующем этапе необходимо обновить приложение так, чтобы в `TasksFragment` выводился список всех вставленных записей.

Вот как должна выглядеть новая версия `TasksFragment`:

Мы воспользуемся методом `getAll()` объекта `TaskDao` для чтения всех записей из базы данных. Затем прочитанные записи отображаются в текстовом представлении `TasksFragment`, для чего они форматируются в строку. Итак, за дело!

Чтение всех задач из базы данных методом getAll()

Начнем с добавления в `TasksViewModel` свойства с именем `tasks`, в котором будет храниться список всех задач в базе данных. Чтобы добавить задачи в свойство, присвойте ему результат вызова `getAll()` объекта `TaskDao`:

```
class TasksViewModel(val dao: TaskDao) : ViewModel() {
 ...
 val tasks = dao.getAll()
 ...
}
```

Как вы, возможно, помните, метод `getAll()` определяется в `TaskDao.kt` следующим кодом:

```
@Dao
interface TaskDao {
...
 @Query("SELECT * FROM task_table ORDER BY taskId DESC")
 fun getAll(): LiveData<List<Task>>
}
```

Метод возвращает объект `LiveData<List<Task>>`: список объектов `Task` с данными `Live Data`. Это означает, что свойство `tasks` объекта `TasksViewModel` тоже относится к типу `LiveData<List<Task>>`:

Так как свойство `tasks` использует данные `Live Data`, оно всегда включает последние изменения, внесенные пользователем. Например, если в `task_table` добавляется новая задача, то значение свойства `tasks` автоматически обновляется и в него включается новая запись. Такое использование `Live Data` весьма удобно для приложения `Tasks`, потому что оно может использоваться для отображения в `TasksFragment` актуального списка задач. Но прежде чем переходить к реализации этой возможности, необходимо еще кое-что узнать.

LiveData<List> — более сложный тип

При описании связывания даных мы показали, как связать представление со строкой или числом, включая значения `Live Data`. Это было возможно благодаря тому, что строки и числа — простые объекты, которые могут легко отображаться в текстовых представлениях.

Однако на этот раз ситуация иная. Свойство tasks имеет тип `LiveData<List<Task>>` — намного более сложный, чем, допустим, `LiveData<String>`. Из-за этого мы не можем просто связать текстовое представление макета со свойством напрямую, потому что текстовое представление не будет знать, как его отобразить. Что же делать?

Использование Transformations.map() для преобразования объектов Live Data

Прежде чем использовать связывание данных для отображения задач пользователя, необходимо сначала преобразовать `LiveData<List<Task>>` к более простой форме, которую умеет отображать текстовое представление. Для этого мы создадим новое свойство с именем `tasksString`, в котором будет храниться версия свойства `tasks` в формате `LiveData<String>`. Для создания `LiveData<String>` будет использоваться метод `Transformations.map()`. Метод получает аргумент `LiveData` и лямбда-выражение, которое указывает, как должен преобразоваться объект `Live Data`. Он возвращает новый объект `LiveData`. Например, для преобразования свойства `tasks` в `LiveData<String>` можно воспользоваться следующим кодом:

```
val tasksString = Transformations.map(tasks) {
 tasks -> formatTasks(tasks)
}
```

где `formatTasks()` — метод (который мы должны написать), форматирующий список задач из свойства `tasks` в `String`. Далее метод `Transformations.map()` упаковывает `String` в объект `LiveData`, чтобы он возвращал `LiveData<String>`. Метод `Transformations.map()` наблюдает за переданным ему объектом `LiveData` и выполняет лямбда-выражение каждый раз, когда он получает оповещение об изменениях. Это означает, что в приведенном выше коде `tasksString` будет автоматически включать все новые записи задач, вводимые пользователем. На нескольких ближайших страницах мы добавим свойство `tasksString` в `TasksViewModel`, а затем воспользуемся связыванием данных для вывода его значения в макете `TasksFragment`. Такой подход означает, что все задачи пользователя будут отображаться в текстовом представлении, которое всегда остается актуальным.

Обновление кода TasksViewModel

Начнем с обновления кода `TasksViewModel`, чтобы он включал новые свойства `tasks` и `tasksString`. Мы также добавим два метода: `formatTasks()` и `formatTask()`, которые упрощают преобразовани `LiveData<List<Task>>` в `LiveData<String>`. Ниже приведен полный код `TasksViewModel`. Обновите файл `TasksViewModel.kt`:

```
package com.hfad.tasks
import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
class TasksViewModel(val dao: TaskDao) : ViewModel() {
 var newTaskName = ""
 private val tasks = dao.getAll()
val tasksString = tasks.map { tasks -> formatTasks(tasks) }
 fun addTask() {
 viewModelScope.launch {
 val task = Task()
 task.taskName = newTaskName
 dao.insert(task)
 }
 }
 fun formatTasks(tasks: List<Task>): String {
```

```
 return tasks.fold("") {
 str, item -> str + '\n' + formatTask(item)
 }
 }
 fun formatTask(task: Task): String {
 var str = "ID: ${task.taskId}"
 str += '\n' + "Name: ${task.taskName}"
 str += '\n' + "Complete: ${task.taskDone}" + '\n'
 return str
 }
}
```

Свойство tasksString будет связано с текстовым представлением в макете

Затем мы используем механизм связывания данных для привязки текстового представления `tasks` в макете `TasksFragment` к свойству `tasksString` в `TasksViewModel`. Ниже приведен код решения этой задачи; обновите файл `fragment_tasks.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".TasksFragment">
 <data>
 <variable
 name="viewModel"
 type="com.hfad.tasks.TasksViewModel" />
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
 <EditText
 android:id="@+id/task_name"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:inputType="text"
 android:hint="Enter a task name"
 android:text="@={viewModel.newTaskName}" />
 <Button
 android:id="@+id/save_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Save Task"
 android:onClick="@{() -> viewModel.addTask()}" />
<TextView
 android:id="@+id/tasks"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
```

```
 android:text="@{viewModel.tasksString}" />
 </LinearLayout>
</layout>
```

И это все, что необходимо изменить в `fragment_tasks.xml` для вывода строки с задачами пользователя. Но прежде чем проводить тест-драйв приложения, необходимо внести еще одно изменение

Макет должен реагировать на обновления данных Live Data

Как вы уже знаете, свойство `tasksString` относится к типу `LiveData<String>`; это означает, что оно автоматически учитывает любые обновления, которые вносятся в записи, хранящиеся в базе данных. Например, если пользователь вставляет новую запись, ее данные добавляются в значение `tasksString`. Так как мы используем связывание данных для вывода значения свойства `tasksString` в текстовом представлении `tasks`, необходимо позаботиться о том, чтобы макет оповещался об обновлении значения свойства `tasksString`. Это означает, что новые записи начнут отображаться в текстовом представлении сразу же после их вставки. Вы уже знаете, как заставить макет реагировать на обновления данных `Live Data` — назначением владельца жизненного цикла макета в коде фрагмента:

```
binding.lifecycleOwner = viewLifecycleOwner
```

Следовательно, код `TasksFragment` необходимо обновить и добавить в него эту строку. Полный код приведен на следующей странице, а потом мы рассмотрим, что же происходит при запуске приложения.

Полный код TasksFragment.kt

Ниже приведен полный код `TasksFragment`; обновите файл `TasksFragment.kt`:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding
class TasksFragment : Fragment() {
 private var _binding: FragmentTasksBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentTasksBinding.inflate(inflater, container, false)
 val view = binding.root
 val application = requireNotNull(this.activity).application
```

```
    val dao = TaskDatabase.getInstance(application).taskDao
    val viewModelFactory = TasksViewModelFactory(dao)
    val viewModel = ViewModelProvider(
    this, viewModelFactory).get(TasksViewModel::class.java)
    binding.viewModel = viewModel
    binding.lifecycleOwner = viewLifecycleOwner
    return view
    }
    override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
    }
  }
```

Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. `TasksFragment` создает объект `TasksViewModelFactory`,который используется провайдером модели представления для создания `TasksViewModel`.
2. Свойству `tasks` объекта `TasksViewModel` присваивается результат вызова метода `getAll()` объекта `TaskDao`, который возвращает `LiveData<List<Task>>`. В нем содержится список всех записей из базы данных в виде данных `Live Data`.
3. Свойство `tasksString` объекта `TasksViewModel` использует метод `Transformations.map()` для преобразования значения свойства `tasks` в `LiveData<String>`. Метод возвращает записи свойства `tasks`, отформатированные в виде одной строки.
4. Текстовое представление в макете использует связывание данных для вывода значения `tasksString`.
5. Пользователь вводит новую запись в базу данных. Свойство tasks объекта `TasksViewModel` автоматически обновляется (с использованием `Live Data`) для включения новой записи.
6. Свойство `tasksString` реагирует на обновление свойства `tasks`. Благодаря использованию механизма `Live Data` новая запись автоматически включается в строку.
7. Макет реагирует на обновление свойства `tasksString`. Запись, только что введенная пользователем, включается в текстовое представление.

При запуске приложения, как и прежде, отображается фрагмент `TasksFragment`, но на этот раз выводится список задач, вводившихся ранее. Когда вы вводите новую задачу, она добавляется в список сразу же после щелчка на кнопке `Save Task`.

Поздравляем! Вы научились строить приложения, использующие паттерн MVVM для взаимодействия с базой данных `Room`. Записи пользователя остаются в базе данных, чтобы они сохранялись при закрытии приложения, а приложение отображает новые записи сразу же после вставки.

Резюме

- Room — библиотека хранения данных, работающая на базе SQLite.
- Структура приложения Room обычно строится по архитектурному паттерну проектирования MVVM, что означает «модель–представление–модель представления».

- Таблицы баз данных определяются классом данных с аннотациями. Аннотация `@Entity` отдает команду `Room` использовать класс для создания таблицы. Аннотация `@PrimaryKey` определяет столбец первичного ключа. Аннотация `@ColumnInfo` определяет имена столбцов.
- Имена методов доступа к данным определяются в интерфейсе с аннотациями. Аннотация `@Dao` отдает команду `Room` использовать методы для доступа к данным. Аннотации `@Insert`, `@Update`, `@Delete` и `@Query` определяют операции с данными.
- База данных определяется абстрактным классом с аннотацией `@Database`.
- При обновлении схемы базы данных необходимо увеличить номер версии базы данных.
- Код доступа к данным должен выполняться в фоновом режиме. Для этого можно преобразовать методы доступа к данным, которые не возвращают данные `Live Data`, в сопрограммы.
- Для преобразования объектов `Live Data` можно воспользоваться методом `Transformations.map()`. Метод наблюдает за объектом `Live Data`, выполняет лямбда-выражение при изменении значения объекта и возвращает другой объект `Live Data`.