12.11.18

Технологии разработки мобильных приложений

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 5

УЧЕБНЫЕ ВОПРОСЫ

1 Хранение данных в OS Android

В OS Android имеется несколько способов хранения данных. Помимо прямого доступа к внутренним и внешним областям хранения Android-устройства, платформа Android предлагает базы данных SQLite для хранения реляционных данных, специальные файлы для хранения пар ключ-значение. Более того, приложения для Android также могут использовать сторонние базы данных, которые предлагают поддержку NoSQL. В 2017 году от компании Google была представлена библиотека Room, представляющая удобную обертку для работы с базой данных SQLite.

1.1 Shared preferences

Для быстрого способа сохранения нескольких строк или номеров используется файл настроек (preferences). Хранение осуществляется в виде ключ-значение. Подходит для хранения глобальных данных. Также возможно хранить небольшие структуры, предварительно конвертированные в JSON и преобразованные в String.

Чтобы получить экземпляр класса SharedPreferences для получения доступа к настройкам в коде приложения используются три метода:

- getPreferences() — внутри активности, чтобы обратиться к определенному для активности предпочтению;
- getSharedPreferences() — внутри активности, чтобы обратиться к предпочтению на уровне приложения;
- getDefaultSharedPreferences() — из объекта PreferencesManager, чтобы получить общедоступную настройку, предоставляемую Android.

Активити и службы Android могут использовать метод getDefaultSharedPreferences() класса PreferenceManager, чтобы ссылаться на объект SharedPreferences, который может быть использован и для чтения, и для записи в файл настроек по умолчанию.

```
SharedPreferences myPreferences
        = PreferenceManager.getDefaultSharedPreferences(this);
```

Чтобы начать запись в файл настроек, вы должны вызвать метод edit() объекта SharedPreferences, который возвращает объект SharedPreferences.Editor.

```
SharedPreferences.Editor myEditor = myPreferences.edit();
```

Объект SharedPreferences.Editor имеет несколько интуитивных методов, которые можно использовать для хранения новых пар ключ-значение в файле настроек. Например, вы можете использовать метод putString(), чтобы поместить пару ключ-значение со значением типа String. Аналогично, вы можете использовать метод putFloat(), чтобы поместить пару ключ-значение, чьё значение типа float. Все эти методы возвращают экземпляр класса SharedPreferences, из которого можно получить соответствующую настройку с помощью ряда методов:

- getBoolean(String key, boolean defValue);
- getFloat(String key, float defValue);
- getInt(String key, int defValue);
- getLong(String key, long defValue);
- getString(String key, String defValue)

Следующий пример кода создаёт три пары ключ-значение:

```
myEditor.putString("NAME", "Alice");
myEditor.putInt("AGE", 25);
myEditor.putBoolean("SINGLE?", true);
```

После того, как вы добавили все пары, вы должны вызвать метод commit() объекта SharedPreferences.Editor, чтобы сохранить их.

```
myEditor.commit();
```

Чтение из объекта SharedPreferences производится вызовом соответствующего метода get*(). Например, чтобы получить пару ключ-значение, чьё значение является String, вы должны вызывать метод getString(). Вот фрагмент кода, который извлекает все значения, которые мы добавили ранее:

```
String name = myPreferences.getString("NAME", "unknown");
int age = myPreferences.getInt("AGE", 0);
boolean isSingle = myPreferences.getBoolean("SINGLE?", false);
```

Как вы видите из кода выше, второй параметр всех методов get*() ожидает значение по-умолчанию, которое является значением, которое должно быть возвращено, если ключ не существует и файле настроек.

Обратите внимание, что файлы настроек ограничены только строками и примитивными типами данных.

Файлы настроек хранятся в каталоге /data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml. Поэтому в отладочных целях, если вам нужно сбросить настройки в эмуляторе, то при помощи перспективы DDMS, используя файловый менеджер, зайдите в нужную папку, удалите файл настроек и перезапустите эмулятор, так как эмулятор хранит данные в памяти, которые он сбрасывает в файл. На устройстве вы можете удалить программу и поставить ее заново, то же самое можно сделать и на эмуляторе, что бывает проще, чем удалять файл настроек вручную и перезапускать эмулятор.

1.2 Задание

Требуется запомнить имя пользователя и название учебного заведения. После загрузки приложения отображаются значения из памяти

На экране требуется разместить – текстовое поле и 2 кнопки для сохранения и загрузки данных.

Создадим проект:

Project name:Pr5_LocalStore
Build Target: Android 2.3.3
Application name: SharedPreferences
Package name: com.mirea.sharedpref
Create Activity: MainActivity

Присвоение обработчиков и реализация onClick реализовано через поля кнопок “onClick”.

saveText – сохранение данных. Сначала с помощью метода getPreferences получаем объект sPref класса SharedPreferences, который позволяет работать с данными (читать и писать). Константа MODE_PRIVATE используется для настройки доступа и означает, что после сохранения, данные будут видны только этому приложению. Далее, чтобы редактировать данные, необходим объект Editor – получаем его из sPref. В метод putString указываем наименование переменной – это константа SAVED_TEXT, и значение – содержимое поля etText. Чтобы данные сохранились, необходимо выполнить commit. И для наглядности выводим сообщение, что данные сохранены.

loadText – загрузка данных. Так же, как и saveText, с помощью метода getPreferences получаем объект sPref класса SharedPreferences. MODE_PRIVATE снова указывается, хотя и используется только при записи данных. Здесь Editor мы не используем, т.к. нас интересует только чтение данных. Читаем с помощью метода getString – в параметрах указываем константу - это имя, и значение по умолчанию (пустая строка). Далее пишем значение в поле ввода etText и выводим сообщение, что данные считаны.

```
public class MainActivity extends Activity implements View.OnClick-
Listener {

    EditText etText;
    Button btnSave, btnLoad;

    SharedPreferences sPref;

    final String SAVED_TEXT = "saved_text";


    /**
     * Called when the activity is first created.
     */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        etText = (EditText) findViewById(R.id.etText);

        btnSave = (Button) findViewById(R.id.btnSave);
        btnSave.setOnClickListener(this);

        btnLoad = (Button) findViewById(R.id.btnLoad);
        btnLoad.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.btnSave:
                saveText();
                break;
            case R.id.btnLoad:
                loadText();
                break;
            default:
                break;
        }
    }

    void saveText() {
        sPref = getPreferences(MODE_PRIVATE);
        SharedPreferences.Editor ed = sPref.edit();
        ed.putString(SAVED_TEXT, etText.getText().toString());
        ed.commit();
        Toast.makeText(this, "Text saved", Toast.LENGTH_SHORT).show();
    }

    void loadText() {
        sPref = getPreferences(MODE_PRIVATE);
```

2 SQLite

SQLite — легковесный фреймворк, который, с одной стороны позволяет по максимуму использовать возможности SQL, с другой — бережно относится к ресурсам устройства. Проект с открытыми исходными кодами, поддерживающий стандартные возможности обычной SQL: синтаксис, транзакции и др. Занимает мало места - около 250 кб. Домашняя страница SQLite: http://www.sqlite.org.

SQLite поддерживает типы TEXT (аналог String в Java), INTEGER (аналог long в Java) и REAL (аналог double в Java). Остальные типы следует конвертировать, прежде чем сохранять в базе данных. SQLite сама по себе не проверяет типы данных, поэтому вы можете записать целое число в колонку, предназначенную для строк, и наоборот.

Тип	Описание
NULL	пустое значение
INTEGER	целочисленное значение
REAL	значение с плавающей точкой
TEXT	строки или символы в кодировке UTF-8, UTF-16BE или UTF-16LE

Отсутствует тип данных, работающий с датами. Возможно использование строковых значений, например, как 2018-12-10 (10 декабря 2018 года). Для даты со временем рекомендуется использовать формат 2018-12-10T09:10. В таких случаях возможно использование некоторых функций SQLite для добавления дней, установки начала месяца и т.д. SQLite не поддерживает часовые пояса. Также не поддерживается тип boolean. Возможно использование числа 0 для false и 1 для true.

Не рекомендуется для использования тип BLOB для хранения данных (картинки) в Android. Лучшим решением будет хранить в базе *путь к изображениям*, а сами изображения хранить в файловой системе.

Для начала работы с базой данных требуется задать необходимые настройки для создания или обновления базы данных.

Так как сама база данных SQLite представляет собой файл, то по сути при работе с базой данных, вы взаимодействуете с файлом. Поэтому операции чтения и записи могут быть довольно медленными. Настоятельно рекомендуется использовать асинхронные операции.

Когда ваше приложение создаёт базу данных, она сохраняется в каталоге DATA */data/имя_пакета/databases/имя_базы.db*.

Метод Environment.getDataDirectory() возвращает путь к каталогу DATA.

Основными пакетами для работы с базой данных являются *android.database* и *android.database.sqlite*.

База данных SQLite доступна только приложению, которое создаёт её. Если вы хотите дать доступ к данным другим приложениям, вы можете использовать контент-провайдеры (ContentProvider).

2.1 Классы для работы с SQLite

Работа с базой данных сводится к следующим задачам:

- Создание и открытие базы данных
- Создание таблицы
- Создание интерфейса для вставки данных (insert)
- Создание интерфейса для выполнения запросов (выборка данных)
- Закрытие базы данных

Класс ContentValues

Класс ContentValues используется для добавления новых строк в таблицу. Каждый объект этого класса представляет собой одну строку таблицы и выглядит как ассоциативный массив с именами столбцов и значениями, которые им соответствуют.

Курсоры

В Android запросы к базе данных возвращают объекты класса Cursor. Вместо того чтобы извлекать данные и возвращать копию значений, курсоры ссылаются на результирующий набор исходных данных. Курсоры позволяют управлять текущей позицией (строкой) в результирующем наборе данных, возвращаемом при запросе.

Класс SQLiteOpenHelper: Создание базы данных

Библиотека Android содержит абстрактный класс **SQLiteOpenHelper**, с помощью которого можно создавать, открывать и обновлять базы данных. Это основной класс, с которым вам придётся работать в своих проектах. При реализации этого вспомогательного класса от вас скрывается логика, на основе которой принимается решение о создании или обновлении базы данных перед ее открытием. Класс **SQLiteOpenHelper** содержит два обязательных абстрактных метода:

- **onCreate()** — вызывается при первом создании базы данных
- **onUpgrade()** — вызывается при модификации базы данных

Также используются другие методы класса:

- onDowngrade(SQLiteDatabase, int, int)
- onOpen(SQLiteDatabase)
- getReadableDatabase()
- getWritableDatabase()

SQLiteDatabase

Для управления базой данных SQLite существует класс SQLiteDatabase. В классе SQLiteDatabase определены методы query(), insert(), delete() и update() для чтения, добавления, удаления, изменения данных. Кроме того, метод execSQL() позволяет выполнять любой допустимый код на языке SQL применимо к таблицам базы данных, если вы хотите провести эти (или любые другие) операции вручную.

Каждый раз, когда вы редактируете очередное значение из базы данных, нужно вызывать метод refreshQuery() для всех курсоров, которые имеют отношение к редактируемой таблице.

Для составления запроса используются два метода: rawQuery() и query(), а также класс SQLiteQueryBuilder.

```
Cursor query (String table,
              String[] columns,
              String selection,
              String[] selectionArgs,
              String groupBy,
              String having,
              String sortOrder)
```

В метод **query()** передают семь параметров. Если какой-то параметр для запроса вас не интересует, то оставляете *null*:

- **table** — имя таблицы, к которой передается запрос;
- **String[] columnNames** — список имен возвращаемых полей (массив). При передаче *null* возвращаются все столбцы;
- **String whereClause** — параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение null возвращает все строки. Например: *_id = 19 and summary = ?*
- **String[] selectionArgs** — значения аргументов фильтра. Вы можете включить ? в "whereClause"". Подставляется в запрос из заданного массива;
- **String[] groupBy** - фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY). Если GROUP BY не нужен, передается null;
- **String[] having** — фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается null;

- String[] orderBy — параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается null.

3 Android architecture components

Android Architecture Components - это набор библиотек от Google, предназначенные для проектирование, тестирование и сопровождение приложений. В новом руководстве по архитектуре Android определены некоторые ключевые принципы, которые должны соответствовать хорошему Android-приложению, а также оно предлагает безопасный путь для разработчика по созданию хорошего приложения. Согласно руководству, хорошее приложение для Android должно обеспечить четкое разделение обязанностей и управлять пользовательским интерфейсом раздельно от модели. Любой код, который не обрабатывает взаимодействие с пользовательским интерфейсом или операционной системой, не должен находиться в Activity или Fragment, поскольку сохранение их как можно более чистыми позволит вам избежать многих проблем, связанных с жизненным циклом приложения. В конце концов, система может уничтожить действия или фрагменты в любое время. Кроме того, данные должны обрабатываться с помощью моделей, которые изолированы от пользовательского интерфейса и, следовательно, от проблем жизненного цикла.

Чтобы понять, что предлагает команда Android, мы должны знать все элементы компонентов архитектуры, так как именно они будут делать все тяжелую работу для нас. Есть четыре основных компонента, каждый из которых имеет определенную роль: Room, ViewModel, LiveData и Lifecycle. У всех этих частей есть свои обязанности, и они работают вместе, чтобы создать прочную архитектуру. Давайте рассмотрим упрощенную схему предлагаемой архитектуры, чтобы лучше понять ее.

1. Activity и Fragment Lifecycle - предоставляют несколько механизмов, сочетание которых, позволяет удобно обрабатывать повороты экрана.
    - Lifecycle - отслеживает текущий статус Activity и имеет возможность уведомлять об этом своих подписчиков;

- LiveData - получает и хранит данные, может отправлять их своим подписчикам;
- ViewModel - сохраняет необходимые объекты при повороте экрана.

2. Room - удобная обертка для работы с базой данных
3. Paging Library - библиотека для постраничной загрузки данных из базы данных, с сервера или любого другого источника.
4. Data Binding - связывание View's с объектами: отображение данных, запись изменений в данные полученные с View, объявление переменных (только объекты) в xml-разметке и взаимодействие с ними непосредственно в xml.
5. Navigation Architecture Component - новый компонент для навигации по экранам приложения
6. WorkManager - позволяет запускать фоновые задачи последовательно или параллельно, передавать в них данные, получать из них результат, отслеживать статус выполнения и запускать только при соблюдении заданных условий.

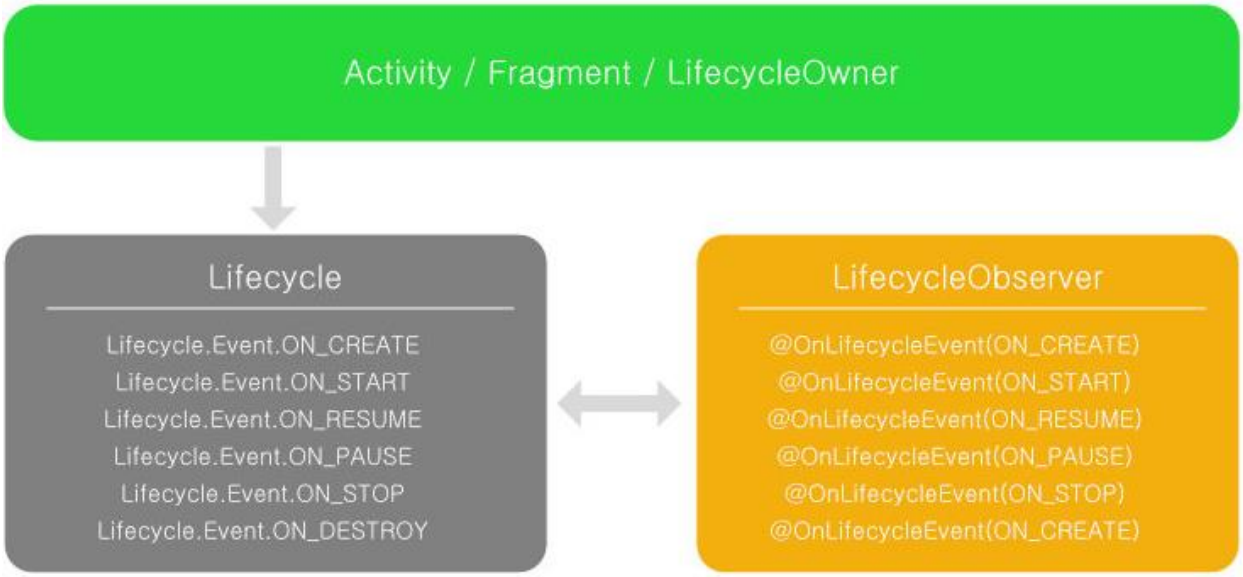
3.1 Activity и Fragment Lifecycle

3.1.1 Lifecycle

Компонент Lifecycle – призван упростить работу с жизненным циклом. Выделены основные понятия такие как LifecycleOwner и LifecycleObserver. LifecycleOwner – это интерфейс с одним методом getLifecycle(), который возвращает состояние жизненного цикла. Являет собой абстракцию владельца жизненного цикла (Activity, Fragment). Для упрощения добавлены классы LifecycleActivity и LifecycleFragment. LifecycleObserver – интерфейс, обозначает слушателя жизненного цикла owner-a. Не имеет методов, завязан на OnLifecycleEvent, который в свою очередь разрешает отслеживать жизненный цикл. Для описания состояния есть два enum. Первый Events — который обозначает изменение цикла и второй State — который описывает текущее состояние.

Events — повторяет стадии жизненного цикла и состоит из ON_CREATE, ON_RESUME, ON_START, ON_PAUSE, ON_STOP, ON_DESTROY, а также ON_ANY который информирует про изменения состояния без привязки к конкретному этапу.

State — состоит из следующих констант: INITIALIZED, CREATED, STARTED, RESUMED, DESTROYED. Для получения состояния используется метод getCurrentState() из Lifecycle. Также в Enum State реализован метод itAtLeast(State), который отвечает на вопрос является State выше или равным от переданного как параметр.



Довольно частая ситуация, когда в нашем приложении работает ряд процессов, которые зависят от этапа жизненного цикла. Будь-то воспроизведение медиа, локация, связь с сервисом и т.д. Таким образом приходится вручную отслеживать состояние и уведомлять о нём процесс. ИТОГ: захламление основного класса (Activity или Fragment) и снижение модульности, ведь требуется поддержка передачи состояния. С помощью же этого компонента возможно переложить всю ответственность на компонент и все что для этого нужно это объявить интересующий наш класс как observer и передать ему в onCreate() методе ссылку на owner.

Задание:

Создать новый модуль. В меню File> New> New Module> Phone & Tablet Module> Empty Activity. Приложение назвать lifecycle.

У Activity есть метод *getLifecycle*, который возвращает объект Lifecycle. На этот объект можно подписать слушателей, которые будут получать уведомления при смене lifecycle-состояния Activity.

Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют интерфейс LifecycleOwner. Именно этот интерфейс и добавляет им метод getLifecycle.

Т.е. версия, указанная в build.gradle файле модуля должна быть не ниже 26.1.0:

```
implementation 'com.android.support:appcompat-v7:26.1.0'
```

Далее требуется создать класс, который будет зависеть от Activity. Чтобы была возможность подписаться на Lifecycle, класс должен наследовать интерфейс LifecycleObserver :

```
public class MyServer  implements LifecycleObserver {
    private String TAG = MyServer.class.getSimpleName();
    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void connect() {
        Log.d(TAG,"connect");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void disconnect() {
        Log.d(TAG,"disconnect");
    }
}
```

Следует обратить внимание, что интерфейс LifecycleObserver пустой. В нем нет методов типа onStart, onStop и т.п. Таким образом требуется просто отметить в классе MyServer его же собственные методы аннотацией

OnLifecycleEvent и указать, при каком lifecycle-событии метод должен быть вызван.

В нашем случае требуется указать, что метод connect должен вызываться в момент onStart, а метод disconnect - в момент onStop.

Осталось связать экземпляр MyServer и Lifecycle:

```
public class MainActivity extends AppCompatActivity {
    private MyServer myServer;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myServer = new MyServer();
        getLifecycle().addObserver(myServer);
    }
}
```

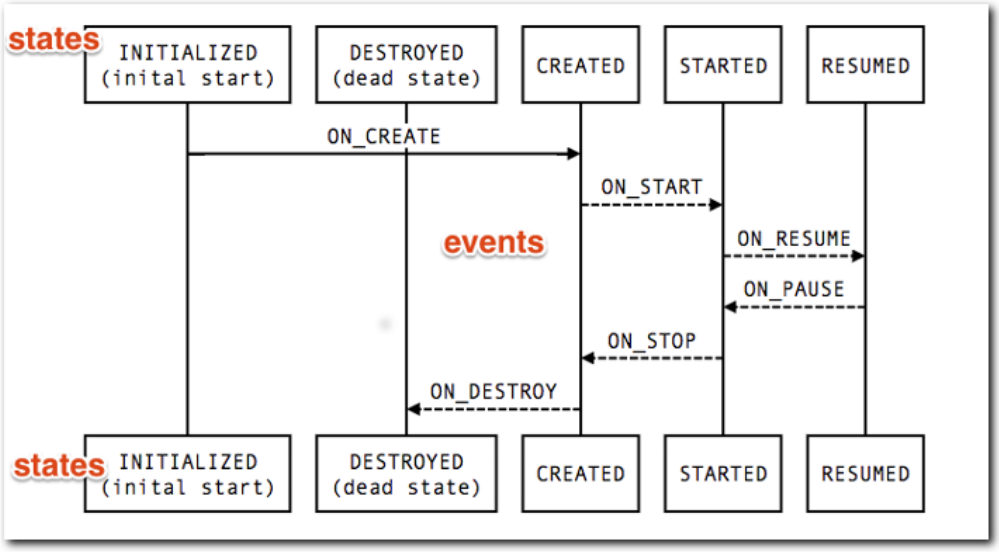
В Activity методом getLifecycle получаем Lifecycle, и методом addObserver подписываем myServer.

А методы onStart и onStop в Activity нам больше не нужны, их можно удалить.

Теперь, при переходе Activity из состояния CREATED в состояние STARTED, его объект Lifecycle вызовет метод myServer.connect. А при переходе из STARTED в CREATED - Lifecycle вызовет myServer disconnect.

При этом в Acivity это потребовало от нас минимум кода - только подписать myServer на Lifecycle. Все остальное решает сам MyServer.

На схеме ниже вы можете увидеть какие состояние проходит Activity и какие события при этом вызываются.



На данной схеме отображены состояния и события. Они связаны следующим образом - при переходе между состояниями происходят события.

Эти события указываются в аннотациях OnLifecycleEvent к методам объекта MyServer.

Добавьте дополнительные методы, чтобы отслеживать все состояния активити.

Запустите программу, попробуйте повернуть экран.

3.1.2 LiveData

Компонент LiveData – хранилище данных, работающее по принципу паттерна Observer (наблюдатель). Возможны следующие операции:

- поместить какой-либо объект для хранения;
- подписаться и получать объекты, которые в него помещают.

Т.е. с одной стороны кто-то помещает объект в хранилище, а с другой стороны кто-то подписывается и получает этот объект.

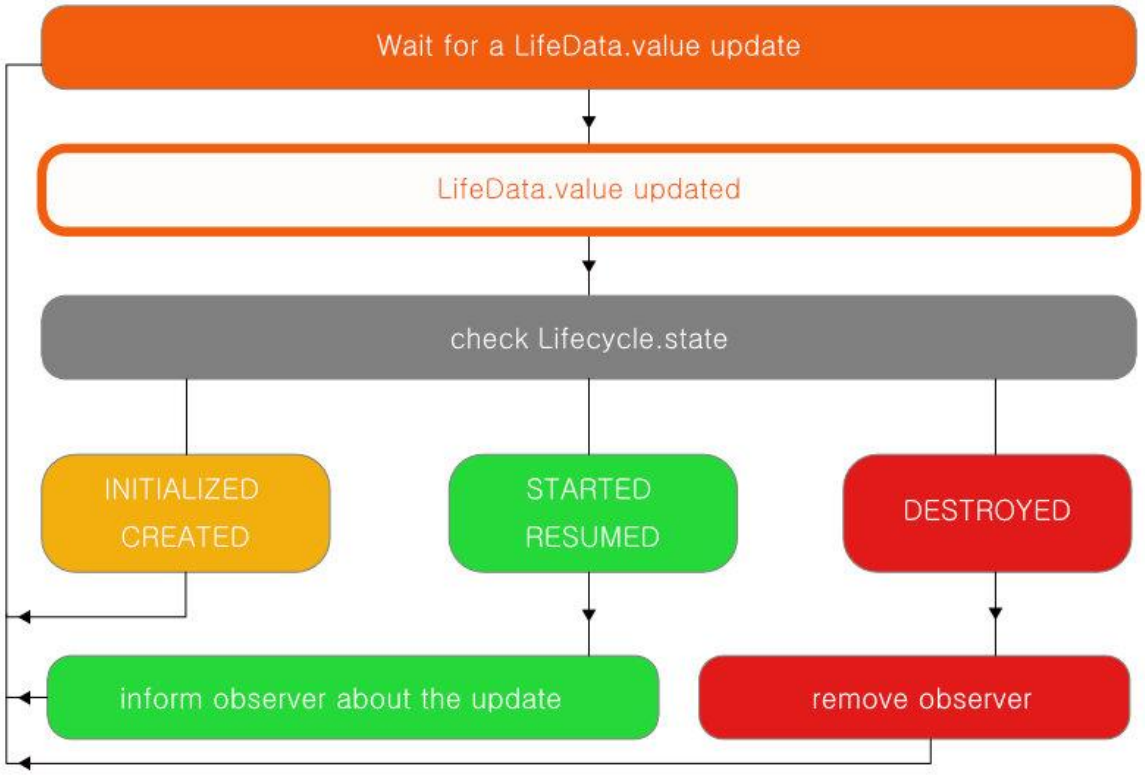
Например, каналы в мессенджерах. Автор пишет пост и отправляет его в канал, а все подписчики получают этот пост.

LiveData умеет определять активен подписчик или нет, и отправлять данные будет только активным подписчикам. Предполагается, что подписчиками LiveData будут Activity и фрагменты. А их состояние активности будет определяться с помощью их Lifecycle объекта.

Рассмотрим следующую схему поведения LiveData:

- если Activity было не активно во время обновления данных в LiveData, то при возврате в активное состояние, его observer получит последнее актуальное значение данных;
- в момент подписки, observer получит последнее актуальное значение из LiveData;
- если Activity будет закрыто, т.е. перейдет в статус DESTROYED, то LiveData автоматически отпишет от себя его observer;
- если Activity в состоянии DESTROYED попробует подписаться, то подписка не будет выполнена;
- если Activity уже подписывало свой observer, и попробует сделать это еще раз, то просто ничего не произойдет;
- всегда возможно получить последнее значение LiveData с помощью его метода getValue.

Как видите, подписывать Activity на LiveData - это удобно. Поворот экрана и полное закрытие Activity - все это корректно и удобно обрабатывается автоматически без каких-либо усилий с нашей стороны.



Сам компонент состоит из классов: *LiveData*, *MutableLiveData*, *MediatorLiveData*, *LiveDataReactiveStreams*, *Transformations* и интерфейса: *Observer*.

Создадим класс для хранения данных.

Наружу мы передаем LiveData, который позволит внешним объектам только получать данные. Но внутри DataController мы используем объект MutableLiveData, который позволяет помещать в него данные.

Чтобы поместить значение в MutableLiveData, используется метод setValue. Этот метод обновит значение LiveData, и все его активные подписчики получат это обновление.

```
public class DataController {
    private static DataController dataControllerInstance;
    private static MutableLiveData<String> liveData = new MutableLiveData<>();

    private DataController(){
    ...}
    public static DataController getInstance(){
      ...
        return dataControllerInstance;
    }

    public LiveData<String> getData() {
        return liveData;
    }

    public void setData(String str) {
        liveData.setValue(str);
    }
}
```

Получаем LiveData из DataController, и методом observe подписываемся. В метод observe нам необходимо передать два параметра:

Первый - это LifecycleOwner. Напомню, что LifecycleOwner - это интерфейс с методом getLifecycle. Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют этот интерфейс, поэтому мы передаем this.

```
dataController = DataController.getInstance();
// ------------------simple button--------------------
LiveData<String> liveData = dataController.getData();
liveData.observe(this, new Observer<String>() {
    @Override
    public void onChanged(@Nullable String value) {
        textView.setText(value);
    }
});
```

LiveData получит из Activity его Lifecycle и по нему будет определять состояние Activity. Активным считается состояние STARTED или RESUMED. Т.е. если Activity видно на экране, то LiveData считает его активным и будет отправлять данные в его колбэк.

Второй параметр - это непосредственно подписчик, т.е. колбэк, в который LiveData будет отправлять данные. В нем только один метод onChanged. В нашем примере туда будет приходить String.

Теперь, когда DataController поместит какой-либо String объект в LiveData, мы сразу получим этот объект в Activity, если Activity находится в состоянии STARTED или RESUMED.

Transformations

Предназначено для изменение типа данных в LiveData.Рассмотрим пример, в котором LiveData<String> будем превращать в LiveData<Integer>:

```
LiveData<Integer> liveDataInt = Transformations.map(liveData, new
Function<String, Integer>() {
    @Override
    public Integer apply(String input) {
        return Integer.parseInt(input);
    }
});
liveDataInt.observe(this, new Observer<Integer>() {
    @Override
    public void onChanged(@Nullable Integer integer) {
        textView.setText("int " + integer);
    }
});
```

В метод map передаем имеющийся LiveData<String> и функцию преобразования. В этой функции мы будем получать String данные из LiveData<String>, и от нас требуется преобразовать их в Integer. В данном случае просто парсим строку в число.

На выходе метода map получим LiveData<Integer>. Можно сказать, что он подписан на LiveData<String> и все получаемые String значения будет конвертировать в Integer и рассылать уже своим подписчикам.

Задание:

Создать новый модуль. В меню File> New> New Module> Phone & Tablet Module> Empty Activity. Приложение назвать livedata.

Отследить изменение сети с помощью livedata.

Для имплементации компонента нам нужно расширить класс LiveData. Для работы нам нужно знать следующие методы из этого класса.

- onActive() – этот метод вызывается когда у нашего экземпляра есть активный(-ые) обсервер. В нем мы должны инициировать интересующий нас сервис или операцию.
- onInactive() – вызывается когда у LiveData нет активных слушателей. Соответственно нужно остановить наш сервис или операцию.
- setValue() – вызываем если изменились данные и LiveData информирует об этом слушателей.
- *postValue(T)* - передать значение из другого потока.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Так как работа будет осуществляться с сетевой часть, в манифесте требуется установить соответствующие разрешения. Давайте рассмотрим, как будет выглядеть LiveData класс, который будет хранить значения состояния подключения к сети и в случае изменения будет его обновлять, для упрощения он реализован как синглтон.

```
public class NetworkLiveData extends LiveData<String> {
    private Context context;
    private BroadcastReceiver broadcastReceiver;
    private static NetworkLiveData instance;

    static NetworkLiveData getInstance(Context context) {
        if (instance == null) {
            instance = new NetworkLiveData(context.getApplicationContext());
        }
        return instance;
    }

    private NetworkLiveData(Context context) {
        if (instance != null) {
            throw new RuntimeException("Use getInstance() method to get the
single instance of this class.");
        }
        this.context = context;
    }

    private void prepareReceiver(Context context) {
        IntentFilter filter = new IntentFilter();
        filter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
        broadcastReceiver = new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                ConnectivityManager cm =
                        (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
                NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
                if (activeNetwork != null) {
                    boolean isConnected =
activeNetwork.isConnectedOrConnecting();
                    setValue(Boolean.toString(isConnected));
                }
                else
                    setValue("false");
            }
        };
        context.registerReceiver(broadcastReceiver, filter);
    }

    @Override
    protected void onActive() {
        prepareReceiver(context);
    }

    @Override
    protected void onInactive() {
        context.unregisterReceiver(broadcastReceiver);
        broadcastReceiver = null;
    }
}
```

Логика класса следующая, если кто-то подписывается, происходит инициализация BroadcastReceiver, который будет нас уведомлять об

изменении сети, после того как отписывается последний подписчик мы перестаем отслеживать изменения сети.

Для того чтобы добавить подписчика есть два метода: observe(LifecycleOwner, Observer<T>) — для добавления подписчика с учетом жизненного цикла и observeForever(Observer<T>) — без учета. Уведомления об изменении данных приходят с помощью реализации интерфейса Observer, который имеет один метод onChanged(T).

Подпишемся на получение сведений об изменении состояния подключения к сети:

```
LiveData<String> networkLiveData =
NetworkLiveData.getInstance(this);
networkLiveData.observe(this, new Observer<String>() {
    @Override
    public void onChanged(@Nullable String value) {
        textView.setText(value);
    }
});
```

Запустите приложение!

3.1.3 ViewModel

Компонент ViewModel — предназначен для хранения и управления данными, связанными с представлением, решает проблемы, связанные с пересозданием активити во время таких операций, как переворот экрана и т.д. Не стоит его воспринимать, как замену onSaveInstanceState, поскольку, после того как система уничтожит активити, к примеру, когда мы перейдем в другое приложение, ViewModel будет также уничтожен и не сохранит свое состояние. В целом же, компонент ViewModel можно охарактеризовать как синглтон с колекцией экземпляров классов ViewModel, который гарантирует, что не будет уничтожен пока есть активный экземпляр активити и освободит ресурсы после ухода с нее. Стоит также отметить, что мы можем привязать любое количество ViewModel к нашей Activity(Fragment). Компонент состоит

из таких классов: *ViewModel*, *AndroidViewModel*, *ViewModelProvider*, *ViewModelProviders*, *ViewModelStore*, *ViewModelStores*. Разработчик будет работать только с ViewModel, AndroidViewModel и для получения инстанса с ViewModelProviders, но для лучшего понимания компонента, мы рассмотрим все классы.

Класс ViewModel, представляет абстрактный класс, без абстрактных методов и с одним protected методом onCleared(). Для реализации собственного ViewModel, нам всего лишь необходимо унаследовать свой класс от ViewModel с конструктором без параметров и это все. Если же нам нужно очистить ресурсы, то необходимо переопределить метод onCleared(), который будет вызван когда ViewModel долго не доступна и должна быть уничтожена. Стоит еще добавить, что во избежания утечки памяти, не нужно ссылаться напрямую на View или Context Activity из ViewModel. В целом, ViewModel должна быть абсолютно изолированная от представления данных. С помощью LiveData происходит уведомление уровнем представления (Activity/Fragment) об изменениях в наших данных.

Задание:

Создать новый модуль. В меню File> New> New Module> Phone & Tablet Module> Empty Activity. Приложение назвать viewmodel.

Добавить в разметку activity_main.xml элемент progressBar.

Все изменяемые данные должны сохранять с помощью LiveData, если же нам необходимо, к примеру, показать и скрыть ProgressBar, мы можем создать MutableLiveData и хранить логику показать\скрыть в компоненте ViewModel. Создадим класс MyViewModel, отвечающий за логику отображения ProgressBar.

```
class MyViewModel extends ViewModel {
    private MutableLiveData<Boolean> showProgress = new
MutableLiveData<>();

    void doSomeThing() {
        showProgress.postValue(true);
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                showProgress.postValue(false);
            }
        }, 10000);
    }
    MutableLiveData<Boolean> getProgressState() {
        return showProgress;
    }
}
```

Для получения ссылки на наш экземпляр ViewModel мы должны воспользоваться ViewModelProviders:

```
public class MainActivity extends AppCompatActivity {
    private ProgressBar progressBar;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        progressBar = findViewById(R.id.progressBar);
        final MyViewModel viewModel =
ViewModelProviders.of(this).get(MyViewModel.class);
        viewModel.getProgressState().observe(this, new
Observer<Boolean>() {
            @Override
            public void onChanged(@Nullable Boolean
isVisibleProgressBar) {
                if (isVisibleProgressBar) {
                   progressBar.setVisibility(View.VISIBLE);
                } else {
                    progressBar.setVisibility(View.GONE);
                }
            }
        });
        viewModel.doSomeThing();
    }
}
```

Запустите приложение! Поверните эмулятор!

Класс AndroidViewModel, являет собой расширение ViewModel, с единственным отличием — в конструкторе должен быть один параметр Application. Является довольно полезным расширением в случаях, когда нам нужно использовать Location Service или другой компонент, требующий Application Context. В работе с ним единственное отличие, это то что мы наследуем наш ViewModel от ApplicationViewModel. В Activity/Fragment инициализируем его точно также, как и обычный ViewModel.
Класс ViewModelProviders, являет собой четыре метода утилиты, которые, называются of и возвращают ViewModelProvider. Адаптированные для работы с Activity и Fragment, а также, с возможностью подставить свою реализацию ViewModelProvider.Factory, по умолчанию используется DefaultFactory, которая является вложенным классом в ViewModelProviders.
Класс ViewModelProvider, класс, возвращающий инстанс ViewModel. В общих чертах класс выполняет роль посредника с ViewModelStore, который, хранит и поднимает наш интанс ViewModel и возвращает его с помощью метода get, который имеет две сигнатуры get(Class) и get(String key, Class modelClass). Смысл заключается в том, что имеется возможность привязать несколько ViewModel к нашему Activity/Fragment даже одного типа. Метод get возвращает их по String key, который по умолчанию формируется как: «android.arch.lifecycle.ViewModelProvider.DefaultKey:» + canonicalName
Класс ViewModelStores, являет собой фабричный метод. На данный момент, в пакете android.arch присутствует как один интерфейс, так и один подкласс ViewModelStore.
Класс ViewModelStore, класс хранит в себе HashMap<String, ViewModel>, методы get и put, соответственно, возвращают по ключу (по тому самому, который мы формируем во ViewModelProvider) или добавляют ViewModel. Метод clear(), вызовет метод onCleared() у всех наших ViewModel которые мы добавляли.

Поведение ViewModel в данном примере:

- activity вызывает метод модели getProgressState();
- модель создает MutableLiveData и стартует асинхронный процесс;

- activity подписывается на получение данных от модели LiveData;
- происходит поворот экрана на модели этот поворот никак не сказывается, она спокойно сидит в провайдере и ждет завершение потока;
- activity пересоздается, получает ту же самую модель от провайдера, получает тот же самый LiveData от модели и подписывается на него и ждет данные;
- поток завершается и устанавливает значение ProgressBar, модель передает иго в MutableLiveData;
- activity получает данные от LiveData.

Context

Не стоит передавать Activity в модель в качестве Context. Это может привести к утечкам памяти. Если в модели требуется объект Context, то вы можете наследовать не ViewModel, а AndroidViewModel.

```
public class MyViewModel extends AndroidViewModel {

    public MyViewModel(@NonNull Application application) {
        super(application);
    }

    public void doSomething() {
        Context context = getApplication();
        // ....
    }
}
```

Передача данных между фрагментами

ViewModel может быть использована для передачи данных между фрагментами, которые находятся в одном Activity.

```
public class SharedViewModel extends ViewModel {
    private final MutableLiveData<Item> selected = new
MutableLiveData<Item>();

    public void select(Item item) {
        selected.setValue(item);
    }

    public LiveData<Item> getSelected() {
        return selected;
    }
}
```

SharedViewModel - модель с двумя методами: один позволяет поместить данные в LiveData, другой - позволяет получить этот LiveData. Соответственно, если два фрагмента будут иметь доступ к этой модели, то один сможет помещать данные в его LiveData, а другой - подпишется и будет получать эти данные. Таким образом два фрагмента будут обмениваться данными ничего не зная друг о друге.

```
public class SharedViewModel extends ViewModel {
    private final MutableLiveData<Item> selected = new
MutableLiveData<Item>();

    public void select(Item item) {
        selected.setValue(item);
    }

    public LiveData<Item> getSelected() {
        return selected;
    }
}
```

Чтобы два фрагмента могли работать с одной и той же моделью, они могут использовать общее Activity. Код получения модели в фрагментах выглядит так:

```
SharedViewModel model = ViewModelProviders
        .of(getActivity()).get(SharedViewModel.class);
```

Для обоих фрагментов getActivity вернет одно и то же Activity. Метод ViewModelProviders.of вернет провайдера этого Activity. Далее методом get получаем модель.

Код фрагментов:

```
public class MasterFragment extends Fragment {
    private SharedViewModel model;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        model =
ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        itemSelector.setOnClickListener(item -> {
            model.select(item);
        });
    }
}

public class DetailFragment extends Fragment {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SharedViewModel model =
ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        model.getSelected().observe(this, { item ->
                // Update the UI.
        });
    }
}
```

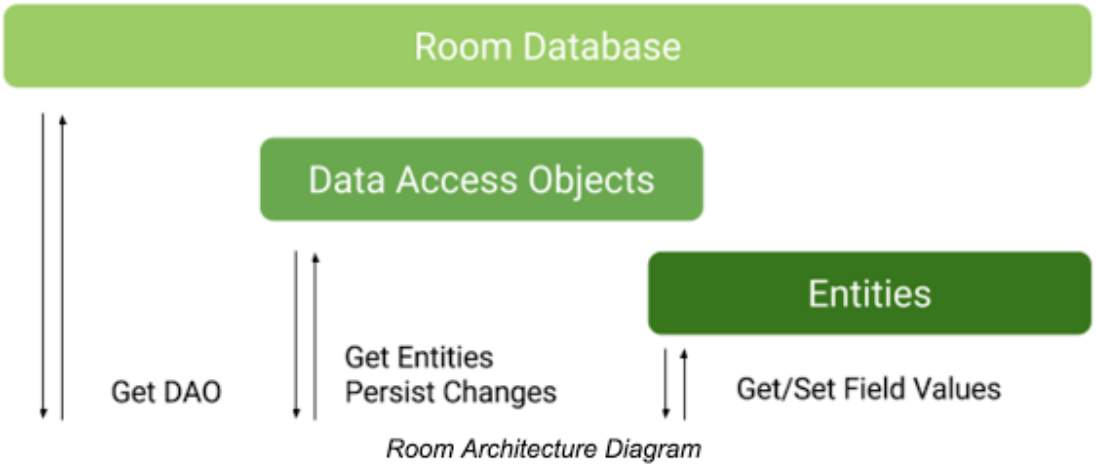
3.1.4 Room

Компонент Room Persistence — предоставляет нам удобную обертку для работы с базой данных SQLite.

В целом же мы получили дефолтную ORM, этот компонент можно разделить на три части: Entity, DAO (Data Access Object), Database.

Entity — объектное представление таблицы. С помощью аннотаций можно легко и без лишнего кода описать наши поля.

Для создания нашей Entity нам нужно создать класс POJO (Plain Old Java Object). Пометить класс аннотацией Entity.



Room Architecture Diagram

Задание:

Создать новый модуль. В меню File> New> New Module> Phone & Tablet Module> Empty Activity. Приложение назвать room

Создать базу данных для хранения данных по сотрудникам.

В build.gradle файле модуля добавьте dependencies:

```
implementation "android.arch.persistence.room:runtime:1.1.1"
annotationProcessor
"android.arch.persistence.room:compiler:1.1.1"
```

Entity

Аннотацией Entity необходимо пометить объект, который мы хотим хранить в базе данных. Для этого создаем класс Employee, который будет представлять собой данные сотрудника: id, имя, зарплата:

```
@Entity
public class Employee {
    @PrimaryKey
    public long id;
    public String name;
    public int salary;
}
```

Класс помечается аннотацией Entity. Объекты класса Employee будут использоваться при работе с базой данных. Например, мы будем получать их от базы при запросах данных и отправлять их в базу при вставке данных.

Этот же класс Employee будет использован для создания таблицы в базе. В качестве имени таблицы будет использовано имя класса. А поля таблицы будут созданы в соответствии с полями класса.

Аннотацией PrimaryKey мы помечаем поле, которое будет ключом в таблице.

В следующих уроках мы рассмотрим возможности Entity более подробно.

Dao

В объекте Dao мы будем описывать методы для работы с базой данных. Нам нужны будут методы для получения списка сотрудников и для добавления/изменения/удаления сотрудников.

Описываем их в интерфейсе с аннотацией Dao.

```
@Dao
public interface EmployeeDao {
    @Query("SELECT * FROM employee")
    List<Employee> getAll();
    @Query("SELECT * FROM employee WHERE id = :id")
    Employee getById(long id);
    @Insert
    void insert(Employee employee);
    @Update
    void update(Employee employee);
    @Delete
    void delete(Employee employee);
}
```

Методы getAll и getById позволяют получить полный список сотрудников или конкретного сотрудника по id. В аннотации Query нам необходимо прописать соответствующие SQL-запросы, которые будут использованы для получения данных.

Обратите внимание, что в качестве имени таблицы мы используем employee. Напомню, что имя таблицы равно имени Entity класса, т.е. Employee, но в SQLite не важен регистр в именах таблиц, поэтому можем писать employee.

Для вставки/обновления/удаления используются методы insert/update/delete с соответствующими аннотациями. Тут никакие запросы указывать не нужно. Названия методов могут быть любыми. Главное - аннотации.

Database

Аннотацией Database помечаем основной класс по работе с базой данных. Этот класс должен быть абстрактным и наследовать RoomDatabase.

```
@Database(entities = {Employee.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract EmployeeDao employeeDao();
}
```

В параметрах аннотации Database указываем, какие Entity будут использоваться, и версию базы. Для каждого Entity класса из списка entities будет создана таблица.

В Database классе необходимо описать абстрактные методы для получения Dao объектов, которые вам понадобятся.

Практика

Создать класс App наследуемый от класса Application и указать его в манифесте:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:name=".App"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
```

Database объект - это стартовая точка. При создании приложения производим инициализацию базы данных в методе onCreate. Используем Application Context, а также указываем AppDatabase класс и имя файла для базы. Учитывайте, что при вызове этого кода Room каждый раз будет создавать новый экземпляр AppDatabase. Эти экземпляры очень тяжелые и рекомендуется использовать один экземпляр для всех ваших операций. Поэтому вам необходимо позаботиться о синглтоне для этого объекта.

```
public class App extends Application {
    public static App instance;

    private AppDatabase database;

    @Override
    public void onCreate() {
        super.onCreate();
        instance = this;
        database = Room.databaseBuilder(this, AppDatabase.class,
"database")
                .build();
    }

    public static App getInstance() {
        return instance;
    }

    public AppDatabase getDatabase() {
        return database;
    }
}
```