ПРАКТИЧЕСКАЯ РАБОТА №8 “ХРАНЕНИЕ ДАННЫХ”

Варианты хранения данных

Работа с SharedPreferences

В Android представлены 4 варианта хранения данных для приложения:

1. Хранилище для конкретного приложения (app-specific storage): в нем хранятся файлы, предназначенные только для использования конкретным приложением.
2. Общее хранилище: храните файлы, которыми ваше приложение намеревается поделиться с другими приложениями, включая мультимедиа, документы и другие файлы.
3. SharedPreferences: хранения данных в виде ключ-значение, доступное только для конкретного приложения.
4. Базы данных: хранение данных в БД используя библиотеку Room.

Работа с app-specific storage:

Если приложению необходимо сохранять данные в файл, то оно может использовать хранение в специально созданной для него папке.

Данную папку можно получить, вызвав метод getFilesDir из класса Context. В этой папке приложение может сохранять данные, доступ к которым будет только у самого приложения. Эта папка может находиться или по пути /data/data/ или /sdcard/Android/data. При этом доступ к папке /sdcard/Android/data, начиная с Android 13 невозможен из других приложений, включая менеджер файлов, что обеспечивает сохранность данных приложения. Взаимодействие с файлами в Android осуществляется также, как и в обычных Java приложениях, с помощью класса File.

Общее хранилище:

Если приложение хочет получить доступ к общим файлам, хранящимся на устройстве, то ему необходимо запросить разрешение READ_EXTERNAL_STORAGE или WRITE_EXTERNAL_STORAGE в зависимости от того, какую задачу необходимо выполнить пользователю. После предоставления разрешения, у приложения будет полный доступ к данным на устройстве. Работа с файлами после предоставления разрешения осуществляется также, как и в обычных Java приложениях, с помощью класса File.

Работа с SharedPreferences

SharedPreferences позволяет хранить данные в виде пар ключ-значение. Такое хранение применяется для сохранения локальных настроек пользователя в приложении. Для получения доступа к файлу, хранящему SharedPreferences необходимо выполнить метод getSharedPreferences из класса Context и указать имя файла и режим доступа.

Например:

```
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key),
Context.MODE_PRIVATE);
```

При именовании SharedPreferences необходимо использовать имя, уникальное для вашего приложения.

Например:

```
"com.example.myapp.PREFERENCE_FILE_KEY"
```

Если в приложении необходим только один файл SharedPreferences, то можно использовать следующий код для создания файла:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
```

Добавление записи в SharedPreferences происходит с помощью SharedPreferences.Editor. Он открывает файл для редактирования и позволяет записать значения с помощью методов putInt, putString, putBoolean. После этого необходимо применить изменения с помощью метода apply. Apply применяет значения сразу, однако запись файла происходит асинхронно. Если необходима синхронная запись, необходимо использовать метод commit. Но, его стоит избегать при выполнении в UI потоке, так как может приводить к заторможенности интерфейса.

Пример:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score_key),
newHighScore);
editor.apply();
```

Чтение значений из SharedPreferences осуществляется с помощью методов getInt, getString, getBoolean, представленных в классе SharedPreferences.

При вызове метода можно указать, какое значение возвращать по умолчанию, если не представлено значение в хранилище.

Пример:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue =
getResources().getInteger(R.integer.saved_high_score_default_key
);
int highScore =
sharedPref.getInt(getString(R.string.saved_high_score_key),
defaultValue);
```

Работа с Room

Room – библиотека, для хранения данных в мобильных приложениях для ОС Android. Она представляет собой абстракцию над БД SQLite и упрощает работу разработчика с БД.

Работа с БД Room

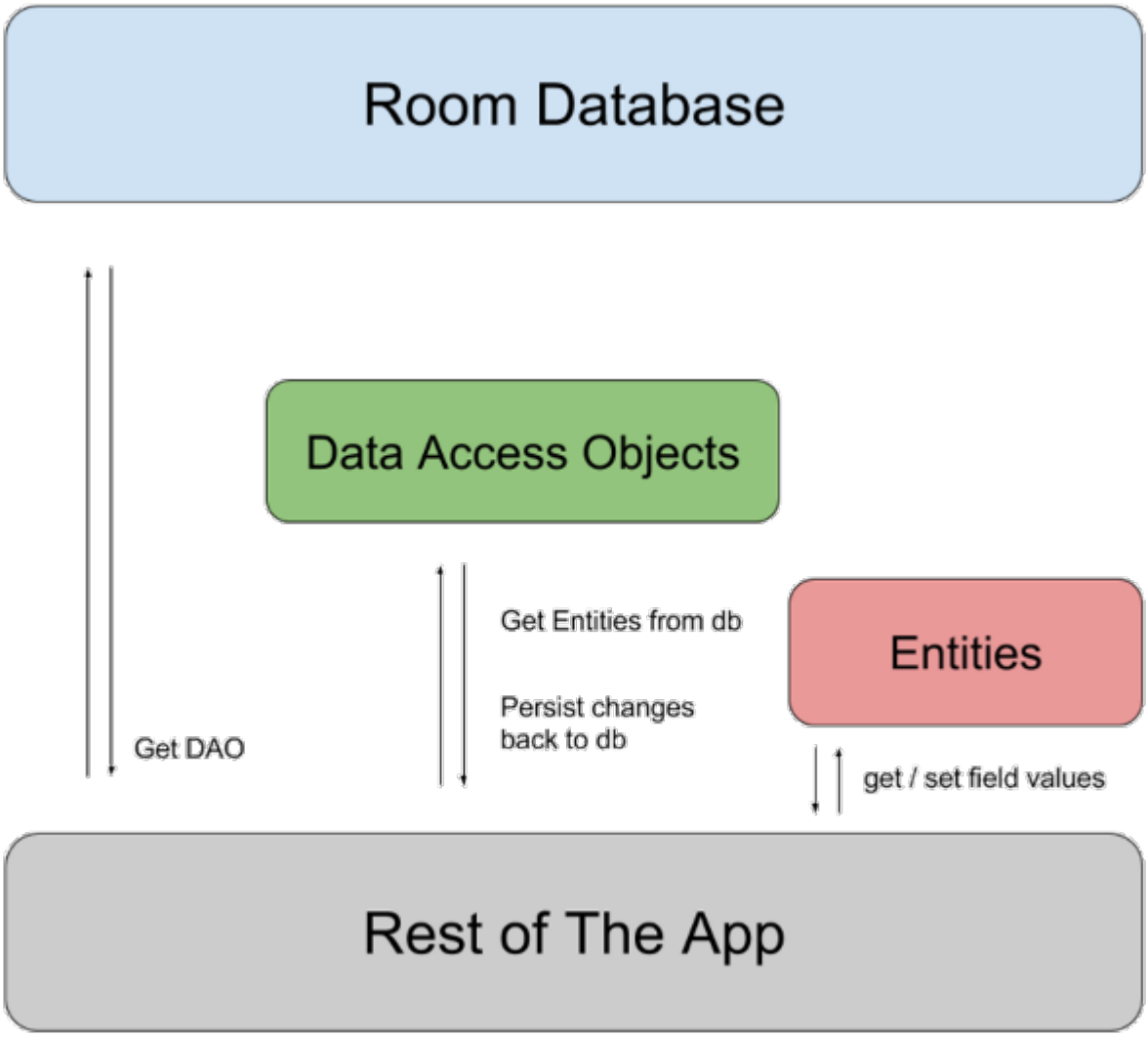

Рисунок 1 – работа с Room DB

Для работы с БД необходимо на уровне Data создать сущности. Сущности помечаются аннотацией @Entity. Они определяют набор полей, в которых будут хранится данные.

Пример:

```
@Entity
public class User {
    @PrimaryKey
    public int uid;
    @ColumnInfo(name = "first_name")
    public String firstName;
    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Для работы с сущностями в БД необходимо описать интерфейсы Data access object (DAO). С их помощью будет осуществляться получение данных

из таблицы, вставка данных, удаление. Интерфейс Dao помечается с помощью аннотации @Dao

Пример:

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();
    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);
    @Query("SELECT * FROM user WHERE first_name LIKE :first AND
" +
           "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);
    @Insert
    void insertAll(User... users);
    @Delete
    void delete(User user);
}
```

Третий класс, который необходим для создания БД – это класс, наследник от RoomDatabase. Он объединяет в себе сущности, которые должны быть в БД и интерфейсы взаимодействия с ними. Класс помечается аннотацией @Database. В аргументах аннотации указываются сущности и версия БД, которая необходима для контроля целостности при обновлении приложения.

Пример создания класса:

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

Далее, после создания сущностей, определения интерфейсов и описания базы данных, можно приступать к использованию базы данных. Для этого необходимо её создать. Создание происходит с помощью Room.databaseBuilder.

Пример:

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

После того, как был получен объект базы данных, можно выполнять запросы к базе данных.

```
UserDao userDao = db.userDao();
List<User> users = userDao.getAll();
```

ДОПОЛНИТЕЛЬНЫЙ МАТЕРИАЛ. ОСНОВЫ ЯЗЫКА SQL

SQL – язык запросов к базе данных. У него есть 4 базовые операции. Это получение (SELECT), удаление (DELETE), обновление (UPDATE) и добавление данных (INSERT).

SELECT

Самый простая и часто употребляемая команда, которая позволяет получить от сервера практически любую информацию из таблиц. Её синтаксис прост:

```
SELECT [имя_поля] FROM имя_таблицы
```

Если дословно: ВЫБРАТЬ поле ИЗ таблицы. Имена полей необходимо указывать через запятую или использовать * для вывода всех полей. В примерах, не забывайте указывать префикс таблиц, если он у вас есть. Вот и вся сложность.

Пример:

```
SELECT type, title FROM node
```

WHERE

```
WHERE поле условие значение
```

Получение данных полей таблиц не составляет большого труда, достаточно знать имена полей (какие в них хранятся логические данные) и таблицы. Но есть одно, но - в большинстве случаев нужны не все записи, а записи, удовлетворяющие определённому условию. Тут на помощь нам приходит выражение WHERE:

```
SELECT [имя_поля] FROM имя_таблицы WHERE поле условие значение
```

К нашему запросу мы добавили условие фильтра, позволяющего из общей массы отобрать только нужные нам данные.

Пример:

```
SELECT name, mail FROM users WHERE uid = 1
```

DELETE

В отличии от команды SELECT, следующие команды приводят к изменениям данных в таблицах. Поэтому будьте внимательны к правильности их ввода. Команда DELETE используется для удаления одной или нескольких строк в таблице. Синтаксис команды прост:

```
DELETE FROM имя_таблиц;
```

Имена столбцов не используются, поскольку применяется для удаления целых строк, а не значений из строк.

UPDATE

Данная инструкция используется для изменения данных в полях таблицы.

```
UPDATE имя_таблицы SET имя_поля = значение
```

Как и в предыдущей команде, тут можно использовать инструкции WHERE, ORDER BY и LIMIT, чтобы ограничить количество и диапазон затрагиваемых строк.

Пример:

```
UPDATE users SET status = 0 WHERE (name like '%sex%') OR (mail
like '%sex%');
```

В любимом нашем примере мы вместо удаления, снимем флаг активности для отфильтрованных пользователей.

INSERT

Инструкция используется для добавления новых данных в таблицу.

```
INSETR INTO имя_таблицы VALUES (значения1), (значения2),...
```

Значения должны быть все (по числу полей в таблице), записаны через запятую, и их порядок должен соответствовать порядку следования столбцов в таблице. Альтернативный синтаксис:

```
INSETR INTO имя_таблицы (имена_полей) VALUES (значения1),
(значения2),...
```

В данном случае порядок значений для полей должен соответствовать перечисленному порядку. Пропущенные поля будут заполнены значениями по умолчанию. И ещё один тип синтаксиса:

```
INSETR INTO имя_таблицы SET имя_поля = значение, …
```

В данном варианте необходимо указывать, через запятую, имена полей и их значения

Пример:

```
INSERT INTO role VALUES (3,"Суперадмин"),(4,"Модератор");
```

Задание

1. Реализовать создание текстового файла в app-specific storage.
2. Реализовать создание текстового файла в общем хранилище.
3. Реализовать сохранение данных в SharedPreferences.
4. Создать БД и реализовать вставку и получение данных из БД