## Хранение данных в базе данных

### Цель работы

Научиться пользоваться встроенной реляционной базой данных для хранения андроид приложения, ознакомиться с работой поставщиков контента и разрешений приложения

### Задания для выполнения

Создайте приложение список дел. На главном окне приложения пользователь может создать дело, введя его название, дедлайн (выбор даты и времени), а также необязательно - категорию. При нажатии кнопки “Добавить” дело записывается в базу данных.

### Методические указания

Разберем простой пример, когда в базу данных записывается всего одно поле. На этом примере мы разберем основные механизмы работы с реляционным хранилищем.

Для начала нам необходимо создать вспомогательный класс, который будет инкапсулировать работу с базой данных. Для этого наследуемся от встроенного класса SQLiteOpenHelper:

**public class** DBHelper **extends** SQLiteOpenHelper {

**public** DBHelper(Context context) {

}

@Override

**public void** onCreate(SQLiteDatabase db) {

}

@Override

**public void** onUpgrade(SQLiteDatabase db, **int** oldVersion, **int** newVersion) {

}

}

Данный класс должен реализовывать как минимум три приведенных метода. Первый метод - это конструктор, он используется при инициализации данного класса. Сейчас нам нужно только вызвать конструктор суперкласса. У конструктора 4 параметра: контекст, название базы данных, фабричный метод (в простых примерах он нам не понадобится) и версия базы данных, которая используется для организации миграций:

**public** DBHelper(Context context) {

**super**(context, **"MyDB"**, **null**, 1);

}

Второй метод используется один раз при создании базы данных. В нем реализуется код, который создает схему данных и при необходимости наполняет базу начальными данными. Обратите внимание, что этот код выполняется только тогда, когда мы пытаемся подключиться к базе, которой еще не существует. Обычно, хорошим тоном при проектировании баз данных является использование специальных контрактных классов, которые абстрагируют реальные названия полей и таблиц, но так как у нас упрощенный пример мы реализуем код сразу по месту:

@Override

**public void** onCreate(SQLiteDatabase db) {

*// создаем таблицу с полями*

db.execSQL(**"create table mytable ("**

+ **"id integer primary key autoincrement,"**

+ **"name text"** + **");"**);

}

То есть это обычная конструкция SQL. В принципе, все остальные операции с базой данных вы тоже можете делать при помощи чистого языка запросов, но это слишком низкоуровневый подход. Большую часть черновой работы фреймворк делает за нас.

И последний метод используется для обновления схемы данных при миграциях. Мы не будем пока использовать эту возможность поэтому оставим данный метод пустым.

После создания вспомогательного класса можно приступить к наполнения макета. Не будем приводить его полностью. В данном примере мы сделаем одно поле ввода и три кнопки, которые будут реализовывать самые базовые операции с БД: запись новой строки в таблицу, чтение всех строк из таблицы и очистка таблицы.

<**EditText**

android:id="@+id/etName"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:ems="10"

android:inputType="textPersonName"

android:text=""

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

**app:layout\_constraintTop\_toTopOf="parent"**/>

<**Button**

android:id="@+id/btnAdd"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Добавить"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintHorizontal\_bias="0.054"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent"

**app:layout\_constraintVertical\_bias="0.717"** />

<**Button**

android:id="@+id/btnDelete"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Удалить"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent"

**app:layout\_constraintVertical\_bias="0.717"** />

<**Button**

android:id="@+id/btnRead"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Получить"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintHorizontal\_bias="0.945"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent"

**app:layout\_constraintVertical\_bias="0.717"** />

Теперь мы можем переходит непосредственно к коду главной активности. Для начала создадим необходимые поля класса:

Button **btnAdd**, **btnDelete**, **btnRead**;

EditText **etName**;

DBHelper **dbHelper**;

Далее как всегда мы получим ссылки на эти элементы и привяжем обработчик событий к кнопкам:

**btnAdd** = findViewById(R.id.***btnAdd***);

**btnDelete** = findViewById(R.id.***btnDelete***);

**btnRead** = findViewById(R.id.***btnRead***);

**etName** = findViewById(R.id.***etName***);

**btnAdd**.setOnClickListener(**this**);

**btnDelete**.setOnClickListener(**this**);

**btnRead**.setOnClickListener(**this**);

Обратите внимание, что заодно мы инстанцировали наш вспомогательный класс:

**dbHelper** = **new** DBHelper(**this**);

Теперь займемся непосредственно обработчиком. В нем мы должны будем получить текст, введенный пользователем, создать вспомогательный объект записи, получить ссылку на саму базу данных и организовать множественный выбор как всегда в таких случаях:

@Override

**public void** onClick(View v) {

String name = **etName**.getText().toString();

ContentValues cv = **new** ContentValues();

SQLiteDatabase db = **dbHelper**.getWritableDatabase();

**switch** (v.getId()){

**case** R.id.***btnAdd***:

**break**;

**case** R.id.***btnDelete***:

**break**;

**case** R.id.***btnRead***:

**break**;

**default**:

**break**;

}

}

Еще в конце этого метода можно закрыт подключение к базе данных для экономии ресурсов. Реализуйте это самостоятельно.

Обратите внимание на вспомогательный объект ContentValues. Они используется для хранения структурированной записи в формате ключ-значение. Именно такая запись и нужна для добавления строки в базу данных. Еще обратите внимание на то, как мы получаем ссылку на саму базу данных - через встроенный метод нашего вспомогательного класса.

Теперь приступим к добавлению строки. Нам остается только создать строку и записать ее в бд в соответствующей ветке:

cv.put(**"name"**, name);

db.insert(**"mytable"**, **null**, cv);

Обратите внимание, что название поля и название таблицы должно соответствовать нашей схеме данных из вспомогательного класса. Именно для того, чтобы не повторяться каждый раз в программе и нужны контрактные классы, однако мы пока обойдемся без них.

Реализуем еще одну простую операцию - удаление данных. Она тоже уже есть в виде встроенных методов, нам только нужно ими воспользоваться:

db.delete(**"mytable"**, **null**, **null**);

У данного метода три параметра: имя таблицы, условия и параметры условий. Условия мы не задаем, что соответствует оператору DELETE без блока WHERE в классическом SQL. На практике, обычно данные из базы удаляются именно по условиям, поэтому посмотрите в документации или в лекциях, как задаются условия. Схема здесь общая и для удаления, и для обновления и для выборки, как и в самом SQL.

Более сложная операция - это чтение из базы. Для этого нам понадобится промежуточный объект - курсор. Именно его возвращает метод запроса, который соответствует оператору SQL SELECT:

Cursor c = db.query(**"mytable"**, **null**, **null**, **null**, **null**, **null**, **null**);

У этого метода довольно много параметров. Давайте перечислим их:

1. Имя таблицы. Это понятно, мы должны указать, из какой таблицы происходит выборка.
2. Колонки. Также мы можем указать, какие именно столбцы будут отбираться. Если, как выше мы не указали имена столбцов, то в результате нам вернуться все.
3. Условия отбора. Здесь мы можем задать условия, то есть как в блоке WHERE. Можно задать одно или несколько условий. Или не задавать вообще, как мы в примере.
4. Параметры условий. Здесь задаются конкретные значения, используемые в условиях отбора. В SQL это выглядит более органично, когда они задаются вместе с самими условиями, но использование фреймворка добавляет своих условностей.
5. Группировка. Также можно задать группировку значений.
6. Условия. Это аналог блока HAVING в SQL. Здесь задаются групповый условия отбора.
7. Сортировка. Также можно задать порядок, то есть один или несколько столбцов, по которым будут отсортированы значения.

И этот сложный и мощный метод возвращает нам специальный объект - курсор, который используется для итерирования по результату отбора. Курсор указывает на определенную строку результата и его можно передвигать на следующую, пока мы не достигнем конца выдачи. Нужно всегда помнить, что такой результат может содержать ноль или более строк. Мы хотим вывести все строки для простоты в консоль, но вообще их можно положить в определенную структуру и связать, например, со списочным представлением в интерфейсе.

Но вначале всегда следует проверить, вернулось ли нам вообще что-нибудь, то есть проверить количество строк. И вот здесь может быть немного сложно. Ведь курсор - это не массив, это именно итератор. И такого понятия как размер у него нет. Поэтому проверка наличия результата проходит немного непривычно:

**if** (c.moveToFirst()) {

} **else**

Log.*d*(**"LOG\_TAG"**, **"0 rows"**);

c.close();

Обратите внимание на использованный метод. Он возвращает истину, если в курсоре еще есть данные. В противном случае мы просто выводим сообщение, что в таблице нет строк. Все дальнейшие действия мы будем производить в положительной ветке данного условия. Ну и после всего условия мы должны закрыть данный курсор.

Теперь мы будем итерировать по курсору и вытаскивать нужные нам данные. И еще одна условность. Доступ к данным в курсоре происходит по номеру столбца. Мы, конечно, можем запомнить какой столбец какой номер имеет, но это непродуктивно. Лучше воспользоваться специальным методом:

**int** idColIndex = c.getColumnIndex(**"id"**);

**int** nameColIndex = c.getColumnIndex(**"name"**);

Осталось организовать цикл по курсору и выбрать необходимые данные:

**do** {

Log.*d*(**"LOG\_TAG"**,

**"ID = "** + c.getInt(idColIndex) +

**", name = "** + c.getString(nameColIndex));

} **while** (c.moveToNext());

Также отметим, что методы доступа к данным являются типизированными, так что вы должны знать тип данных в каждом столбце таблицы, с которым планируете работать.

Основные операции выполнены. Остальные операции с базой данных осуществляются аналогично. Предоставляем вам самостоятельно расширить схему данных и включить в приложение остальные операции работы с базой данных.

### Контрольные вопросы

1. Что такое контрактные классы и как они облегчают работу со схемой базы данных?

Класс контракта – это public final класс, который содержит постоянные определения для URI, имен столбцов, типов MIME и других метаданных о ContentProvider . Он также может содержать static вспомогательные методы для управления URI.

1. Что такое миграции базы данных и как их реализовать?

Миграция – изменение схемы базы данных, причем с поддержкой работоспособности для предыдущего состояния, наподобие контроля версий

1. Какие методы объекта SQLiteDatabase соответствуют другим операторам языка SQL?

SQLiteDatabase.insert() – INSERT

SQLiteDatabase.delete() DELETE

SQLiteDatabase.query() SELECT

1. Зачем нужны контент-провайдеры и чем они отличаются от контент-резолверов?

Контент-провайдер или "Поставщик содержимого" (Content Provider) - это оболочка (wrapper), в которую заключены данные. Если ваше приложение использует базу данных SQLite, то только ваше приложение имеет к ней доступ. Но бывают ситуации, когда данные желательно сделать общими.

ContentProviders используются для абстрагирования базы данных из других частей и действуют как интерфейс между вашей базой данных и UI/другими классами. Вы должны создать свой собственный ContentProvider , чтобы поделиться своими данными приложений с другими приложениями.

ContentResolver используется для выбора правильного ContentProvider на основе ContentUris. ContentUri может выглядеть как

1. Зачем нужно объектно-реляционное отображение?

### Дополнительные задания

1. Модифицируйте программу таким образом, чтобы все действия с базой данных выполнялись в отдельных функциях.
2. (\*) Добавьте в программу возможность динамически просматривать записи в базе данных с помощью RecyclerView. Создайте адаптер и свяжите его с базой данных. Возможно, придется использовать сторонние компоненты для реализации адаптера, поддерживающего курсоры.
3. Создайте на главном окне программы список дел, читаемый из базы данных, а окно добавления дела сделайте второй активностью вызываемой по кнопке.
4. (\*) Реализуйте в своем приложении контент провайдер, который предоставляет доступ другим приложениям к записям по студентам из своей базы данных. Ограничьте доступ к базе данных при помощи разрешения уровня компонента.