**Лекция 11. Поставщики и базы данных**

В этой лекции подробно обсуждается работа с [**базами данных**](https://developer.android.com/training/basics/data-storage/databases.html) и [**поставщиками контента**](https://developer.android.com/guide/topics/providers/content-providers.html) в Android. *Доступ к* поставщикам контента (через загрузчик) обсуждался в [предыдущей лекции](https://info448.github.io/loaders.html#loaders) . В этой лекции мы поговорим о том, как создать собственную базу данных и поставщика для нее.

Базы данных подходят, когда вы хотите хранить структурированные данные *локально* на устройстве (например, не в облаке, что может потребовать дорогостоящих сетевых транзакций, а также учетных записей пользователей), но эти данные имеют больший масштаб или сложность, чем это подходит для XML-файла SharedPreferences, то есть вам нужно хранить больше, чем просто пары «ключ-значение».

Эта лекция ссылается на код, найденный по адресу <https://github.com/info448/lecture11-databases> . Обратите внимание, что хотя стартовый код обращается к User Dictionary устройства, который доступен только в API 22 (Lollipop) *или более ранних версиях* , остальная часть руководства заменяет этого поставщика и будет работать в любой версии Android.

**11.1 Обзор: поставщики и загрузчики**

Как обсуждалось ранее, **поставщик контента** — это абстракция для источника структурированных данных (например, базы данных, но также, возможно, файлов, интернет-ресурсов и т. д.). Он действует как интерфейс для взаимодействия с этими данными, поддерживая разработку при чтении, добавлении, обновлении или удалении данных из источника (например, базовые операции [CRUD](https://en.wikipedia.org/wiki/Create%2C_read%2C_update_and_delete) ).

Ранее мы продемонстрировали, как получить доступ к данным структуры — в частности, к словарю пользователя — через Content Provider с использованием [Loader](https://developer.android.com/guide/components/loaders.html) . Вы можете прочитать этот пример, чтобы ознакомиться с тем, как использовать существующий Provider:

* Приложение отображает ListView, который поддерживается SimpleCursorAdapter. Этот адаптер принимает «Курсор» (представьте себе: «указатель» или итератор — также список данных, загруженных *в память* ) и подключает каждый элемент к представлению на экране. Это представление показывает слово ( WORD) и «частоту»/выдаемость ( FREQUENCY) этого слова.
* Чтобы получить этот список элементов в память из самого хранилища данных (для выполнения операции **чтения** ), мы настраиваем Loader. Загрузчик извлекает 3 «столбца» (атрибута) из хранилища данных: \_ID, WORD, и FREQUENCY; \_IDполе *не* отображается в представлении, но оно необходимо загрузчику для отслеживания отображаемых элементов). Мы не используем никаких критериев выбора или сортировки, хотя могли бы добавить их, если бы захотели.

Загрузчик извлекает ( ***запрашивает*** ) данные из хранилища данных, чтобы загрузить их в память. Затем он сообщает адаптеру, что нужно взять загруженные данные и отобразить их в представлении. Используя загрузчик для этого процесса, мы получаем два преимущества: (1) данные загружаются в память *в фоновом потоке* (не в потоке пользовательского интерфейса) и (2) данные *автоматически перезагружаются* при изменении содержимого хранилища данных.

* Пример также поддерживает операции **создания** и **обновления** (нажатием кнопки «Добавить слово» и отдельных записей слов соответственно). Эти операции работают путем создания объекта ContentValues(похожего на Bundle, но для поставщиков контента), который содержит значения атрибутов для новой записи поставщика. Затем мы используем ContentResolverto insert()или update()эти значения в поставщике (указанном его URI).

Для обзора: Content Provider действует как хранилище данных, абстрагируя информацию в каком-то месте (например, в базе данных). Loader извлекает набор строк и столбцов из этой базы данных и передает его Adapter. Adapter берет подмножество этих строк и столбцов и помещает их в ListView, чтобы они отображались пользователю. Взаимодействие с пользователем позволяет нам добавлять или изменять данные в этой базе данных.

**11.2 Базы данных SQLite**

Поставщики контента могут абстрагировать все виды хранилищ данных (файлы, URL-адреса и т. д.). Они абстрагируют их как *структурированную информацию,* похожую на базу данных… и на самом деле наиболее распространенным типом хранилища, которое они представляют, является [реляционная база данных](https://en.wikipedia.org/wiki/Relational_database_management_system) (в частности, [база данных SQLite](https://en.wikipedia.org/wiki/SQLite) ). Android [поставляется с API](https://developer.android.com/guide/topics/data/data-storage.html#db) для создания и запроса базы данных; эти базы данных хранятся во [*внутреннем хранилище,*](https://developer.android.com/guide/topics/data/data-storage.html#filesInternal) что означает, что каждое приложение может иметь свою собственную частную базу данных (или несколько баз данных, на самом деле)!

Если вы работали с SQLили другой системой реляционных баз данных (например, в курсе INFO 340 iSchool), это взаимодействие покажется вам знакомым. Если вы никогда не работали с базой данных, самое простое объяснение — думать о них как о электронной таблице (например, в файле Excel), где вы манипулируете *строками* данных, заданными набором предопределенных *столбцов* . [SQL](https://en.wikipedia.org/wiki/SQL) (язык структурированных запросов) — это собственный командный язык для работы с этими электронными таблицами; мы увидим несколько примеров таких запросов в этой лекции. SQLite — это «разновидность» (версия) SQL; полную спецификацию SQLite можно найти [здесь](http://sqlite.org/lang.html) . Краткое руководство (позаимствованное у Google) также доступно в репозитории кода.

В этой лекции мы создадим собственную базу данных слов (отдельную от пользовательского словаря, которая будет работать на API 23+), к которой мы сможем получить доступ через Content Provider. Мы просто изменим, *к какому* «хранилищу данных» осуществляется доступ; остальная часть интерфейса приложения останется прежней. Это позволит нам продемонстрировать, как собрать Content Provider с нуля. Мы начнем с настройки базы данных, а затем реализуем , который ContentProviderее абстрагирует.

* Создание базы данных — дело довольно многословное и окольное, хотя оно и не подразумевает много новых концепций.

Шаг к эффективному использованию базы данных в Android — создать класс (например, WordDatabase), который будет действовать как «пространство имен» для различных частей нашей базы данных. Этот класс не будет инстанцирован (и поэтому может даже иметь privateконструктор по умолчанию). Из соображений экономии времени начало класса включено в код начала лекции.

Класс WordDatabaseбудет содержать ряд *констант* :

* DATABASE\_NAMEдля ссылки на имя файла базы данных, хранящегося на устройстве (например, words.db)
* DATABASE\_VERSIONдля ссылки на текущий номер версии схемы нашей базы данных. Это используется больше для поддержки [миграций](https://en.wikipedia.org/wiki/Schema_migration) , например, если мы хотим обновить нашу базу данных позже.

**Класс также включает константы, которые определяют схему** или **контракт** базы данных . Это сделано для того, чтобы другие классы (например, ContentProviderи MainActivity) могли ссылаться на имена столбцов последовательно, не запоминая или даже не зная конкретный текст, который мы используем в базе данных. Это похоже на то, как мы использовали переменную UserDictionary.Words.WORDвместо значения String "word". По соглашению мы определяем эту схему как отдельный *staticвложенный класс* (например, WordEntry), чтобы все было организовано. Этот класс содержит константы для хранения имен столбцов:

static **class** WordEntry **implements** BaseColumns {

 *//class cannot be instantiated*

 **private** WordEntry(){}

 **public** static final String TABLE\_NAME = "words";

 **public** static final String COL\_WORD = "word";

 **public** static final String COL\_COUNT = "count";

}

* Класс реализует [BaseColumns](https://developer.android.com/reference/android/provider/BaseColumns.html), что позволяет ему бесплатно наследовать несколько констант, специфичных для фреймворка, в частности, \_IDпеременную, которую поставщики контента полагаются на базу данных в качестве первичного ключа.
* Мы создаем отдельный вложенный класс для каждой таблицы в базе данных (лист в электронной таблице). Это позволяет нам использовать пространство имен в стиле Java (точечная нотация) для ссылки на различные таблицы в одной базе данных.

После того, как мы определили схему, мы готовы создать и работать с базой данных. Чтобы помочь нам сделать это, мы будем использовать класс под названием [37](https://info448.github.io/databases.html#fn37) . Этот класс предлагает набор методов, помогающих управлять создаваемой и обновляемой базой данных (например, для миграций). В частности, мы создадим *подкласс* , создав еще один вложенный класс, который представляет собой определенный помощник для нашей базы данных.[SQLiteOpenHelper](https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html) SQLiteOpenHelper

Подкласс имеет конструктор, который принимает Context, а затем «передает» имя и версию базы данных родительскому классу.

**public** DatabaseHelper(Context context){

 **super**(context, DATABASE\_NAME, **null**, DATABASE\_VERSION);

}

SQLiteOpenHelperтакже есть два abstractметода (обратные вызовы событий), которые нам необходимо реализовать: что происходит при *создании* базы данных и что происходит при *обновлении* базы данных .

Когда база данных будет создана впервые, нам нужно будет фактически создать таблицу для хранения наших слов. Это включает отправку ей SQLкоманды на создание таблицы! Это было предоставлено как константа в стартовом коде.

**private** static final String CREATE\_TASKS\_TABLE =

 "CREATE TABLE " + WordEntry.TABLE\_NAME + "(" +

 WordEntry.\_ID + " INTEGER PRIMARY KEY AUTOINCREMENT" + ", "+

 WordEntry.COL\_WORD + " TEXT" + ","+

 WordEntry.COL\_COUNT + " INTEGER" +

 ")";

**private** static final String DROP\_TASKS\_TABLE =

 "DROP TABLE IF EXISTS "+ WordEntry.TABLE\_NAME;

* То же самое можно сделать и для удаления таблицы.
* Это единственное, SQLчто вам понадобится в этом уроке!

Мы можем запустить эти SQLоператоры, используя execSQL()метод, вызываемый на [SQLiteDatabase](http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html)объекте, который передается этим обратным вызовам. Обратите внимание, что этот метод запускает «сырой» SQL-запрос (тот, который ничего не возвращает, так что нет SELECT), без каких-либо проверок на предмет атак SQL-инъекций. Но поскольку мы жестко кодируем информацию для запуска, это не проблема. За исключением этой ситуации, вам ***никогда не*** следует использовать этот метод.

* Мы также можем использовать этот insert()метод для добавления некоторых примеров слов в базу данных (аналогично тому, как мы использовали Content Provider) для ясности при тестировании:
* ContentValues sample1 = **new** ContentValues();
* sample1.put(WordEntry.COL\_WORD, "Embiggen");
* sample1.put(WordEntry.COL\_COUNT, 0);

db.insert(WordEntry.TABLE\_NAME, **null**, sample1);

Вторым параметром .insert()является a nullColumnHack, который представляет собой столбец, NULLв который следует явно поместить значение, если вы не указали никаких других значений (поскольку вы не можете вставить пустую строку).

* В onUpdate()обратном вызове мы просто «удалим» таблицу и создадим ее заново (вызвав onCreate()). В производственной системе это будет включать логику миграции.

Если мы хотим взаимодействовать с этой базой данных в MainActivity, мы можем инициализировать объект (который *при необходимости*DatabaseHelper создаст базу данных ), а затем использовать этот помощник для извлечения базы данных, к которой мы хотим выполнить запрос (используя ).getReadableDatabase()

* Обратите внимание, что запрос к базе данных может занять много времени, поэтому нам *не* следует делать это в потоке пользовательского интерфейса… этот пример приведен просто для тестирования.

Проверить правильность настройки нашей базы данных можно двумя способами:

* Мы можем напрямую исследовать базу данных SQLite, которая находится на вашем устройстве, используя adbи sqlite3инструмент. См. [эту ссылку](http://developer.android.com/tools/help/sqlite3.html) для получения более подробной информации.
* $ adb -s emulator-5554 shell
* # sqlite3 /data/data/edu.uw.package.name/databases/words.db
* # sqlite> select \* from words;

# sqlite> .exit

* Мы можем вызвать query()метод на нашем SQLiteDatabaseи вывести результаты. A SQLiteQueryBuilderможет оказать некоторую помощь, если наш запрос будет сложным (например, с JOIN):
* SQLiteQueryBuilder builder = **new** SQLiteQueryBuilder();
* builder.setTables(WordDatabase.WordEntry.TABLE\_NAME); *//set the table to use*
* Cursor results = builder.query(
* db,
* **new** String[] {WordDatabase.WordEntry.COL\_WORD, WordDatabase.WordEntry.COL\_COUNT},
* **null**, **null**, **null**, **null**, **null**); *//5 nulls!*
* **while**(results.moveToNext()) {
* String word = results.getString(results.getColumnIndexOrThrow(WordDatabase.WordEntry.COL\_WORD));
* int freq = results.getInt(results.getColumnIndexOrThrow(WordDatabase.WordEntry.COL\_COUNT));
* Log.v(TAG, "'"+word+"' ("+freq+")");

}

* + Это точно такая же работа по обработке курсора, которая используется при выходе из системы по выбранному элементу, но вместо этого используются наши имена столбцов!
	+ Мы могли бы даже удалить вызов Loader и просто передать этот запрос напрямую Adapter, если бы мы хотели отобразить нашу базу данных в списке.

Вуаля, у нас есть база данных, к которой мы можем обращаться с помощью методов!

**11.3 Реализация ContentProvider**

Мы не хотим делать это создание базы данных и запросы в основном потоке (потому что это может занять некоторое время). И поскольку мы также хотим легко разрешить наше ListViewобновление при изменении базы данных, мы хотели бы иметь возможность использовать Loader для доступа к этой базе данных. Чтобы использовать Loader, нам нужно обернуть базу данных в ContentProvider.

В создании , задействовано много шагов и кода ContentProvider, и большинство из них являются «шаблонными» для большинства баз данных. Настолько, что в документации Google есть [подробный пример кода](https://developer.android.com/guide/topics/providers/content-provider-creating.html#ContentProvider) , из которого вы можете копировать и вставлять по мере необходимости.

Начнем с создания еще одного класса, который расширяет ContentProvider( *понимаете, почему?* ). Поскольку у него будет много абстрактных методов, которые нам нужно будет заполнить, мы можем использовать один из генераторов Android Studio через , New > Other > Content Providerчтобы он нам помог (обычно я говорю не использовать их, но с это ContentProviderне так уж и запутанно).

Нам нужно будет указать [**полномочия**](https://developer.android.com/guide/topics/providers/content-provider-creating.html#ContentURI) для Поставщика. Это действует как уникальное внутреннее «имя» Android для базы данных (чтобы указать, что это такое или кто «владеет» ею). Это «имя», по которому другие смогут ссылаться на нашего конкретного Поставщика. Таким образом, это своего рода имя пакета — и на самом деле мы обычно используем имя пакета с дополнительным, .providerприкрепленным в качестве имени полномочия.

Также обратите внимание, что запись для этого <provider>была добавлена ​​в Manifest, включая имя органа. android:enabledозначает, что Поставщик может быть создан и android:exportedдоступен другим приложениям.

**URI и типы**

Самая важная часть ContentProvider(которая делает его чем-то большим, чем просто вспомогательные методы для базы данных) — это то, как к нему можно получить доступ по определенному [**URI**](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier) . Поэтому первое, что нам нужно сделать, это указать этот URI для нашего провайдера.

* На самом деле мы хотим указать *несколько* URI. Это связано с тем, что каждый фрагмент контента, который мы предоставляем (каждая запись в базе данных!), сам по себе является отдельным *ресурсом* и, таким образом, должен иметь свой собственный URI. Таким образом, нам нужно разработать **схему** для URI, чтобы мы знали, как ссылаться на каждый вид контента, предлагаемого нашим поставщиком.

Разработка схемы URI похожа на разработку структуры URL для веб-сайта; это покажется вам знакомым при указании *маршрутов* для веб-приложения.

Наиболее распространенный подход к URI для поставщиков контента — присваивать каждому предоставляемому ресурсу URI в следующем формате:

content://authority/resource/id

* Этот URI указывает, что это идентификатор конкретного поставщика ( authority), который имеет определенный resourceтип (например, таблица базы данных с информацией), который может иметь определенный ресурс id(например, идентификатор записи в таблице).
* Если убрать, то idбудет ссылаться на всю таблицу или, скорее, на «список» ресурсов. Так что на самом деле у нас есть две разные «категории» URI: весь список и отдельный ресурс в этом списке. Оба имеют одну и ту же «базу», но обрабатывать их нужно будет немного по-разному.
* Более подробную информацию о том, как их структурировать, см. [в разделе «Разработка URI контента» .](https://developer.android.com/guide/topics/providers/content-provider-creating.html#ContentURI)

Мы определим эти URI по частям, используя константы (конечно). Один для *полномочия* , один для *типа ресурса* (который является именем таблицы базы данных, но не обязательно должен быть таковым), и, наконец, общий Content URI (разобранный в Uriобъект):

**public** static final Uri CONTENT\_URI =

 Uri.parse("content://" + AUTHORITY + "/"+WORD\_RESOURCE);

Но нам также нужно обрабатывать оба типа ресурсов: «список» слов и сами отдельные слова. Чтобы включить это, мы собираемся использовать класс, называемый a [UriMatcher](https://developer.android.com/reference/android/content/UriMatcher.html). Этот класс обеспечивает *сопоставление* между URI и фактическим «типом» данных, которые нас интересуют (либо списки, либо объекты слов). Это поможет нам выполнить работу по «маршрутизации», без необходимости самостоятельно анализировать путь URI.

* Мы будем представлять «тип» или «вид» с помощью intконстант (вроде перечислений), что позволит нам легко ссылаться на «какой» вид ресурса мы говорим.
* *//integer values representing each supported resource Uri*
* **private** static final int WORD\_LIST\_URI = 1; *// /words*

**private** static final int WORD\_SINGLE\_URI = 2;*// /words/:id*

* + Так что если вы дадите мне /wordsURI, я смогу сказать, что вас интересует «вид ресурса № 1».

Мы хотим создать static UriMatcherобъект (вроде константы), который можно использовать для сопоставления... но поскольку для его настройки требуется больше одной строки (мы добавляем запись для каждого сопоставления), нам нужно поместить его в staticблок, чтобы весь этот код выполнялся вместе на уровне класса (а не для каждого экземпляра):

**private** static final UriMatcher sUriMatcher; *//for handling Uri requests*

static {

 *//setup mapping between URIs and IDs*

 sUriMatcher = **new** UriMatcher(UriMatcher.NO\_MATCH);

 sUriMatcher.addURI(AUTHORITY, WORD\_RESOURCE, WORD\_LIST\_URI);

 sUriMatcher.addURI(AUTHORITY, WORD\_RESOURCE + "/#", WORD\_SINGLE\_URI);

}

* Обратите внимание на подстановочный знак #, означающий, что «любое число» (после косой черты) будет «соответствовать» этому URI.

Затем мы можем выяснить, какой «вид» задачи, используя UriMatcher#match(uri)метод, который вернет «вид» int, соответствующий заданному Uri.

В качестве примера давайте заполним getType()метод. Цель этого метода — позволить ContentProvider сообщать тому, кто его запрашивает, [тип MIME](https://en.wikipedia.org/wiki/Media_type) (тип носителя) ресурса, к которому обращается URI. Это позволяет программе указать, является ли контент, предоставляемый ContentProvider, изображением, текстом, музыкой или каким-либо другим типом.

* Тип, который мы собираемся вернуть, — это Cursor(список строк в таблице), поэтому мы укажем [для него типы MIME](http://developer.android.com/guide/topics/providers/content-provider-creating.html#TableMIMETypes) :
* **public** String getType(Uri uri) {
* **switch**(sUriMatcher.match(uri)){
* **case** WORD\_LIST\_URI:
* **return** "vnd.android.cursor.dir/"+AUTHORITY+"."+WORD\_RESOURCE;
* **case** WORD\_SINGLE\_URI:
* **return** "vnd.android.cursor.item/"+AUTHORITY+"."+WORD\_RESOURCE;
* **default**:
* **throw** **new** IllegalArgumentException("Unknown URI "+uri);
* }

}

* + vndозначает «специфичный для поставщика» — в данном случае формат, специфичный для Android.

**Методы запроса**

После того, как все URI указаны, мы можем начать отвечать на запросы контента по этим URI. В частности, когда поступает запрос контента по URI, мы собираемся извлечь данные из базы *данных* , которую мы сделали ранее, а затем вернуть эти данные. Мы обрабатываем эти «запросы» с помощью 4 различных методов: query(), insert(), update(), и delete()(отражая операции CRUD, опираясь на стандартные имена запросов SQL). Мы заполним эти методы, чтобы они извлекали и возвращали данные базы данных.

Во-первых, нам нужно получить доступ к базе данных (через помощника), как мы это делали в MainActivity. Мы создадим экземпляр DatabaseHelperв ContentProvider#onCreate()обратном вызове, сохранив этого помощника как переменную экземпляра для ссылки позже. Затем в методах CRUD (которые будут выполняться *в фоновом потоке* ) мы можем вызвать , getWriteableDatabase()чтобы получить доступ к этой базе данных.

Начнем с реализации метода **query()** . По сути, нам нужно сделать тот же запрос, который мы использовали в MainActivity—хотя можно передать дополнительные параметры запроса (например, projection, selection, sortOrder) вместо того, чтобы всегда определять их nullвручную.

Однако нам также необходимо иметь возможность обрабатывать оба типа ресурсов, которые обслуживает наш Поставщик (списки или отдельные слова). Мы можем использовать , UriMatcherчтобы определить, как скорректировать наш запрос: например, используя UriBuilder#appendWhere()метод , чтобы добавить WHEREпредложение к нашему выбору SQL:

**switch**(sUriMatcher.match(uri)){

 **case** WORD\_LIST\_URI: *//all words*

 **break**; *//no change*

 **case** WORD\_SINGLE\_URI: *//single word*

 builder.appendWhere(WordDatabase.WordEntry.\_ID + "=" + uri.getLastPathSegment()); *//restrict to that item*

 **default**:

 **throw** **new** IllegalArgumentException("Unknown URI "+uri);

}

Затем мы просто вернем то, Cursorчто получим в результате запроса.

Но есть еще одна часть. Мы хотим убедиться, что Loader, который считывает данные из нашего Content Provider (который загрузил этот Cursorобъект), уведомлен о любых изменениях в результатах своего запроса. Это позволит Loader «автоматически» запрашивать новый контент, если какие-либо данные *в этом URI* изменятся.

cursor.setNotificationUri(getContext().getContentResolver(), uri);

После этого шага мы можем вернуться к нашему MainActivityи заменить все имена столбцов и URI на наши собственные WordProvider! Перезапустите приложение… и вуаля, мы видим наш собственный список слов!

По сути, мы можем сделать то же самое для поддержки **insert()**и **update()**обеспечения всех наших вариантов использования.

* Используйте UriMatchermake, чтобы отвечать только на правильные URI — вы не можете вставить данные в отдельную запись и не можете обновить весь список.
* **if**(sUriMatcher.match(uri) != WORD\_LIST\_URI) {
* **throw** **new** IllegalArgumentException("Unknown URI "+uri);

}

* Для insert()также можно убедиться, что в базу данных не добавлено ни одной «пустой» записи, и вернуть результат, если вставка прошла успешно:
* **if**(!values.containsKey(WordDatabase.WordEntry.COL\_WORD)){
* values.put(WordDatabase.WordEntry.COL\_WORD, "");
* }
* **if**(!values.containsKey(WordDatabase.WordEntry.COL\_COUNT)){
* values.put(WordDatabase.WordEntry.COL\_COUNT, 0);
* }
* long rowId = db.insert(WordDatabase.WordEntry.TABLE\_NAME, **null**, values);
* **if** (rowId > 0) { *//if successful*
* Uri wordUri = ContentUris.withAppendedId(CONTENT\_URI, rowId);
* getContext().getContentResolver().notifyChange(wordUri, **null**);
* **return** wordUri; *//return the URI for the entry*
* }

**throw** **new** SQLException("Failed to insert row into " + uri);

* Этот update()метод может оказаться несколько неудобным, поскольку нам, по сути, нужно добавить наше idограничение к заданным пользователем аргументам выбора:
* int count;
* **switch** (sUriMatcher.match(uri)) {
* **case** WORD\_LIST\_URI:
* count = db.update(WordDatabase.WordEntry.TABLE\_NAME, values, selection, selectionArgs); *//just pass in params*
* **break**;
* **case** WORD\_SINGLE\_URI:
* String wordId = uri.getLastPathSegment();
* count = db.update(WordDatabase.WordEntry.TABLE\_NAME, values, WordDatabase.WordEntry.\_ID + "=" + wordId *//select by id*
* + (!TextUtils.isEmpty(selection) ? " AND (" + selection + ')' : ""), selectionArgs); *//apply params*
* **break**;
* **default**:
* **throw** **new** IllegalArgumentException("Unknown URI " + uri);
* }
* **if** (count > 0) {
* getContext().getContentResolver().notifyChange(uri, **null**);
* **return** count;
* }

**throw** **new** SQLException("Failed to update row " + uri);

Но в конце концов у нас есть работающий ContentProvider, который поддерживает те же поведения, что и встроенный User Dictionary (ну, за исключением delete()). Теперь мы можем хранить данные в нашей собственной базе данных и легко получать к ним доступ из UI Thread для использования в таких вещах, как ListViews. Это отлично подходит, если вы хотите отслеживать и хранить любую структурированную информацию в своих приложениях.

1. <https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html>[↩](https://info448.github.io/databases.html#fnref37)