**Chapter 7 Data Storage**

In this chapter, we will cover the following topics:

1. Storing simple data
2. Read and writing a text file to internal storage
3. Read and writing a text file to external storage
4. Including resource files in your project
5. Creating and using an SQLite database
6. Accessing data in the background using a Loader
7. Accessing external storage with scoped directories

**Introduction**

Since most applications, big or small, require **saving data** – from **default user selections to user accounts** – Android offers many options. From **saving a simple value to creating full databases** using SQLite, storage options include the following:

* **Shared preferences**: Simple name/value pairs
* **Internal storage**: Data files in private storage
* **External storage**: Data files in private or public storage
* **SQLite database: Private data** (can be made public through a Content Provider)
* **Cloud storage**: Private server or service provider

There are benefits and trade-offs to using internal and external storage. We will **list some of the differences here to help you decide which option best fits your needs**:

**Internal storage:**

* Unlike external storage, internal storage **is always available but generally has less free space**
* Files are **not accessible to the user** (unless the device has root access)
* Files **are automatically deleted when your app is uninstalled** (or with the Clear Cache/Cleanup File option in the App Manager)

**External storage:**

* The device **may not have external storage** or it may be inaccessible (such as when it's connected to a computer)
* **Files are accessible to the user** (and other apps) without requiring root access
* **Files are not deleted when your app is uninstalled** (unless you use getExternalFilesDir() to get **appspecific public storage**)

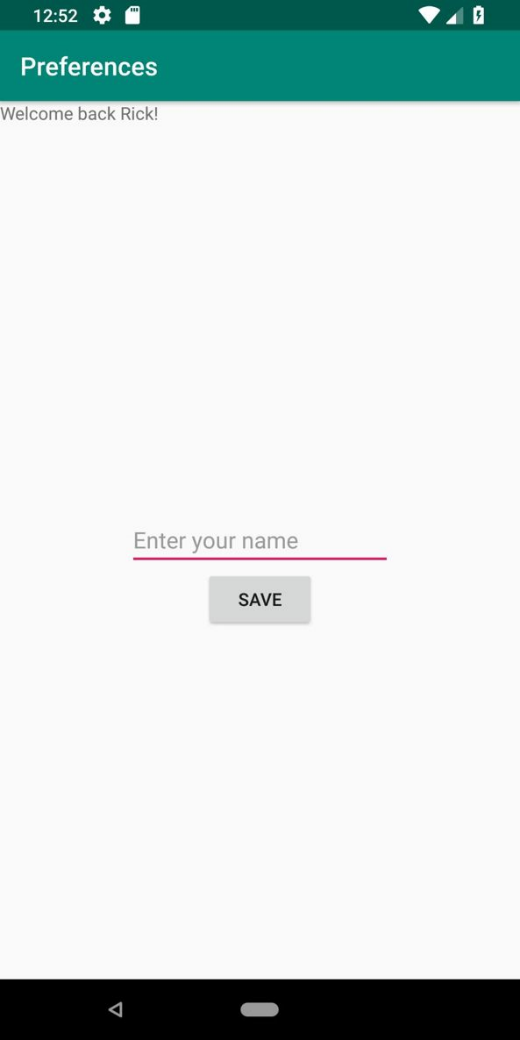
In this chapter, we will demonstrate working with **shared preferences, internal and external storage, and SQLite databases**. For **cloud storage, take a look at the internet recipes in Chapter 12,** Telephony, Networks.

**1 Storing simple data**

It's a common requirement to store simple data, and Android makes it **simple using the Preferences API**. It's not limited to just user preferences either; **you can store any of the primitive data types using a name/value pair.**

We'll demonstrate **saving a name from an EditText and displaying it when the application starts**. The following screenshots shows how the application looks the first time with **no saved name:**

This is an example of how it **looks after saving a name**:

**Getting ready**

Create a new project in Android Studio and call it **Preferences**. Use the default Phone & Tablet options and select Empty Activity in the Add an Activity to Mobile dialog.

**How to do it...**

We'll use the existing TextView to display a Welcome back message and create a new EditText button to save the name. Start by opening **activity\_main.xml**:

1. **Replace the existing TextView** with the following new views:

<TextView

android:id="@+id/textView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

app:layout\_constraintLeft\_toLeftOf="parent"

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

<EditText

android:id="@+id/editTextName"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:ems="10"

android:hint="Enter your name"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

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

<Button

android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Save"

app:layout\_constraintTop\_toBottomOf="@+id/editTextName"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

android:onClick="saveName"/>

2. Open **ActivityMain.java** and add the following global declarations:

**private final String NAME="NAME";**

**private EditText mEditTextName;**

3. Add the following **code to onCreate() to save a reference to EditText and to load any saved name**:

TextView textView = (TextView)findViewById(R.id.textView);

SharedPreferences sharedPreferences = getPreferences(MODE\_PRIVATE);

String name = sharedPreferences.getString(NAME,null);

if (name==null) {

textView.setText("Hello");

} else {

textView.setText("Welcome back " + name + "!");

}

mEditTextName = findViewById(R.id.editTextName);

4. Add the **following saveName() method:**

public void saveName(View view) {

SharedPreferences.Editor editor = getPreferences(MODE\_PRIVATE).edit();

editor.putString(NAME, mEditTextName.getText().toString());

editor.commit();

}

5. **Run the program** on a device or emulator. Since we are demonstrating **persisting data, it loads the name during onCreate()**, so **save a name and restart the program to see it load**.

**How it works...**

To load the name, we first **get a reference to SharedPreference and call the getString() method**. We pass in the key for our name/value pair (**we created a constant called NAME**) and the default value to return **if the key is not found**.

To save the preference, we first need to **get a reference to the Preference Editor**. We use **putString**() with our **NAME** constant and follow it with **commit**(). Without commit(), the change will not be saved.

**There's more...**

Our example **stores all the preferences in a single file**. We can also **store preferences in different files using getSharedPreferences() and passing in the name**. One example where this option could be useful is **you wanted to have separate profiles in a multi-user app**.

**2 Read and write a text file to internal storage**

When simple name/value pairs are not sufficient, **Android also supports regular file operations, including working with text and binary data**.

The following **recipe demonstrates how to read and write a file to internal or private storage**.

**Getting ready**

Create a new project in Android Studio and call it **InternalStorageFile**. Use the default Phone & Tablet options and select Empty Activity in the Add an Activity to Mobile dialog.

**How to do it...**

To demonstrate both **reading and writing text**, we'll need a layout with an **EditText and two buttons**.

Start by opening **main\_activity.xml** and follow these steps:

1. Replace the existing <TextView> element **with the following views**:

<EditText

android:id="@+id/editText"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:inputType="textMultiLine"

android:ems="10"

app:layout\_constraintTop\_toTopOf="parent"

app:layout\_constraintBottom\_toTopOf="@+id/buttonRead"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent" />

<Button

android:id="@+id/buttonRead"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Read"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintBottom\_toTopOf="@+id/buttonWrite"

android:onClick="readFile"/>

<Button

android:id="@+id/buttonWrite"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Write"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintBottom\_toBottomOf="parent"

android:onClick="writeFile"/>

2. Now, open **ActivityMain.java and add the following global variables**:

**private final String FILENAME="testfile.txt";**

**EditText mEditText;**

3. Add the following to the **onCreate() method**, **after setContentView ()**:

**mEditText = (EditText)findViewById(R.id.editText);**

4. Add the **following writeFile() method:**

public void writeFile(View view) {

try {

FileOutputStream fileOutputStream = openFileOutput(FILENAME, Context.MODE\_PRIVATE);

fileOutputStream.write(mEditText.getText().toString().getBytes());

fileOutputStream.close();

} catch (java.io.IOException e) {

e.printStackTrace();

}

}

5. Now, **add the readFile() method**:

public void readFile(View view) {

StringBuilder stringBuilder = new StringBuilder();

try {

InputStream inputStream = openFileInput(FILENAME);

if ( inputStream != null ) {

InputStreamReader inputStreamReader = new InputStreamReader(inputStream);

BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

String newLine = null;

while ((newLine = bufferedReader.readLine()) != null ) {

stringBuilder.append(newLine+"\n");

}

inputStream.close();

}

} catch (java.io.IOException e) {

e.printStackTrace();

}

mEditText.setText(stringBuilder);

}

6. **Run the program** on a device or emulator.

**How it works...**

We use the **InputStream and FileOutputStream classes** to read and write, respectively. Writing to the file **is as simple as getting the text from EditText and calling the write() method**.

Reading back the contents is a little more involved. We could use the **FileInputStream class for reading, but when working with text, the helper classes make it easier**.

In our example, we open the file with **openFileInput(),** which returns an **InputStream object**. We then use **InputStream to get a BufferedReader, which offers the ReadLine() method**. We **loop through each line in the file and append it to our StringBuilder.** When we're finished reading the file, **we assign the text to EditText**.

**There's more...**

The previous example used the **private storage to save the file**. Here's how you can use **the cache folder**.

**Caching files**

If all you need is to **temporarily store data**, you can also use the **cache folder**. The following method **returns the cache folder as a File object** (the next recipe demonstrates working with the File object):

**getCacheDir()**

The main benefit of the cache folder is that **the system can clear the cache if running low on storage space**. (The user can also clear the cache folder from Apps Management in Settings.)

For example, if your app **downloads news** articles, **you could store those in the cache**. When your app starts, **you can display the news already downloaded**.

These are files that are **not required to make your app work**. If the system is low on resources, **the cache can be cleared without adversely affecting your app**. (Even though the system may clear the cache, it's **still a good idea for your app to remove old files as well.**)

**See also**

The next recipe, **Read and write a text file to external storage**.

**3 Read and write a text file to external storage**

The process of reading and writing files to external storage is **basically the same as using internal storage**.

The difference is in **obtaining a reference to the storage location**. Also, as mentioned in the Introduction, external storage may not be available, so it's **best to check availability before attempting to access it**.

This recipe will read and write a text file, as we did in the previous recipe. We'll also demonstrate **how to check the external storage state before we access it**.

**Getting ready**

Create a new project in Android Studio and call it **ExternalStorageFile**. Use the default Phone & Tablet options and select Empty Activity on the Add an Activity to Mobile dialog. **We will use the same layout as the previous recipe, so you can just copy and paste if you typed it in already**.

Otherwise, use the layout from step 1 in the previous recipe, Read and write a text file to internal storage.

**How to do it...**

As mentioned previously in the Getting ready section, we'll use **the layout from the previous recipe**. With the layout file done, the first step will be to **add permission to access the write to external storage. Here are the steps**:

1**. Open the Android Manifest and add the following permission**:

**<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" />**

**<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />**

2. Next, open **ActivityMain.java** and add the following global **variables**:

private final String FILENAME="testfile.txt";

private final String[] PERMISSIONS\_STORAGE = {

Manifest.permission.READ\_EXTERNAL\_STORAGE,

Manifest.permission.WRITE\_EXTERNAL\_STORAGE

};

EditText mEditText;

3. Add the following to the **onCreate() method, after setContentView()**:

**mEditText = (EditText)findViewById(R.id.editText);**

4. Add the **following two methods to check the storage state**:

public boolean isExternalStorageWritable() {

if (Environment.MEDIA\_MOUNTED.equals(Environment.getExternalStorageState())) {

return true;

}

return false;

}

public boolean isExternalStorageReadable() {

if (Environment.MEDIA\_MOUNTED.equals(Environment.getExternalStorageState()) ||

Environment.MEDIA\_MOUNTED\_READ\_ONLY.equals(Environment.getExternalStorageState())) {

return true;

}

return false;

}

5. Add the **following method to verify the app has permission to access the external storage**:

public void checkStoragePermission() {

int permission = ActivityCompat.checkSelfPermission(this,

Manifest.permission.WRITE\_EXTERNAL\_STORAGE);

if (permission != PackageManager.PERMISSION\_GRANTED) {

ActivityCompat.requestPermissions(this, PERMISSIONS\_STORAGE,101);

}

}

6. Add the **following writeFile() method:**

public void writeFile(View view) {

if (isExternalStorageWritable()) {

checkStoragePermission();

try {

File textFile = new File(Environment.getExternalStorageDirectory(), FILENAME);

FileOutputStream fileOutputStream = new FileOutputStream(textFile);

fileOutputStream.write(mEditText.getText().toString().getBytes());

fileOutputStream.close();

} catch (java.io.IOException e) {

e.printStackTrace();

Toast.makeText(this, "Error writing file", Toast.LENGTH\_LONG).show();

}

} else {

Toast.makeText(this, "Cannot write to External Storage", Toast.LENGTH\_LONG).show();

}

}

7. Add the **following readFile()method**:

public void readFile(View view) {

if (isExternalStorageReadable()) {

checkStoragePermission();

StringBuilder stringBuilder = new StringBuilder();

try {

File textFile = new File(Environment.getExternalStorageDirectory(), FILENAME);

FileInputStream fileInputStream = new FileInputStream(textFile);

if (fileInputStream != null ) {

InputStreamReader inputStreamReader = new InputStreamReader(fileInputStream);

BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

String newLine = null;

while ( (newLine = bufferedReader.readLine()) != null ) {

stringBuilder.append(newLine+"\n");

}

fileInputStream.close();

}

mEditText.setText(stringBuilder);

} catch (java.io.IOException e) {

e.printStackTrace();

Toast.makeText(this, "Error reading file", Toast.LENGTH\_LONG).show();

}

} else {

Toast.makeText(this, "Cannot read External Storage",

Toast.LENGTH\_LONG).show();

}

8. **Run the program** on a device or emulator with external storage.

**How it works...**

Reading and writing files are basically the same for both internal and external storage. **The main difference is that we should check for the availability of the external storage before attempting to access it, which we do with** the **isExternalStorageWritable()** and **isExternalStorageReadable() methods**. When checking the storage state, **MEDIA\_MOUNTED means we can read and write to it**.

Unlike the internal storage example, **we request the working path**, which we do in this line of code:

**File textFile = new File(Environment.getExternalStorageDirectory(), FILENAME);**

The actual reading and writing is **done with the same classes**, as it is **just the location that is different**. It is **not safe to hard code an external folder path**. The path can vary between versions of the OS and especially between hardware manufacturers. It is **always best to call getExternalStorageDirectory() as shown**.

**There’s more…**

You probably noticed the **checkStoragePermission() function** from step 5 wasn't mentioned. This is because **permissions aren't specific to storage but are required for the app to access various device features**.

Unlike the previous recipe, which used local app storage, "external" storage **is considered risky for the user**. (It wouldn't be good if just any app could go through a user's private files.)

For that reason, the **app must make additional effort to check if it has the required permission to access storage**. If it does not, the user will be prompted. Note that **this additional dialog is coming from the OS, not the app itself**.

When you first run the app, if you are prompted for permission but still get an error writing, **exit the app and restart**. For a more in-depth explanation and handling of the new Android permission model, see the See also... section.

**Getting public folders**

The **getExternalStorageDirectory()** method **returns the root folder of the external storage**. If you want to obtain specific public folders, such as the **Music or Ringtone folder**, use **getExternalStoragePublicDirectory()** and **pass in the desired folder type, for example**:

**getExternalStoragePublicDirectory(Environment.DIRECTORY\_MUSIC)**

**Checking available space**

One issue consistent between internal and external storage **is limited space**. If you know how much space you will need ahead of time, **you can call the getFreeSpace() method on the File object**.

(**getTotalSpace() will return the total space**.) Here is a simple example, using the call to getFreeSpace():

if (Environment.getExternalStorageDirectory().getFreeSpace() < RQUIRED\_FILE\_SPACE) {

//Not enough space

} else {

//We have enough space

}

**Deleting a file**

There are many helper methods available through the File object, **including deleting a file**. If we wanted to delete the text file we created in the example, **we could call delete() as follows**:

**textFile.delete()**

**Working with directories**

Although it's called a File object, **it supports directory commands as well, such as making and removing directories**. If you want to make or remove a directory, **build the File object, then call the respective methods: mkdir() and delete()**. (There's also a method called mkdirs() (plural) that will create parent folders as well.)

Refer to the link in the See also section for a complete list.

**Preventing files from being included in galleries**

Android employs a media scanner **that automatically includes sound, video, and image files in system collections**, such as the **Image Gallery**. **To exclude your directory, create an empty file called .nomedia (note the preceding period) in the same directory as the files you wish to exclude**.

**See also**

For more information on the Android 6.0 permission model, see the **corresponding recipe in Chapter 15**, Getting Your App Ready for the Play Store.

For a complete **list of methods available in the File class**, visit

<http://developer.android.com/reference/java/io/File.html>

**5 Including resource files in your project**

Android provides **two options for including files in your project: the raw folder and the assets folder**.

Which option you use depends on your requirements. To start, we'll give a **brief overview of each option** to help you decide the best use:

**Raw files**

* Included in the resource directory: **/res/raw**
* As a resource, **accessed through the raw identifier**: **R.raw.<resource>**
* A **good place for storing media files such as MP3, MP4, and OGG files**

**Asset files**

* Creates a **file compiled in your APK** (does not provide a resource ID)
* Access files **using their filenames**, generally making them easier to **use with dynamically created names**
* Some **APIs do not support a Resource Identifier and therefore require including as an Asset**

Generally, **raw files are easier to work with** since they are accessed through the resource identifier. As we'll demonstrate in this recipe**, the main difference is how you access the file**. In this example, we will **load both a raw text file and an asset text file and display the contents**.

**Getting ready**

Create a new project in Android Studio and call it **ReadingResourceFiles**. Use the default Phone & Tablet options and select Empty Activity in the Add an Activity to Mobile dialog.

**How to do it…**

To demonstrate **reading content from both resource locations**, we'll create a **split layout**. We also need to create **both resource folders as they are not included in the default Android project**. Here are the steps:

1. Open **activity\_main.xml** and replace the contents with the following layout:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:id="@+id/textViewRaw"

android:layout\_width="match\_parent"

android:layout\_height="0dp"

android:layout\_weight="1"

android:gravity="center\_horizontal|center\_vertical"/>

<TextView

android:id="@+id/textViewAsset"

android:layout\_width="match\_parent"

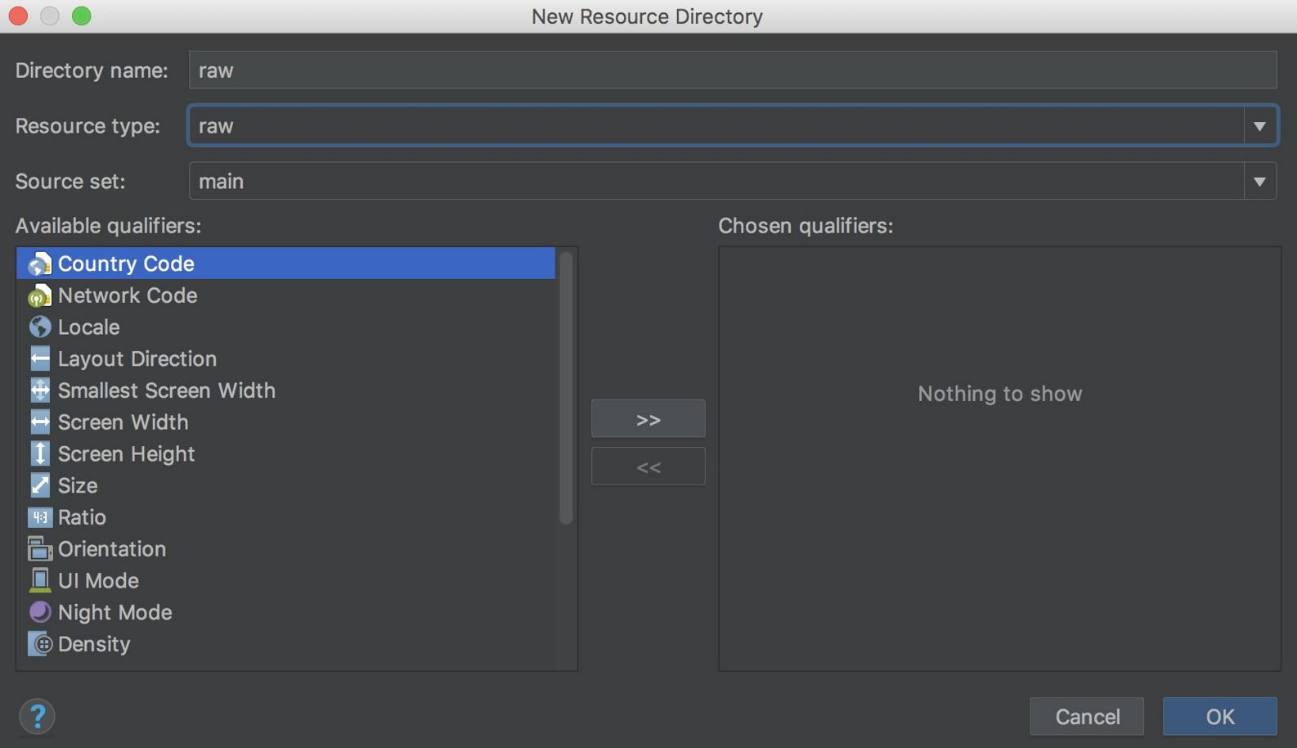
android:layout\_height="0dp"

android:layout\_weight="1"

android:gravity="center\_horizontal|center\_vertical"/>

</LinearLayout>

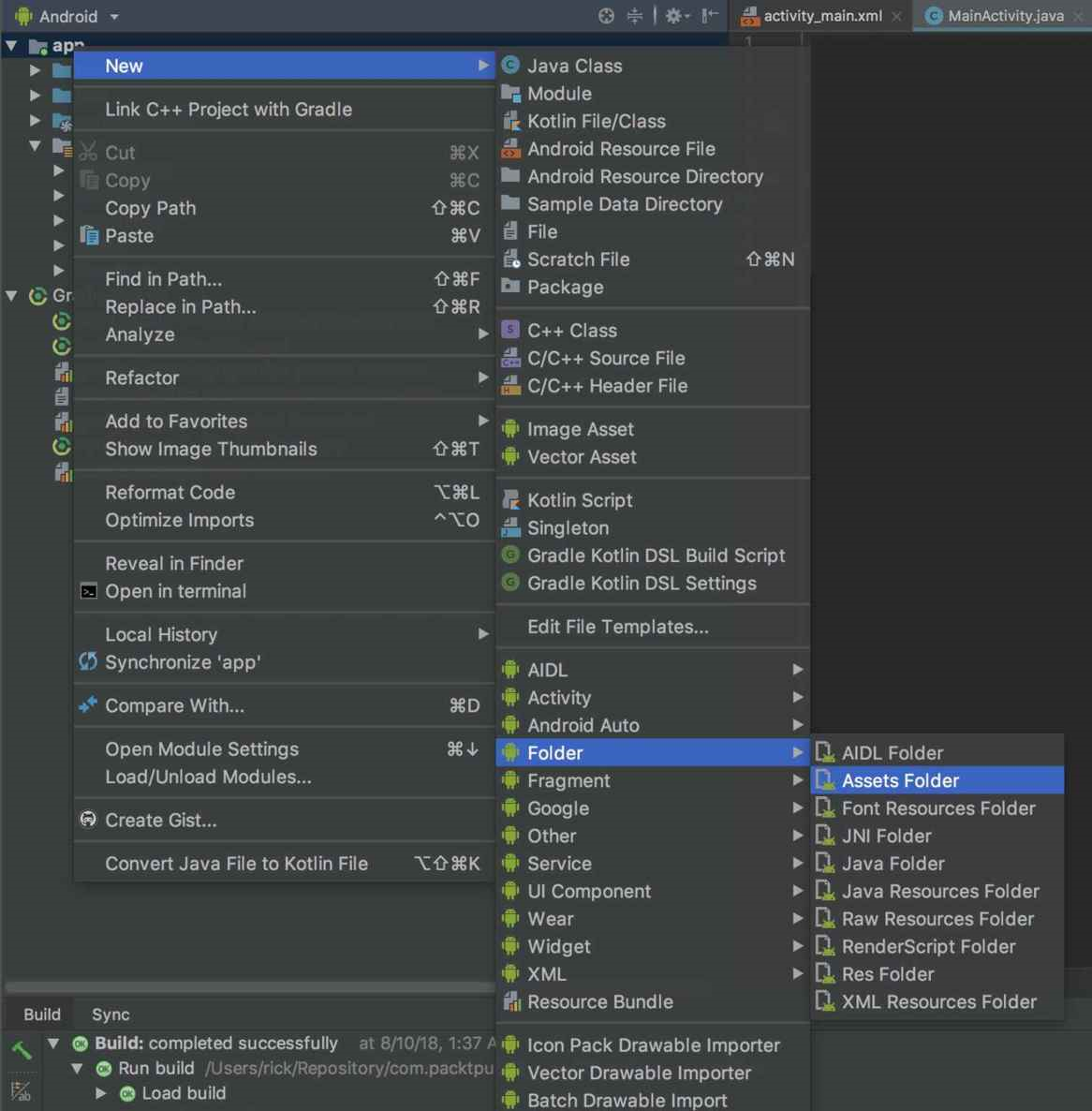
2. Create the **raw resource folder in the res folder**. It will read as follows: res/raw. You can easily create it manually or let Android Studio do it for you by right-clicking on the res folder and selecting **New | Android Resource Directory**. When the Select Resource Directory dialog opens, select raw as the Resource type, as shown in this screenshot:



3. Create **a new text file by right-clicking on the raw folder** and select New | File. Name the file **raw\_text.txt** and type some text in the file. (This text will be displayed when you run the application.)

4. Create **the asset folder. The asset folder is trickier to create manually as it needs to be at the correct folder level**. Fortunately, Android Studio provides a menu option that makes creating it very easy. **Go to the File menu** (or right-click on the app node) and select **New | Folder | Assets**

Folder, as shown in this screenshot:



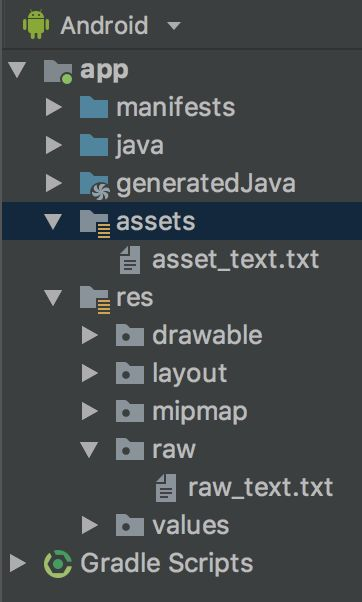
5. Create a **text file in the asset folder called asset\_text.txt.** Again, whatever text you type here will be shown when you run the app. **Here's how the final result should look** after both text files are created:

6. Now, it's time for the code. **Open MainActivity.java** and add the following **method to read the text file** (which is passed into the method):

private String getText(InputStream inputStream) {

StringBuilder stringBuilder = new StringBuilder();

try {;

if ( inputStream != null ) {

InputStreamReader inputStreamReader = new InputStreamReader(inputStream);

BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

String newLine = null;

while ((newLine = bufferedReader.readLine()) != null ) {

stringBuilder.append(newLine+"\n");

}

inputStream.close();

}

} catch (java.io.IOException e) {

e.printStackTrace();

}

return stringBuilder.toString();

}

7. Finally, **add the following code to the onCreate()** method:

TextView textViewRaw = findViewById(R.id.textViewRaw);

textViewRaw.setText(getText(this.getResources().openRawResource(R.raw.raw\_text)));

TextView textViewAsset = findViewById(R.id.textViewAsset);

try {

textViewAsset.setText(getText(this.getAssets().open("asset\_text.txt")));

} catch (IOException e) {

e.printStackTrace();

}

8. **Run the program** on a device or emulator.

**How it works...**

To summarize, **the only difference is in how we get a reference to each file. This line of code reads the raw resource**:

**this.getResources().openRawResource(R.raw.raw\_text)**

And this **code reads the asset file**:

**this.getAssets().open("asset\_text.txt")**

Both calls **return an InputStream, which the getText() method uses to read the file contents**. It is worth noting, though, that the **call to open the asset text file requires an additional try/catch**.

As noted in the recipe introduction, **resources are indexed so we have compile-time verification, which the asset folder does not have.**

**There's more...**

A common approach is to **include resources in your APK**, but **download new resources as they become available**. (See the **network communication in Chapter 13**, Telephony, Networks, and the Web.)

If **new resources aren't available, you can always fall back on the resources in your APK**.

**See also**

**Network communication recipes in Chapter 13**, Telephony, Networks, and the Web.

**5 Creating and using an SQLite database**

In this recipe, we're going to **demonstrate working with an SQLite database**. If you are already familiar with SQL databases from other platforms, then much of what you know will apply.

If you are new to SQLite, take a look at the reference links in the See also section as this recipe assumes a **basic understanding of database concepts, including schemas, tables, cursors, and raw SQL.**

To get you up and running with an SQLite database quickly, our **example implements the basic CRUD operations**. Generally, when creating a database in Android, **you create a class that extends SQLiteOpenHelper**, which is where your database functionality is implemented. Here is a list of the CRUD (create, read, update, and delete) functions:

* **Create: insert()**
* **Read: query() and rawQuery()**
* **Update: update()**
* **Delete: delete()**

To demonstrate a fully working database, we will create a **simple Dictionary database where we'll store words and their definitions.** We'll demonstrate the CRUD operations by **adding new words** (with their definitions) **and updating existing word definitions**. We'll **show words in a ListView using a cursor**.

**Pressing a word in the ListView will read the definition from the database and display it in a Toast message**. **A long press will delete the word.**

**Getting ready**

Create a new project in Android Studio and call it **SQLiteDatabase**. Use the default Phone & Tablet options and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

First, **we'll create the UI, which will consist of two EditText fields, a button, and a ListView**. As we add words to the database, they will **populate the ListView.** Start by opening **activity\_main.xml** and follow these steps:

1. Replace the default XML with the following:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<EditText

android:id="@+id/editTextWord"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentTop="true"

android:layout\_alignParentLeft="true"

android:layout\_alignParentStart="true"

android:hint="Word"/>

<EditText

android:id="@+id/editTextDefinition"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_below="@+id/editTextWord"

android:layout\_alignParentLeft="true"

android:layout\_alignParentStart="true"

android:hint="Definition"/>

<Button

android:id="@+id/buttonAddUpdate"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Save"

android:layout\_alignParentRight="true"

android:layout\_alignParentTop="true" />

<ListView

android:id="@+id/listView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_below="@+id/et\_definition"

android:layout\_alignParentLeft="true"

android:layout\_alignParentBottom="true" />

</LinearLayout>

2. Add a new **Java class to the project named DictionaryDatabase**. This class **extends from SQLiteOpenHelper** and handles all the SQLite functions. Here is the class declaration:

**public class DictionaryDatabase extends SQLiteOpenHelper {**

3. Below the declaration, **add the following constants:**

private static final String DATABASE\_NAME = "dictionary.db";

private static final String TABLE\_DICTIONARY = "dictionary";

private static final String FIELD\_WORD = "word";

private static final String FIELD\_DEFINITION = "definition";

private static final int DATABASE\_VERSION = 1;

4. Add the **following constructor, OnCreate(), and onUpgrade()** methods:

DictionaryDatabase(Context context) {

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

}

@Override

public void onCreate(SQLiteDatabase db) {

db.execSQL("CREATE TABLE " + TABLE\_DICTIONARY +

"(\_id integer PRIMARY KEY," +

FIELD\_WORD + " TEXT, " +

FIELD\_DEFINITION + " TEXT);");

}

@Override

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

//Handle database upgrade as needed

}

5. The following **methods are responsible for creating, updating, and deleting the records**:

public void saveRecord(String word, String definition) {

long id = findWordID(word);

if (id>0) {

updateRecord(id, word,definition);

} else {

addRecord(word,definition);

}

}

public long addRecord(String word, String definition) {

SQLiteDatabase db = getWritableDatabase();

ContentValues values = new ContentValues();

values.put(FIELD\_WORD, word);

values.put(FIELD\_DEFINITION, definition);

return db.insert(TABLE\_DICTIONARY, null, values);

}

public int updateRecord(long id, String word, String definition) {

SQLiteDatabase db = getWritableDatabase();

ContentValues values = new ContentValues();

values.put("\_id", id);

values.put(FIELD\_WORD, word);

values.put(FIELD\_DEFINITION, definition);

return db.update(TABLE\_DICTIONARY, values, "\_id = ?", new String[]{String.valueOf(id)});

}

public int deleteRecord(long id) {

SQLiteDatabase db = getWritableDatabase();

return db.delete(TABLE\_DICTIONARY, "\_id = ?", new String[]{String.valueOf(id)});

}

6. And **these methods handle reading the information from the database**:

public long findWordID(String word) {

long returnVal = -1;

SQLiteDatabase db = getReadableDatabase();

Cursor cursor = db.rawQuery(

"SELECT \_id FROM " + TABLE\_DICTIONARY + " WHERE " + FIELD\_WORD + " = ?",

new String[]{word});

if (cursor.getCount() == 1) {

cursor.moveToFirst();

returnVal = cursor.getInt(0);

}

return returnVal;

}

public String getDefinition(long id) {

String returnVal = "";

SQLiteDatabase db = getReadableDatabase();

Cursor cursor = db.rawQuery(

"SELECT definition FROM " + TABLE\_DICTIONARY + " WHERE \_id = ?",

new String[]{String.valueOf(id)});

if (cursor.getCount() == 1) {

cursor.moveToFirst();

returnVal = cursor.getString(0);

}

return returnVal;

}

public Cursor getWordList() {

SQLiteDatabase db = getReadableDatabase();

String query = "SELECT \_id, " + FIELD\_WORD +

" FROM " + TABLE\_DICTIONARY + " ORDER BY " + FIELD\_WORD +

" ASC";

return db.rawQuery(query, null);

}

7. With the database, class finished, **open MainActivity.java**. Add the following **global variables below the class declaration**:

EditText mEditTextWord;

EditText mEditTextDefinition;

DictionaryDatabase mDB;

ListView mListView;

8. Add the following **method to save the fields** when the button is clicked:

private void saveRecord() {

mDB.saveRecord(mEditTextWord.getText().toString(), mEditTextDefinition.getText().toString());

mEditTextWord.setText("");

mEditTextDefinition.setText("");

updateWordList();

}

9. Add this **method to populate the ListView**:

private void updateWordList() {

SimpleCursorAdapter simpleCursorAdapter = new SimpleCursorAdapter(

this,

android.R.layout.simple\_list\_item\_1,

mDB.getWordList(),

new String[]{"word"},

new int[]{android.R.id.text1},

0);

mListView.setAdapter(simpleCursorAdapter);

}

10. **Finally, add the following code to onCreate():**

mDB = new DictionaryDatabase(this);

mEditTextWord = findViewById(R.id.editTextWord);

mEditTextDefinition = findViewById(R.id.editTextDefinition);

Button buttonAddUpdate = findViewById(R.id.buttonAddUpdate);

buttonAddUpdate.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

saveRecord();

}

});

mListView = findViewById(R.id.listView);

mListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

@Override

public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

Toast.makeText(MainActivity.this, mDB.getDefinition(id), Toast.LENGTH\_SHORT).show();

}

});

mListView.setOnItemLongClickListener(new AdapterView.OnItemLongClickListener() {

@Override

public boolean onItemLongClick(AdapterView<?> parent, View view, int position, long id) {

Toast.makeText(MainActivity.this,

"Records deleted = " + mDB.deleteRecord(id), Toast.LENGTH\_SHORT).show();

updateWordList();

return true;

}

});

updateWordList();

11. **Run the program** on a device or emulator and try it out.

**How it works...**

We'll start by explaining the **DictionaryDatabase class** as that's the heart of an SQLite database. The first item to note is the **constructor**:

DictionaryDatabase(Context context) {

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

}

Notice **DATABASE\_VERSION? Only when you make changes to your database schema do you need to increment this value**.

Next is **onCreate**(), where the database is actually created. This is only called the **first time the database is created**, not each time the class is created.

It's also worth **noting the \_id field**. Android does not require tables to have a primary field, **but some classes, such as the SimpleCursorAdapter, may require an \_id**.

We're required to implement the **onUpgrade()** callback, but as this is a new database, there's nothing to do. This method will only be called **when the database version is incremented**.

The **saveRecord**() method handles calling **addRecord**() or **updateRecord**(), **as appropriate**. Since we are going to modify the database, both methods use **getWritableDatabase() to get an updatable database reference.** A writable database requires more resources so if you don't need to make changes, get a read-only database instead.

The last method to note is **getWordList(),** which **returns all the words in the database using a cursor object**. We use this cursor to **populate the ListView**, which brings us to ActivityMain.java.

The onCreate() method **does the standard initialization** we've seen before and also creates an instance of the database with the following line of code:

**mDB = new DictionaryDatabase(this);**

The onCreate() method is also **where we set up the events to show the word definition (with a Toast) when an item is pressed** and to **delete the word on a long press**. Probably the most complicated code is in updateWordList().

This isn't the first time we've used an adapter, but this is the **first cursor adapter**, so we'll explain. We use the **SimpleCursorAdapter to create a mapping between our field in the cursor and the ListView item**.

We use the **layout.simple\_list\_item\_1 layout,** which only includes a **single text field with the ID android.R.id.text1**. In a real application, **we'd probably create a custom layout and include the definition in the ListView item**, but we wanted to demonstrate a method to read the definition from the database.

We call **updateWordList() in three places: during onCreate()** to create the initial list, **then again after we add/update an item**, **and lastly when deleting an item**.

**There's more...**

Although this is a fully functioning example of SQLite, it is still just the basics. **There are many books dedicated to SQLite for Android and they are worth checking out**.

**Upgrading a database**

As we mentioned previously, **when we increment the database version, the onUpgrade() method will be called**. What you do here is dependent on the changes made to the database.

If you **changed an existing table, ideally you'll want to migrate the user data to the new format by querying the existing data and inserting it into the new format**.

Keep in mind that **there is no guarantee the user will upgrade in consecutive order, so they could jump from version 1 to version 4, for example**.

**See also**

SQLite homepage: <https://www.sqlite.org/>

SQLite database Android reference:

<http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>

**6 Accessing data in the background using a Loader**

Any **potentially long-running operations should not be done on the UI thread**, as this can cause your application to be slow or become unresponsive. The Android OS will bring up the **Application Not Responding (ANR) dialog** when apps become unresponsive.

Since querying databases can be time-consuming, **Android introduced the Loader API in Android 3.0**. A Loader **processes the query on a background thread and notifies the UI thread when it finishes**.

**The two primary benefits to Loaders** are the following:

* **Querying the database is (automatically) handled on a background thread**
* **The query auto-updates (when using a Content Provider data source)**

To demonstrate a Loader, **we will modify the previous SQLite** **database** example **to use a CursorLoader to populate ListView**.

**Getting ready**

We will use the project from the previous example, Creating and using an SQLite database, as the base for this recipe. Create a new project in Android Studio and call it **Loader**.

Use the default Phone & Tablet options and select Empty Activity on the Add an Activity to Mobile dialog. **Copy the DictionaryDatabase class and the layout from the previous recipe**.

Although we will use parts of the previous ActivityMain.java code, **we will start at the beginning in this recipe to make it easier to follow**.

**How to do it...**

With the project set up as described in Getting ready, we'll continue by creating **two new Java classes, and then tie it all together in ActivityMain.java. Here are the steps**:

1. Create a new Java class called **DictionaryAdapter** that extends **CursorAdapter**. This class replaces the SimpleCursorAdapater used in the previous recipe. Here is the full code:

public class DictionaryAdapter extends CursorAdapter {

public DictionaryAdapter(Context context, Cursor c, int flags) {

super(context, c, flags);

}

@Override

public View newView(Context context, Cursor cursor, ViewGroup parent) {

return LayoutInflater.from(context)

.inflate(android.R.layout.simple\_list\_item\_1,parent, false);

}

@Override

public void bindView(View view, Context context, Cursor cursor) {

TextView textView = view.findViewById(android.R.id.text1);

textView.setText(cursor.getString(getCursor().getColumnIndex("word")));

}

}

2. Next, create another new Java class and call this one **DictionaryLoader. Although this is the class that handles the data loading on the background thread**, it's actually very simple:

public class DictionaryLoader extends CursorLoader {

Context mContext;

public DictionaryLoader(Context context) {

super(context);

mContext = context;

}

@Override

public Cursor loadInBackground() {

DictionaryDatabase db = new DictionaryDatabase(mContext);

return db.getWordList();

}

}

3. Next, **open ActivityMain.java**. We need to change the declaration to **implement the LoaderManager.LoaderCallbacks<Cursor> interface** as follows:

**public class MainActivity extends AppCompatActivity implements LoaderManager.LoaderCallbacks<Cursor> {**

4. **Add the adapter to the global declarations**. The complete list is as follows:

EditText mEditTextWord;

EditText mEditTextDefinition;

DictionaryDatabase mDB;

ListView mListView;

DictionaryAdapter mAdapter;

5. Change **onCreate() to use the new adapter and add a call to update the Loader after deleting a record**. The final **onCreate()** method should look as follows:

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

mDB = new DictionaryDatabase(this);

mEditTextWord = findViewById(R.id.editTextWord);

mEditTextDefinition = findViewById(R.id.editTextDefinition);

Button buttonAddUpdate = findViewById(R.id.buttonAddUpdate);

buttonAddUpdate.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

saveRecord();

}

});

mListView = findViewById(R.id.listView);

mListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

@Override

public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

Toast.makeText(MainActivity.this,

mDB.getDefinition(id),

Toast.LENGTH\_SHORT).show();

}

});

mListView.setOnItemLongClickListener(new AdapterView.OnItemLongClickListener() {

@Override

public boolean onItemLongClick(AdapterView<?> parent, View view, int position, long id) {

Toast.makeText(MainActivity.this, "Records deleted = " + mDB.deleteRecord(id),

Toast.LENGTH\_SHORT).show();

getSupportLoaderManager().restartLoader(0, null, MainActivity.this);

return true;

}

});

getSupportLoaderManager().initLoader(0, null, this);

mAdapter = new DictionaryAdapter(this,mDB.getWordList(),0);

mListView.setAdapter(mAdapter);

}

6. We **no longer have the updateWordList() method**, **so change saveRecord()** as follows:

private void saveRecord() {

mDB.saveRecord(mEditTextWord.getText().toString(), mEditTextDefinition.getText().toString());

mEditTextWord.setText("");

mEditTextDefinition.setText("");

getSupportLoaderManager().restartLoader(0, null, MainActivity.this);

}

7. Finally, **implement these three methods for the Loader interface**:

@Override

public Loader<Cursor> onCreateLoader(int id, Bundle args) {

return new DictionaryLoader(this);

}

@Override

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {

mAdapter.swapCursor(data);

}

@Override

public void onLoaderReset(Loader<Cursor> loader) {

mAdapter.swapCursor(null);

}

8. **Run the program** on a device or emulator.

**How it works...**

The default CursorAdapter **requires a Content Provider URI**. Since we are accessing the **SQLite database directly** (and not through a Content Provider), **we don't have a URI to pass**, so instead, we **created a custom adapter by extending the CursorAdapter class**.

**DictionaryAdapter still performs the same functionality as SimpleCursorAdapter** from the previous recipe, namely **mapping the data from the cursor to the item layout**.

The next class we added **was DictionaryLoader, which handles populating the adapter**. As you can see, it's actually very simple. **All it does is return the cursor from getWordList()**.

The key here is that **this query is being handled in a background thread and will call the onLoadFinished() callback** (in MainActivity.java) **when it finishes**. Fortunately, most of the heavy lifting is handled in the base class.

This takes us to ActivityMain.java, **where we implemented the following three callbacks from the LoaderManager.LoaderCallbacks interface**:

* **onCreateLoader():** It's initially called in onCreate() with the **initLoader**() call. It's called again with the **restartLoader**() call after we make changes to the database.
* **onLoadFinished**(): It's called when the Loader **loadInBackground() finishes**.
* **onLoaderReset**(): It's called when the **Loader is being recreated (such as with the restart() method**).

**We set the old cursor to null** because it will be invalidated and we don't want a reference kept around.

**There's more...**

As you saw in the previous example, **we need to manually notify the Loader to re-query the database using restartLoader().** One of the benefits of using a Loader is that **it can auto-update**, but it **requires a Content Provider as the data source**.

A **Content Provider supports using an SQLite database as the data source and is recommended for a serious application**. (See the following Content Provider link to get started.)

**See also**

The **AsyncTask recipe in Chapter 14**, Location and Using Geofencing.

Creating a Content Provider:

<http://developer.android.com/guide/topics/providers/content-provider-creating.html>.

It's also worth checking out **Paging and LiveData in the Android Jetpack Components**: <https://developer.android.com/jetpack/>

**The Loader (and AsyncTask) are both included in the Android SDK**.

**A non-SDK option (and highly recommended) is RXJava for Android**: <https://github.com/ReactiveX/RxAndroid>

RXJava is gaining popularity on Android and we're seeing more and more support for RXJava observables.

**7 Accessing external storage with scoped directories in Android N**

With security awareness on the rise, **users are becoming more skeptical about allowing apps to have unnecessary permissions**.

Android N introduces a **new option called Scoped Directory Access,** **allowing your application to request access to only the required permissions**, **instead of general access to all folders**.

If your application **requests READ\_EXTERNAL\_STORAGE and/or WRITE\_EXTERNAL\_STORAGE permission,** but only needs access to a specific directory, **you can use Scoped Directory access instead**.

This recipe will demonstrate how to request access to a specific directory, the Music folder in this case.

**Getting ready**

Create a new project in Android Studio and call it **ScopedDirectoryAccess**. In the Target Android Device dialog, be sure to select API 24: Android 7.0 (Nougat) or higher for the Phone & Tablet option. Select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

To initiate the user access request, we'll add a button to the layout. **Start by opening activity\_main.xml** and follow these steps:

1. **Replace the existing TextView with this button XML**:

<Button

android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Request Access"

android:onClick="onAccessClick"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

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

2. Now, **open MainActivity.java** and add the following line of code to the class:

**private final int REQUEST\_FOLDER\_MUSIC=101;**

3. Add the **method to handle the button click**:

public void onAccessClick(View view) {

StorageManager storageManager = (StorageManager)getSystemService(Context.STORAGE\_SERVICE);

StorageVolume storageVolume = storageManager.getPrimaryStorageVolume();

Intent intent = storageVolume.createAccessIntent(Environment.DIRECTORY\_MUSIC);

startActivityForResult(intent, REQUEST\_FOLDER\_MUSIC);

}

4. **Override the onActivityResult() method** as follows:

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

super.onActivityResult(requestCode, resultCode, data);

switch (requestCode) {

case REQUEST\_FOLDER\_MUSIC:

if (resultCode == Activity.RESULT\_OK) {

getContentResolver().takePersistableUriPermission(data.getData(), 0);

}

break;

}

}

5. You're ready **to run the application on a device** or emulator.

**How it works...**

The **access request is handled by the OS, not by the app**. To request access, **we need to call createAccessIntent()**, which we do with this line of code:

**Intent intent = storageVolume.createAccessIntent(Environment.DIRECTORY\_MUSIC);**

We **call the Intent using the startActivityForResult() method,** which we’ve used before. Since we are looking for a result to come back, **we need to pass a unique identifier to know when the returned result callback is from our request**. (The onActivityResult() callback method can receive callbacks for multiple requests.)

If the **request code matches our request**, we then **check whether the result code equals Activity.RESULT\_OK,** which means the user **granted the permission request**. We pass the result to **takePersistableUriPermission() so we will not need to prompt the user the next time we need to access the same directory**. Access to a directory also includes access to all sub-directories.

**There's more...**

For the best user experience, **observe the following best practices**:

1. Make sure **to persist the URI after the user grants permission** to avoid repeatedly requesting the same permission (as we do with takePersistableUriPermission())
2. If **the user denies the permission request**, don't annoy your users by continuously asking

**See also**

See the following link for **more information on the Storage Access** Framework: <http://developer.android.com/guide/topics/providers/document-provider.html>