Content Providers

Key classes

1. ContentProvider
2. ContentResolver
3. Cursor

In this document

1. Content provider basics
2. Querying a content provider
3. Modifying data in a provider
4. Creating a content provider
5. Content URI summary

Content providers store and retrieve data and make it accessible to all applications. They're the only way to share data across applications; there's no common storage area that all Android packages can access.

Android ships with a number of content providers for common data types (audio, video, images, personal contact information, and so on). You can see some of them listed in the android.provider package. You can query these providers for the data they contain (although, for some, you must acquire the proper permission to read the data).

If you want to make your own data public, you have two options: You can create your own content provider (a ContentProvider subclass) or you can add the data to an existing provider — if there's one that controls the same type of data and you have permission to write to it.

This document is an introduction to using content providers. After a brief discussion of the fundamentals, it explores how to query a content provider, how to modify data controlled by a provider, and how to create a content provider of your own.

Content Provider Basics

How a content provider actually stores its data under the covers is up to its designer. But all content providers implement a common interface for querying the provider and returning results — as well as for adding, altering, and deleting data.

It's an interface that clients use indirectly, most generally through ContentResolver objects. You get a ContentResolver by calling getContentResolver() from within the implementation of an Activity or other application component:

ContentResolver cr = getContentResolver();
You can then use the ContentResolver's methods to interact with whatever content providers you're interested in.

When a query is initiated, the Android system identifies the content provider that's the target of the query and makes sure that it is up and running. The system instantiates all ContentProvider objects; you never need to do it on your own. In fact, you never deal directly with ContentProvider objects at all. Typically, there's just a single instance of each type of ContentProvider. But it can communicate with multiple ContentResolver objects in different applications and processes. The interaction between processes is handled by the ContentResolver and ContentProvider classes.

The data model

Content providers expose their data as a simple table on a database model, where each row is a record and each column is data of a particular type and meaning. For example, information about people and their phone numbers might be exposed as follows:

<table>
<thead>
<tr>
<th>_ID</th>
<th>NUMBER</th>
<th>NUMBER_KEY</th>
<th>LABEL</th>
<th>NAME</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>(425) 555</td>
<td>6677</td>
<td>Kirkland office</td>
<td>Bully Pulpit</td>
<td>TYPE_WORK</td>
</tr>
<tr>
<td>44</td>
<td>(212) 555-1234</td>
<td>1234</td>
<td>NY apartment</td>
<td>Alan Vain</td>
<td>TYPE_HOME</td>
</tr>
<tr>
<td>45</td>
<td>(212) 555-6657</td>
<td>6657</td>
<td>Downtown office</td>
<td>Alan Vain</td>
<td>TYPE_MOBILE</td>
</tr>
<tr>
<td>53</td>
<td>201.555.4433</td>
<td>4433</td>
<td>Love Nest</td>
<td>Rex Cars</td>
<td>TYPE_HOME</td>
</tr>
</tbody>
</table>

Every record includes a numeric _ID field that uniquely identifies the record within the table. IDs can be used to match records in related tables — for example, to find a person's phone number in one table and pictures of that person in another.

A query returns a Cursor object that can move from record to record and column to column to read the contents of each field. It has specialized methods for reading each type of data. So, to read a field, you must know what type of data the field contains. (There's more on query results and Cursor objects later.)

URIs

Each content provider exposes a public URI (wrapped as a Uri object) that uniquely identifies its data set. A content provider that controls multiple data sets (multiple tables) exposes a separate
URI for each one. All URIs for providers begin with the string "content://". The content: scheme identifies the data as being controlled by a content provider.

If you're defining a content provider, it's a good idea to also define a constant for its URI, to simplify client code and make future updates cleaner. Android defines CONTENT_URI constants for all the providers that come with the platform. For example, the URI for the table that matches phone numbers to people and the URI for the table that holds pictures of people (both controlled by the Contacts content provider) are:

```java
android.provider.Contacts.Phones.CONTENT_URI
android.provider.Contacts.Photos.CONTENT_URI
```

Similarly, the URIs for the table of recent phone calls and the table of calendar entries are:

```java
android.provider.CallLog.Calls.CONTENT_URI
android.provider.Calendar.CONTENT_URI
```

The URI constant is used in all interactions with the content provider. Every ContentResolver method takes the URI as its first argument. It's what identifies which provider the ContentResolver should talk to and which table of the provider is being targeted.

**Querying a Content Provider**

You need three pieces of information to query a content provider:

- The URI that identifies the provider
- The names of the data fields you want to receive
- The data types for those fields

If you're querying a particular record, you also need the ID for that record.

**Making the query**

To query a content provider, you can use either the `ContentResolver.query()` method or the `Activity.managedQuery()` method. Both methods take the same set of arguments, and both return a Cursor object. However, `managedQuery()` causes the activity to manage the life cycle of the Cursor. A managed Cursor handles all of the niceties, such as unloading itself when the activity pauses, and requerying itself when the activity restarts. You can ask an Activity to begin managing an unmanaged Cursor object for you by calling `Activity.startManagingCursor()`.

The first argument to either `query()` or `managedQuery()` is the provider URI — the CONTENT_URI constant that identifies a particular ContentProvider and data set (see URIs earlier).
To restrict a query to just one record, you can append the _ID value for that record to the URI — that is, place a string matching the ID as the last segment of the path part of the URI. For example, if the ID is 23, the URI would be:

    content://. . . ./23

There are some helper methods, particularly ContentUris.withAppendedId() and Uri.withAppendedPath(), that make it easy to append an ID to a URI. Both are static methods that return a Uri object with the ID added. So, for example, if you were looking for record 23 in the database of people contacts, you might construct a query as follows:

```java
import android.provider.Contacts.People;
import android.content.ContentUris;
import android.net.Uri;
import android.database.Cursor;

// Use the ContentUris method to produce the base URI for the contact with _ID == 23.
Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);

// Alternatively, use the Uri method to produce the base URI. // It takes a string rather than an integer.
Uri myPerson = Uri.withAppendedPath(People.CONTENT_URI, "23");

// Then query for this specific record:
Cursor cur = managedQuery(myPerson, null, null, null, null);
```

The other arguments to the query() and managedQuery() methods delimit the query in more detail. They are:

- The names of the data columns that should be returned. A null value returns all columns. Otherwise, only columns that are listed by name are returned. All the content providers that come with the platform define constants for their columns. For example, the android.provider.Contacts.Phones class defines constants for the names of the columns in the phone table illustrated earlier — _ID, NUMBER, NUMBER_KEY, NAME, and so on.
- A filter detailing which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). A null value returns all rows (unless the URI limits the query to a single record).
- Selection arguments.
- A sorting order for the rows that are returned, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). A null value returns the records in the default order for the table, which may be unordered.

Let's look at an example query to retrieve a list of contact names and their primary phone numbers:

```java
import android.provider.Contacts.People;
import android.database.Cursor;

// Form an array specifying which columns to return.
```
String[] projection = new String[] {
    People._ID,
    People._COUNT,
    People.NAME,
    People.NUMBER
};

// Get the base URI for the People table in the Contacts content provider.
Uri contacts = People.CONTENT_URI;

// Make the query.
Cursor managedCursor = managedQuery(contacts,
    projection, // Which columns to return
    null,       // Which rows to return (all rows)
    null,       // Selection arguments (none)
    // Put the results in ascending order by name
    People.NAME + " ASC");

This query retrieves data from the People table of the Contacts content provider. It gets the name, primary phone number, and unique record ID for each contact. It also reports the number of records that are returned as the _COUNT field of each record.

The constants for the names of the columns are defined in various interfaces — _ID and _COUNT in BaseColumns, NAME in PeopleColumns, and NUMBER in PhoneColumns. The Contacts.People class implements each of these interfaces, which is why the code example above could refer to them using just the class name.

**What a query returns**

A query returns a set of zero or more database records. The names of the columns, their default order, and their data types are specific to each content provider. But every provider has an ID column, which holds a unique numeric ID for each record. Every provider can also report the number of records returned as the _COUNT column; its value is the same for all rows.

Here is an example result set for the query in the previous section:

<table>
<thead>
<tr>
<th>_ID</th>
<th>_COUNT</th>
<th>NAME</th>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>3</td>
<td>Alan Vain</td>
<td>212 555 1234</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>Bully Pulpit</td>
<td>425 555 6677</td>
</tr>
<tr>
<td>53</td>
<td>3</td>
<td>Rex Cars</td>
<td>201 555 4433</td>
</tr>
</tbody>
</table>

The retrieved data is exposed by a Cursor object that can be used to iterate backward or forward through the result set. You can use this object only to read the data. To add, modify, or delete data, you must use a ContentResolver object.
**Reading retrieved data**

The Cursor object returned by a query provides access to a recordset of results. If you have queried for a specific record by ID, this set will contain only one value. Otherwise, it can contain multiple values. (If there are no matches, it can also be empty.) You can read data from specific fields in the record, but you must know the data type of the field, because the Cursor object has a separate method for reading each type of data — such as `getString()`, `getInt()`, and `getFloat()`.(However, for most types, if you call the method for reading strings, the Cursor object will give you the String representation of the data.) The Cursor lets you request the column name from the index of the column, or the index number from the column name.

The following snippet demonstrates reading names and phone numbers from the query illustrated earlier:

```java
import android.provider.Contacts.People;

private void getColumnData(Cursor cur){
    if (cur.moveToFirst()) {
        String name;
        String phoneNumber;
        int nameColumn = cur.getColumnIndex(People.NAME);
        int phoneColumn = cur.getColumnIndex(People.NUMBER);
        String imagePath;

        do {
            // Get the field values
            name = cur.getString(nameColumn);
            phoneNumber = cur.getString(phoneColumn);

            // Do something with the values.
            ...
        } while (cur.moveToNext());
    }
}
```

If a query can return binary data, such as an image or sound, the data may be directly entered in the table or the table entry for that data may be a string specifying a `content:` URI that you can use to get the data. In general, smaller amounts of data (say, from 20 to 50K or less) are most often directly entered in the table and can be read by calling `Cursor.getBlob()`. It returns a byte array.

If the table entry is a `content:` URI, you should never try to open and read the file directly (for one thing, permissions problems can make this fail). Instead, you should call `ContentResolver.openInputStream()` to get an `InputStream` object that you can use to read the data.

**Modifying Data**
Data kept by a content provider can be modified by:

- Adding new records
- Adding new values to existing records
- Batch updating existing records
- Deleting records

All data modification is accomplished using `ContentResolver` methods. Some content providers require a more restrictive permission for writing data than they do for reading it. If you don't have permission to write to a content provider, the `ContentResolver` methods will fail.

**Adding records**

To add a new record to a content provider, first set up a map of key-value pairs in a `ContentValues` object, where each key matches the name of a column in the content provider and the value is the desired value for the new record in that column. Then call `ContentResolver.insert()` and pass it the URI of the provider and the `ContentValues` map. This method returns the full URI of the new record — that is, the provider's URI with the appended ID for the new record. You can then use this URI to query and get a Cursor over the new record, and to further modify the record. Here's an example:

```java
import android.provider.Contacts.People;
import android.content.ContentResolver;
import android.content.ContentValues;

ContentValues values = new ContentValues();

// Add Abraham Lincoln to contacts and make him a favorite.
values.put(People.NAME, "Abraham Lincoln");
// 1 = the new contact is added to favorites
// 0 = the new contact is not added to favorites
values.put(People.STARRED, 1);

Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
```

**Adding new values**

Once a record exists, you can add new information to it or modify existing information. For example, the next step in the example above would be to add contact information — like a phone number or an IM or e-mail address — to the new entry.

The best way to add to a record in the Contacts database is to append the name of the table where the new data goes to the URI for the record, then use the amended URI to add the new data values. Each Contacts table exposes a name for this purpose as a `CONTENT_DIRECTORY` constant. The following code continues the previous example by adding a phone number and e-mail address for the record just created:

```java
Uri phoneUri = null;
Uri emailUri = null;
```
// Add a phone number for Abraham Lincoln. Begin with the URI for
// the new record just returned by insert(); it ends with the _ID
// of the new record, so we don't have to add the ID ourselves.
// Then append the designation for the phone table to this URI,
// and use the resulting URI to insert the phone number.
phoneUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);

values.clear();
values.put(People.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);

// Now add an email address in the same way.
emailUri = Uri.withAppendedPath(uri,
People.ContactMethods.CONTENT_DIRECTORY);

values.clear();
// ContactMethods.KIND is used to distinguish different kinds of
// contact methods, such as email, IM, etc.
values.put(People.ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(People.ContactMethods.DATA, "test@example.com");
values.put(People.ContactMethods.TYPE, People.ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);

You can place small amounts of binary data into a table by calling the version of
ContentValues.put() that takes a byte array. That would work for a small icon-like image or a
short audio clip, for example. However, if you have a large amount of binary data to add, such as
a photograph or a complete song, put a content: URI for the data in the table and call
ContentResolver.openOutputStream() with the file's URI. (That causes the content provider
to store the data in a file and record the file path in a hidden field of the record.)

In this regard, the MediaStore content provider, the main provider that dispenses image, audio,
and video data, employs a special convention: The same URI that is used with query() or
managedQuery() to get meta-information about the binary data (such as, the caption of a
photograph or the date it was taken) is used with openInputStream() to get the data itself.
Similarly, the same URI that is used with insert() to put meta-information into a MediaStore
record is used with openOutputStream() to place the binary data there. The following code
snippet illustrates this convention:

import android.provider.MediaStore.Images.Media;
import android.content.ContentValues;
import java.io.OutputStream;

// Save the name and description of an image in a ContentValues map.
ContentValues values = new ContentValues(3);
values.put(Media.DISPLAY_NAME, "road_trip_1");
values.put(Media.DESCRIPTION, "Day 1, trip to Los Angeles");
values.put(Media.MIME_TYPE, "image/jpeg");

// Add a new record without the bitmap, but with the values just set.
// insert() returns the URI of the new record.
Uri uri = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, values);
Now get a handle to the file for that record, and save the data into it. Here, sourceBitmap is a Bitmap object representing the file to save to the database.

```java
try {
    OutputStream outStream = getContentResolver().openOutputStream(uri);
    sourceBitmap.compress(Bitmap.CompressFormat.JPEG, 50, outStream);
    outStream.close();
} catch (Exception e) {
    Log.e(TAG, "exception while writing image", e);
}
```

### Batch updating records

To batch update a group of records (for example, to change "NY" to "New York" in all fields), call the `ContentResolver.update()` method with the columns and values to change.

### Deleting a record

To delete a single record, call `ContentResolver.delete()` with the URI of a specific row.

To delete multiple rows, call `ContentResolver.delete()` with the URI of the type of record to delete (for example, `android.provider.Contacts.People.CONTENT_URI`) and an SQL WHERE clause defining which rows to delete. *(Caution: Be sure to include a valid WHERE clause if you're deleting a general type, or you risk deleting more records than you intended!)*

### Creating a Content Provider

To create a content provider, you must:

- Set up a system for storing the data. Most content providers store their data using Android's file storage methods or SQLite databases, but you can store your data any way you want. Android provides the `SQLiteOpenHelper` class to help you create a database and `SQLiteDatabase` to manage it.
- Extend the `ContentProvider` class to provide access to the data.
- Declare the content provider in the manifest file for your application (AndroidManifest.xml).

The following sections have notes on the last two of these tasks.

### Extending the ContentProvider class

You define a `ContentProvider` subclass to expose your data to others using the conventions expected by ContentResolver and Cursor objects. Principally, this means implementing six abstract methods declared in the ContentProvider class:

```java
query()
insert()
```
The `query()` method must return a `Cursor` object that can iterate over the requested data. Cursor itself is an interface, but Android provides some ready-made Cursor objects that you can use. For example, `SQLiteCursor` can iterate over data stored in an SQLite database. You get the Cursor object by calling any of the `SQLiteDatabase` class's `query()` methods. There are other Cursor implementations — such as `MatrixCursor` — for data not stored in a database.

Because these ContentProvider methods can be called from various ContentResolver objects in different processes and threads, they must be implemented in a thread-safe manner.

As a courtesy, you might also want to call `ContentResolver.notifyChange()` to notify listeners when there are modifications to the data.

Beyond defining the subclass itself, there are other steps you should take to simplify the work of clients and make the class more accessible:

- Define a `public static final Uri` named `CONTENT_URI`. This is the string that represents the full `content: URI` that your content provider handles. You must define a unique string for this value. The best solution is to use the fully-qualified class name of the content provider (made lowercase). So, for example, the URI for a TransportationProvider class could be defined as follows:

  ```java
  public static final Uri CONTENT_URI = 
  Uri.parse("content://com.example.codelab.transporationprovider");
  ```

  If the provider has subtables, also define `CONTENT_URI` constants for each of the subtables. These URIs should all have the same authority (since that identifies the content provider), and be distinguished only by their paths. For example:

  ```java
  content://com.example.codelab.transporationprovider/train
  content://com.example.codelab.transporationprovider/air/domestic
  content://com.example.codelab.transporationprovider/air/international
  ```

  For an overview of `content:` URIs, see the [Content URI Summary](#) at the end of this document.

- Define the column names that the content provider will return to clients. If you are using an underlying database, these column names are typically identical to the SQL database column names they represent. Also define `public static String` constants that clients can use to specify the columns in queries and other instructions.

  Be sure to include an integer column named "_id" (with the constant `_ID`) for the IDs of the records. You should have this field whether or not you have another field (such as a
URL) that is also unique among all records. If you're using the SQLite database, the _ID field should be the following type:

```plaintext
INTEGER PRIMARY KEY AUTOINCREMENT
```

The AUTOINCREMENT descriptor is optional. But without it, SQLite increments an ID counter field to the next number above the largest existing number in the column. If you delete the last row, the next row added will have the same ID as the deleted row. AUTOINCREMENT avoids this by having SQLite increment to the next largest value whether deleted or not.

- Carefully document the data type of each column. Clients need this information to read the data.
- If you are handling a new data type, you must define a new MIME type to return in your implementation of `ContentProvider.getType()`. The type depends in part on whether or not the content: URI submitted to `getType()` limits the request to a specific record. There's one form of the MIME type for a single record and another for multiple records. Use the Uri methods to help determine what is being requested. Here is the general format for each type:
  - For a single record:
    ```plaintext
    vnd.android.cursor.item/vnd.yourcompanyname.contenttype
    ```
    For example, a request for train record 122, like this URI,
    ```plaintext
    content://com.example.transportationprovider/trains/122
    ```
    might return this MIME type:
    ```plaintext
    vnd.android.cursor.item/vnd.example.rail
    ```
  - For multiple records:
    ```plaintext
    vnd.android.cursor.dir/vnd.yourcompanyname.contenttype
    ```
    For example, a request for all train records, like the following URI,
    ```plaintext
    content://com.example.transportationprovider/trains
    ```
    might return this MIME type:
    ```plaintext
    vnd.android.cursor.dir/vnd.example.rail
    ```
- If you are exposing byte data that's too big to put in the table itself — such as a large bitmap file — the field that exposes the data to clients should actually contain a content: URI string. This is the field that gives clients access to the data file. The record should also have another field, named "_data" that lists the exact file path on the device for that file. This field is not intended to be read by the client, but by the ContentResolver. The client will call `ContentResolver.openInputStream()` on the
user-facing field holding the URI for the item. The ContentResolver will request the "_data" field for that record, and because it has higher permissions than a client, it should be able to access that file directly and return a read wrapper for the file to the client.

For an example of a private content provider implementation, see the NodePadProvider class in the Notepad sample application that ships with the SDK.

**Declaring the content provider**

To let the Android system know about the content provider you've developed, declare it with a <provider> element in the application's AndroidManifest.xml file. Content providers that are not declared in the manifest are not visible to the Android system.

The name attribute is the fully qualified name of the ContentProvider subclass. The authorities attribute is the authority part of the content: URI that identifies the provider. For example if the ContentProvider subclass is AutoInfoProvider, the <provider> element might look like this:

```xml
<provider name="com.example.autos.AutoInfoProvider"
   authorities="com.example.autos.autoinfoprovider"
   . . . />
</provider>
```

Note that the authorities attribute omits the path part of a content: URI. For example, if AutoInfoProvider controlled subtables for different types of autos or different manufacturers,

```xml
content://com.example.autos.autoinfoprovider/honda
content://com.example.autos.autoinfoprovider/gm/compact
content://com.example.autos.autoinfoprovider/gm/suv
```

those paths would not be declared in the manifest. The authority is what identifies the provider, not the path; your provider can interpret the path part of the URI in any way you choose.

Other <provider> attributes can set permissions to read and write data, provide for an icon and text that can be displayed to users, enable and disable the provider, and so on. Set the multiprocess attribute to "true" if data does not need to be synchronized between multiple running versions of the content provider. This permits an instance of the provider to be created in each client process, eliminating the need to perform IPC.

**Content URI Summary**

Here is a recap of the important parts of a content URI:
A. Standard prefix indicating that the data is controlled by a content provider. It's never modified.

B. The authority part of the URI; it identifies the content provider. For third-party applications, this should be a fully-qualified class name (reduced to lowercase) to ensure uniqueness. The authority is declared in the `<provider>` element's `authorities` attribute:

C. `<provider name=".TransportationProvider"
D. authorities="com.example.transportationprovider"
   ...` >

E. The path that the content provider uses to determine what kind of data is being requested. This can be zero or more segments long. If the content provider exposes only one type of data (only trains, for example), it can be absent. If the provider exposes several types, including subtypes, it can be several segments long — for example, "land/bus", "land/train", "sea/ship", and "sea/submarine" to give four possibilities.

F. The ID of the specific record being requested, if any. This is the `_ID` value of the requested record. If the request is not limited to a single record, this segment and the trailing slash are omitted:

   `content://com.example.transportationprovider/trains`