Content Providers

Sang Shin
Michèle Garoche
www.javapassion.com
“Learn with Passion!”

Disclaimer

- Portions of this presentation are modifications based on work created and shared by the Android Open Source Project
  - http://code.google.com/policies.html
- They are used according to terms described in the Creative Commons 2.5 Attribution License
  - http://creativecommons.org/licenses/by/2.5/

Topics

- What is Content Provider?
- Data model
- URIs
- Querying Content Provider
- Modifying data
- Creating a Content Provider
- Contacts Content Provider

What is a Content Provider?

What is a Content Provider?

- Store and retrieve data and make it accessible to all applications
- Only way to share data across applications
  - There's no common storage area that all Android packages can access
- Two types of content providers
  - Android's built-in content providers
  - Custom content providers you create

Android's Built-in Content Providers

- Android ships with a number of content providers for common data types
  - audio, video, images, personal contact information, and so on
  - *android.provider* package
- You can query these providers for the data they contain

Android's Built-in Content Providers

- Browser
  - Browser bookmarks, browser history
- CallLog
  - Missed calls, call details
- Contacts
  - Contact details
- MediaStore
  - Media files
- Settings
  - Device settings and preferences

How do you make your data public?

- Two options
  - You can create your own content provider (extending *ContentProvider* class) or
  - You can add your data to an existing provider — if there's one that controls the same type of data and you have permission to write to it.

Content Provider Implementation & Usage Model

- All content providers implement a common interface for
  - querying the provider and returning results
  - adding
  - altering
  - deleting
- How a content provider actually stores its data under the cover is up to its designer
  - Internally, it is likely to use SQLite database
- Clients access content providers indirectly through *ContentResolver*

Data Model

Data Model is Table based

- Content providers expose their data as a simple table (like in a database) model
  - Each row is a record and each column is data of a particular type
  - Every record includes a numeric *_ID* field that uniquely identifies the record within the table

Query returns Cursor object

- A query returns a *Cursor* object that can move from record to record (row to row)
  - *move (int offset), moveToFirst(), moveToLast(), moveToNext(), moveToPosition(int position). etc.*
- The *Cursor* object has specialized methods for reading each type of data. So, to read a field, you must know what type of data the field contains
  - *getDouble(int columnindex), getFloat(int columnindex), getInt(int columnindex), etc.*

Content Provider Data set (Table) URIs

Content Provider exposes public URI

- Each content provider exposes a public URI (wrapped as a *Uri* object) that uniquely identifies its data set (table).
  - 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.

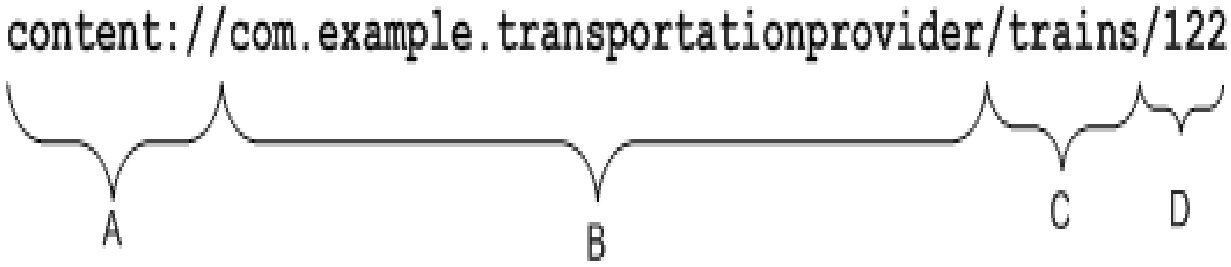
Built-in URI Definitions

- 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:
  - *android.provider.Contacts.Phones.CONTENT_URI*
  - *android.provider.Contacts.Photos.CONTENT_URI*

URI Usage

- 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.

URI Structure



- 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.
- C: The path that the content provider uses to determine what data set (which table) is being requested.
- D: The ID of the specific record being requested

Querying a Content Provider

Querying a Content Provider

- You need three pieces of information to query a content provider
  - The content URI that identifies the content provider and a data set (table)
  - The names of the data fields you want to receive
  - The data types for those fields (used after the query)
- If you're querying a particular record, you also need the ID for that record to the URI

Example: Querying a Table

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

// Form an array specifying which columns to retrieve
private static final String[] PROJECTION = new String[] {
                                    BooksProvider._ID,
                                    BooksProvider.TITLE };

// Perform a query.
// BooksProvider.CONTENT_URI is defined as
// “content://com.javapassion.BooksProvider/books”
Cursor mCursor = managedQuery(
            BooksProvider.CONTENT_URI,  // Which content provider
                                        // and which table
            PROJECTION,         // Which columns to retrieve
            null,               // Which rows to return (all rows)
            null,               // Selection arguments (none)
             "_ID ASC");        // Put the results in ascending order by _ID
```

Querying a Single Record

- To restrict a query to just one record, you can append the _ID value for that record to the URI
  - *content://. . . ./23*
- Use helper methods to append the _ID value
  - *ContentUris.withAppendedId()* or
  - *Uri.withAppendedPath()*

Example: Querying a Single Record

```
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 myBookUri =
ContentUris.withAppendedId(BooksProvider.CONTENT_URI, 23);

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

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

What a Query Returns

- A query returns a set of zero or more records
- 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 *Cursor* object only to read the data.
  - To add, modify, or delete data, you must use a *ContentResolver* object.

Reading Retrieved Data via Cursor

- *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
- To read field values, you must know the data types of the fields
  - Because the Cursor object has a separate method for reading each type of data — such as *getString(int column_index)*, *getInt(int column_index)*, and *getFloat(int column_index)*
  - However, for most types, if you call the method for reading strings, the Cursor object will give you the String representation of the data

Reading retrieved data using Cursor

```
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());

    }
}
```

Modifying Data

Modifying Data

- Tasks for modifying data
  - Adding new records
  - Adding new values to existing records
  - Batch updating (Bulk updating) existing records
  - Deleting records
- All data modification is accomplished using methods of *ContentResolver* class
- Some content providers require a more restrictive permission for writing data than they do for reading it.

Steps for Adding new records

- Step #1: Set up a map of key-value pairs in a *ContentValues* object
  - 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.
- Step #2: Call *ContentResolver.insert()* and pass it the URI of the provider and the *ContentValues* object.
  - This method returns the full URI of the newly added record - useful for further manipulation

Example: Adding a new record

```
// Set up a map of key-value pairs in a ContentValues object
ContentValues values = new ContentValues();
values.put(BooksProvider._ID, "1");
values.put(BooksProvider.TITLE,
        "Android Programming");
values.put(BooksProvider.ISBN, "1234567890");

// Call ContentResolver.insert() and pass it the URI of the
// provider and the ContentValues object.
uriNewlyAdded = getContentResolver().insert(
        BooksProvider.CONTENT_URI, // Specifies which content
                                   // provider and which table
        values);
```

Batch (Bulk) updating records

- Used to batch update a group of records (for example, to change "NY" to "New York" for “state” column for all or selected set of rows)
- Call the *ContentResolver.update()* method with the columns and values to change.

Example: Batch Update

```
ContentValues editedValues = new ContentValues();

// Change the title to “Asian Cooking”
editedValues.put(BooksProvider.TITLE, "Asian Cooking");

getContentResolver().update(
                        BooksProvider.CONTENT_URI, // URI
                        editedValues, // ContentValues
                        null,              // Where clause
                        null);             // Selection arguments
```

Deleting a record or records

- 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

Creating a Content Provider

Steps for Creating Content Provider

- Declare the content provider in the manifest file for your application (AndroidManifest.xml).
- Set up a storage 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.
- Extend the *ContentProvider* class to provide access to the data.
  - Implement six abstract methods - query(), insert(), update(), delete(), getType(), onCreate()

Declare Content Provider

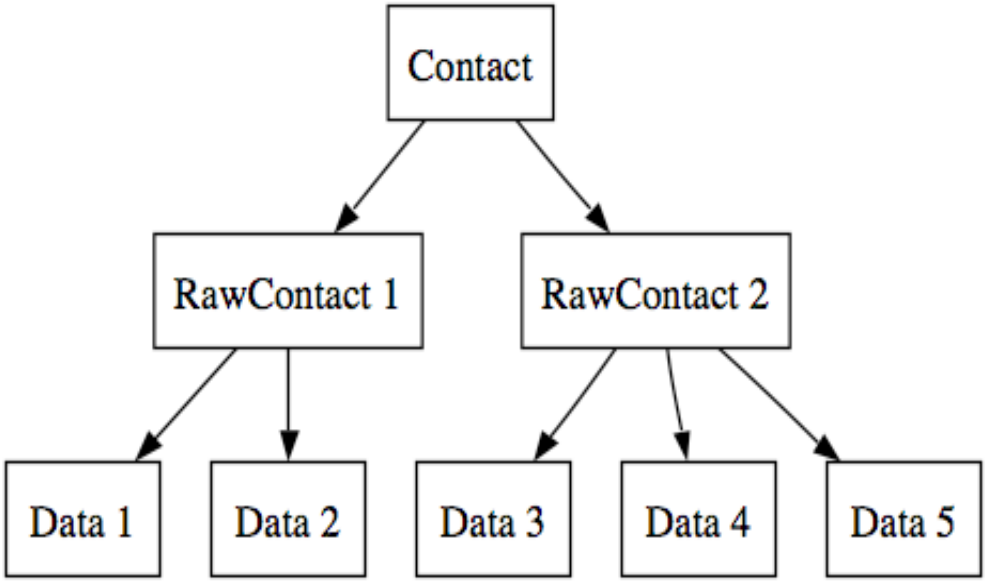
- To let the Android system know about the content provider you've developed
- Declare it with a <provider> element in the application's AndroidManifest.xml file

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

Contacts Content Provider

Data Models of Contacts

- Data is laid out in three primary tables: Contacts, RawContacts, and Data
- Designed to easily store and manage information for a specific contact from multiple contacts sources.



Tables of Contacts

- Data tables
  - A row can store any kind of personal data, such as a phone number or email addresses.
  - The set of data kinds that can be stored in this table is open-ended. There is a predefined set of common kinds.
- RawContact table
  - A row represents a set of data describing a person and associated with a single account (for example, one of the user's Gmail accounts).
- Contact table
  - A row represents an aggregate of one or more RawContacts presumably describing the same person.

Contacts Content Provider: Data Tables

Data Tables

- Data is a generic table that stores all of the data points associated with a raw contact.
- Each row stores data of a specific kind — for example name, photo, email addresses, phone numbers, and group memberships.
- Each row is tagged with a MIME type to identify what type of data it can contain, across the entire column.

Data Tables - Columns

- Columns are generic and the type of data they contain is determined by the kind of data stored in each row.
  - For example, if a row's data kind is *Phone.CONTENT_ITEM_TYPE Mime type*, then the first column stores the phone number, but if the data kind is *Email.CONTENT_ITEM_TYPE Mime type*, then the column stores the email address.

Examples of Common MIME Types

- Phone.CONTENT_ITEM_TYPE
- Email.CONTENT_ITEM_TYPE
- Photo.CONTENT_ITEM_TYPE
- StructuredName.CONTENT_ITEM_TYPE
- Organization.CONTENT_ITEM_TYPE
- Im.CONTENT_ITEM_TYPE
- Nickname.CONTENT_ITEM_TYPE
- Note.CONTENT_ITEM_TYPE
- StructuredPostal.CONTENT_ITEM_TYPE
- GroupMembership.CONTENT_ITEM_TYPE
- Website.CONTENT_ITEM_TYPEEvent.CONTENT_ITEM_TYPE
- Relation.CONTENT_ITEM_TYPE

Data Table Insert Schemes

- Scheme #1: Incremental insertion
  - An individual data row is inserted using the traditional *insert(Uri, ContentValues)* method.
- Scheme #2: Batch insertion
  - Multiple rows can be inserted as a batch using *ContentProviderOperations*
  - This is a preferred approach

Data Table Insert Scheme #1

- Incremental Insertion

```
// Example of traditional insert(Uri, ContentValues) method
ContentValues values = new ContentValues();
values.put(Data.RAW_CONTACT_ID, rawContactId);
values.put(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE);
values.put(Phone.NUMBER, "1-800-GOOG-411”)
values.put(Phone.TYPE, Phone.TYPE_CUSTOM);
values.put(Phone.LABEL, "free directory assistance");

Uri dataUri = getContentResolver().
                    insert(Data.CONTENT_URI, values);
```

Data Table Insert Scheme #2

- Batch insersion

```
ArrayList<ContentProviderOperation> ops = Lists.newArrayList();
ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
        .withValue(Data.RAW_CONTACT_ID, rawContactId)
        .withValue(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE)
        .withValue(Phone.NUMBER, "1-800-GOOG-411")
        .withValue(Phone.TYPE, Phone.TYPE_CUSTOM)
        .withValue(Phone.LABEL, "free directory assistance")
        .build());
ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
        .withValue(Data.RAW_CONTACT_ID, rawContactId2)
        .withValue(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE)
        .withValue(Phone.NUMBER, "1-234-455-555")
        .withValue(Phone.TYPE, Phone.TYPE_CUSTOM)
        .withValue(Phone.LABEL, "whatever")
        .build());
getContentResolver().applyBatch(ContactsContract.AUTHORITY,
                                                    ops);
```

Data Table Update

- Just as with insert, update can be done incrementally or as a batch, the batch mode being the preferred method:

```
// Example of batch update
ArrayList<ContentProviderOperation> ops = Lists.newArrayList();
ops.add(ContentProviderOperation.newUpdate(
        Data.CONTENT_URI).withSelection(Data._ID + "=?",
                new String[]{String.valueOf(dataId)})
        .withValue(Email.DATA, "somebody@android.com")
        .build());

getContentResolver().applyBatch(ContactsContract.AUTHORITY,
                                ops);
```

Data Table Delete

- Just as with insert and update, delete can be done incrementally or as a batch, the batch mode being the preferred method:

```
// Example of batch delete
ArrayList<ContentProviderOperation> ops = Lists.newArrayList();
ops.add(ContentProviderOperation.newDelete(
        Data.CONTENT_URI)
        .withSelection(Data._ID + "=?",
        new String[]{String.valueOf(dataId)})
        .build());

getContentResolver().applyBatch(ContactsContract.AUTHORITY,
                                                ops);
```

Data Table Query #1

- Finding all Data of a given type for a given contact id

```
Cursor c = getContentResolver().query(
        Data.CONTENT_URI,
        new String[] {Data._ID, Phone.NUMBER,
                      Phone.TYPE, Phone.LABEL},
        Data.CONTACT_ID + "=?" + " AND "
              + Data.MIMETYPE + "='"
              + Phone.CONTENT_ITEM_TYPE + "'",
        new String[] {String.valueOf(contactId)}, null);
```

Data Table Query #2

- Finding all Data of a given type for a given raw contact

```
Cursor c = getContentResolver().query(
        Data.CONTENT_URI,
        new String[] {Data._ID, Phone.NUMBER,
                      Phone.TYPE, Phone.LABEL},
        Data.RAW_CONTACT_ID + "=?" + " AND "
           + Data.MIMETYPE + "='" +
              Phone.CONTENT_ITEM_TYPE + "'",
        new String[] {String.valueOf(rawContactId)}, null);
```

Contacts Content Provider: RawContact Tables

RawContact Tables

- A row in the RawContacts table represents the set of Data and other information describing a person and associated with a single contacts source.
  - For example, a row might define the data associated with a person's Google or Exchange account or Facebook friend - in this case, one row contains info. from Google and another row contains info. on Exchange, another row contains info. from Facebook
- Sync adapters and contact management apps are the primary consumers

RawContact Table - Aggregation

- As soon as a raw contact is inserted or whenever its constituent data changes, the provider will check if the raw contact matches other existing raw contacts and if so will aggregate it with those.
  - The aggregation is reflected in the ContactsContract.RawContacts table by the change of the CONTACT_ID field, which is the reference to the aggregate contact.
- Changes to the structured name, organization, phone number, email address, or nickname trigger a re-aggregation.

RawContacts Table Insert Scheme #1

- Raw contacts can be inserted incrementally or in a batch.
- The incremental method is more traditional but less efficient. It should be used only if no ContactsContract.RawContacts.Data values are available at the time the raw contact is created:

```
ContentValues values = new ContentValues();
values.put(RawContacts.ACCOUNT_TYPE, accountType);
values.put(RawContacts.ACCOUNT_NAME, accountName);

Uri rawContactUri =
getContentResolver().insert(RawContacts.CONTENT_URI, values);
long rawContactId = ContentUris.parseId(rawContactUri);
```

RawContacts Tables

- Contacts.ACCOUNT_TYPE
  - The type of account to which this row belongs, which when paired with ACCOUNT_NAME identifies a specific account.
  - To ensure uniqueness, new account types should be chosen according to the Java package naming convention. Thus a Google account is of type "com.google".
- Contacts.ACCOUNT_NAME
  - The name of the account instance to which this row belongs, which when paired with ACCOUNT_TYPE identifies a specific account.
  - For example, this will be the Gmail address if it is a Google account.

RawContacts Table Insert Scheme #2

- The batch method is by far preferred. It inserts the raw contact and its constituent data rows in a single database transaction and causes at most one aggregation pass

```
ArrayList<ContentProviderOperation> ops = Lists.newArrayList();
...
int rawContactInsertIndex = ops.size();
ops.add(ContentProviderOperation.newInsert(RawContacts.CONTENT_URI)
        .withValue(RawContacts.ACCOUNT_TYPE, accountType)
        .withValue(RawContacts.ACCOUNT_NAME, accountName)
        .build());
// Note the use of withValueBackReference(String, int)  to refer to the
// as-yet-unknown index value of the raw contact inserted in the first operation.
ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
        .withValueBackReference(Data.RAW_CONTACT_ID, rawContactInsertIndex)
        .withValue(Data.MIMETYPE, StructuredName.CONTENT_ITEM_TYPE)
        .withValue(StructuredName.DISPLAY_NAME, "Mike Sullivan")
        .build());

getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
```

RawContacts Table Update

- Raw contacts can be updated incrementally or in a batch.
- Batch mode should be used whenever possible.
- The procedures and considerations are analogous to those for inserts.

RawContacts Table Delete

- When a raw contact is deleted, all of its Data rows as well as StatusUpdates, AggregationExceptions, PhoneLookup rows are deleted automatically.

RawContacts Table Query #1

- Find all raw contacts in a Contact:

```
// Find all raw contacts in a Contact:
Cursor c =
getContentResolver().query(RawContacts.CONTENT_URI,
        new String[]{RawContacts._ID},
        RawContacts.CONTACT_ID + "=?",
        new String[]{String.valueOf(contactId)}, null);
```

RawContacts Table Query #2

- To find raw contacts within a specific account, you can either put the account name and type in the selection or pass them as query parameters. The latter approach is preferable, especially when you can reuse the URI:

```
Uri rawContactUri = RawContacts.URI.buildUpon()
   .appendQueryParameter(RawContacts.ACCOUNT_NAME, accountName)
   .appendQueryParameter(RawContacts.ACCOUNT_TYPE, accountType)
   .build();
Cursor c1 = getContentResolver().query(rawContactUri,
         RawContacts.STARRED + "<>0", null, null, null);
...
Cursor c2 = getContentResolver().query(rawContactUri,
         RawContacts.DELETED + "<>0", null, null, null);
```

RawContacts Table Query #3

- The best way to read a raw contact along with all the data associated with it is by using the *ContactsContract.RawContacts.Entity* directory.
- If the raw contact has data rows, the Entity cursor will contain a row for each data row.
- If the raw contact has no data rows, the cursor will still contain one row with the raw contact-level information.

RawContacts Table Query #3

```
Uri rawContactUri =
 ContentUris.withAppendedId(RawContacts.CONTENT_URI, rawContactId);
Uri entityUri = Uri.withAppendedPath(rawContactUri,
                                     Entity.CONTENT_DIRECTORY);
Cursor c = getContentResolver().query(entityUri,
        new String[]{RawContacts.SOURCE_ID, Entity.DATA_ID,
                     Entity.MIMETYPE, Entity.DATA1},
                     null, null, null);
 try {
    while (c.moveToNext()) {
       String sourceId = c.getString(0);
       if (!c.isNull(1)) {
          String mimeType = c.getString(2);
          String data = c.getString(3);
          ...
       }
    }
 } finally {
    c.close();
}
```

Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"