Content Provider Basics

A content provider manages access to a central repository of data. A provider is part of an Android application, which often provides its own UI for working with the data. However, content providers are primarily intended to be used by other applications, which access the provider using a provider client object. Together, providers and provider clients offer a consistent, standard interface to data that also handles inter-process communication and secure data access.

This topic describes the basics of the following:

- How content providers work.
- The API you use retrieve data from a content provider.
- The API you use to insert, update, or delete data in a content provider.
- Other API features that facilitate working with providers.

IN THIS DOCUMENT

KEY CLASSES

`ContentProvider`
`ContentResolver`
`Cursor`
`Uri`

RELATED SAMPLES

Cursor (People)
Cursor (Phones)

SEE ALSO

Creating a Content Provider
Calendar Provider

Overview

A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance.

For example, one of the built-in providers in the Android platform is the user dictionary, which stores the spellings of non-standard words that the user wants to keep. Table 1 illustrates what the data might look like in this provider's table:

Table 1: Sample user dictionary table.

word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

In table 1, each row represents an instance of a word that might not be found in a standard dictionary. Each column represents some data for that word, such as the locale in which it was first encountered. The column headers are column names that are stored in the provider. To refer to a row's locale, you refer to its `locale` column. For this provider, the `_ID` column serves as a "primary key" column that the provider automatically maintains.

> **Note:** A provider isn't required to have a primary key, and it isn't required to use `_ID` as the column name of a primary key if one is present. However, if you want to bind data from a provider to a `ListView` (/reference/android/widget/ListView.html), one of the column names has to be `_ID`. This requirement is explained in more detail in the section Displaying query results (#DisplayResults).

Accessing a provider

An application accesses the data from a content provider with a `ContentResolver` (/reference/android/content/ContentResolver.html) client object. This object has methods that call identically-named methods in the provider object, an instance of one of the concrete subclasses of

ContentProvider (/reference/android/content/ContentProvider.html). The ContentResolver (/reference/android/content/ContentResolver.html) methods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.

The ContentResolver (/reference/android/content/ContentResolver.html) object in the client application's process and the ContentProvider (/reference/android/content/ContentProvider.html) object in the application that owns the provider automatically handle inter-process communication. ContentProvider (/reference/android/content/ContentProvider.html) also acts as an abstraction layer between its repository of data and the external appearance of data as tables.

> **Note:** To access a provider, your application usually has to request specific permissions in its manifest file. This is described in more detail in the section Content Provider Permissions (#Permissions)

For example, to get a list of the words and their locales from the User Dictionary Provider, you call ContentResolver.query() (/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)). The query() (/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)) method calls the ContentProvider.query() (/reference/android/content/ContentProvider.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)) method defined by the User Dictionary Provider. The following lines of code show a ContentResolver.query() (/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)) call:

```
// Queries the user dictionary and returns results
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    mProjection,                        // The columns to return for each row
    mSelectionClause                    // Selection criteria
    mSelectionArgs,                     // Selection criteria
    mSortOrder);                        // The sort order for the returned rows
```

Table 2 shows how the arguments to query(Uri,projection,selection,selectionArgs,sortOrder) (/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)) match an SQL SELECT statement:

Table 2: Query() compared to SQL query.

query() argument	SELECT keyword/parameter	Notes
`Uri`	`FROM` *`table_name`*	`Uri` maps to the table in the provider named *table_name*.
`projection`	*`col,col,col,...`*	`projection` is an array of columns that should be included for each row retrieved.
`selection`	`WHERE` *`col`* `=` *`value`*	`selection` specifies the criteria for selecting rows.
`selectionArgs`	(No exact equivalent. Selection arguments replace `?` placeholders in the selection clause.)	
`sortOrder`	`ORDER BY` *`col,col,...`*	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

Content URIs

A **content URI** is a URI that identifies data in a provider. Content URIs include the symbolic name of the entire provider (its **authority**) and a name that points to a table (a **path**). When you call a client method to access a table in a provider, the content URI for the table is one of the arguments.

In the preceding lines of code, the constant `CONTENT_URI` (/reference/android/provider/UserDictionary.Words.html#CONTENT_URI) contains the content URI of the user dictionary's "words" table. The `ContentResolver` (/reference/android/content/ContentResolver.html) object parses out the URI's authority, and uses it to "resolve" the provider by comparing the authority to a system table of known providers. The `ContentResolver` (/reference/android/content/ContentResolver.html) can then dispatch the query arguments to the correct provider.

The `ContentProvider` (/reference/android/content/ContentProvider.html) uses the path part of the content URI to choose the table to access. A provider usually has a **path** for each table it exposes.

In the previous lines of code, the full URI for the "words" table is:

```
content://user_dictionary/words
```

where the `user_dictionary` string is the provider's authority, and `words` string is the table's path. The string `content://` (the **scheme**) is always present, and identifies this as a content URI.

Many providers allow you to access a single row in a table by appending an ID value to the end of the URI. For example, to retrieve a row whose `_ID` is `4` from user dictionary, you can use this content URI:

```
Uri singleUri = ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
```

You often use id values when you've retrieved a set of rows and then want to update or delete one of them.

> **Note:** The `Uri` (/reference/android/net/Uri.html) and `Uri.Builder` (/reference/android/net/Uri.Builder.html) classes contain convenience methods for constructing well-formed Uri objects from strings. The `ContentUris` (/reference/android/content/ContentUris.html) contains convenience methods for appending id values to a URI. The previous snippet uses `withAppendedId()` (/reference/android/content/ContentUris.html#withAppendedId(android.net.Uri, long)) to append an id to the UserDictionary content URI.

Retrieving Data from the Provider

This section describes how to retrieve data from a provider, using the User Dictionary Provider as an example.

> For the sake of clarity, the code snippets in this section call `ContentResolver.query()` (/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)) on the "UI thread"". In actual code, however, you should do queries asynchronously on a separate thread. One way to do this is to use the `CursorLoader` (/reference/android/content/CursorLoader.html) class, which is described in more detail in the Loaders (/guide/components/loaders.html) guide. Also, the lines of code are snippets only; they don't show a complete application.

To retrieve data from a provider, follow these basic steps:

1. Request the read access permission for the provider.
2. Define the code that sends a query to the provider.

Requesting read access permission

To retrieve data from a provider, your application needs "read access permission" for the provider. You can't request this permission at run-time; instead, you have to specify that you need this permission in your manifest, using the `<uses-permission>` (/guide/topics/manifest/uses-permission-element.html) element and the exact permission name defined by the provider. When you specify this element in your manifest, you are in effect "requesting" this permission for your application. When users install your application, they implicitly grant this request.

To find the exact name of the read access permission for the provider you're using, as well as the names for other access permissions used by the provider, look in the provider's documentation.

The role of permissions in accessing providers is described in more detail in the section Content Provider Permissions (#Permissions).

The User Dictionary Provider defines the permission `android.permission.READ_USER_DICTIONARY` in its manifest file, so an application that wants to read from the provider must request this permission.

Constructing the query

The next step in retrieving data a provider is to construct a query. This first snippet defines some variables for accessing the User Dictionary Provider:

```
// A "projection" defines the columns that will be returned for each row
String[] mProjection =
{
    UserDictionary.Words._ID,    // Contract class constant for the _ID column name
    UserDictionary.Words.WORD,   // Contract class constant for the word column name
    UserDictionary.Words.LOCALE  // Contract class constant for the locale column name
};

// Defines a string to contain the selection clause
String mSelectionClause = null;

// Initializes an array to contain selection arguments
String[] mSelectionArgs = {""};
```

The next snippet shows how to use `ContentResolver.query()` `(/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String))`, using the User Dictionary Provider as an example. A provider client query is similar to an SQL query, and it contains a set of columns to return, a set of selection criteria, and a sort order.

The set of columns that the query should return is called a **projection** (the variable `mProjection`).

The expression that specifies the rows to retrieve is split into a selection clause and selection arguments. The selection clause is a combination of logical and Boolean expressions, column names, and values (the variable `mSelectionClause`). If you specify the replaceable parameter `?` instead of a value, the query method retrieves the value from the selection arguments array (the variable `mSelectionArgs`).

In the next snippet, if the user doesn't enter a word, the selection clause is set to `null`, and the query returns all the words in the provider. If the user enters a word, the selection clause is set to `UserDictionary.Words.WORD + " = ?"` and the first element of selection arguments array is set to the word the user enters.

```
/*
 * This defines a one-element String array to contain the selection argument.
 */
String[] mSelectionArgs = {""};

// Gets a word from the UI
mSearchString = mSearchWord.getText().toString();

// Remember to insert code here to check for invalid or malicious input.

// If the word is the empty string, gets everything
if (TextUtils.isEmpty(mSearchString)) {
    // Setting the selection clause to null will return all words
    mSelectionClause = null;
    mSelectionArgs[0] = "";

} else {
    // Constructs a selection clause that matches the word that the user entered.
    mSelectionClause = UserDictionary.Words.WORD + " = ?";

    // Moves the user's input string to the selection arguments.
    mSelectionArgs[0] = mSearchString;

}

// Does a query against the table and returns a Cursor object
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,  // The content URI of the words table
    mProjection,                       // The columns to return for each row
    mSelectionClause                   // Either null, or the word the user entered
    mSelectionArgs,                    // Either empty, or the string the user enterec
    mSortOrder);                       // The sort order for the returned rows

// Some providers return null if an error occurs, others throw an exception
if (null == mCursor) {
    /*
     * Insert code here to handle the error. Be sure not to use the cursor! You may wa
     * call android.util.Log.e() to log this error.
     *
     */
// If the Cursor is empty, the provider found no matches
} else if (mCursor.getCount() < 1) {

    /*
     * Insert code here to notify the user that the search was unsuccessful. This isn
     * an error. You may want to offer the user the option to insert a new row, or re-
     * search term.
     */

} else {
    // Insert code here to do something with the results

}
```

This query is analogous to the SQL statement:

```
SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY word ASC;
```

In this SQL statement, the actual column names are used instead of contract class constants.

Protecting against malicious input

If the data managed by the content provider is in an SQL database, including external untrusted data into raw SQL statements can lead to SQL injection.

Consider this selection clause:

```
// Constructs a selection clause by concatenating the user's input to the column name
String mSelectionClause =  "var = " + mUserInput;
```

If you do this, you're allowing the user to concatenate malicious SQL onto your SQL statement. For example, the user could enter "nothing; DROP TABLE *;" for `mUserInput`, which would result in the selection clause `var = nothing; DROP TABLE *;`. Since the selection clause is treated as an SQL statement, this might cause the provider to erase all of the tables in the underlying SQLite database (unless the provider is set up to catch SQL injection (http://en.wikipedia.org/wiki/SQL_injection) attempts).

To avoid this problem, use a selection clause that uses `?` as a replaceable parameter and a separate array of selection arguments. When you do this, the user input is bound directly to the query rather than being interpreted as part of an SQL statement. Because it's not treated as SQL, the user input can't inject malicious SQL. Instead of using concatenation to include the user input, use this selection clause:

```
// Constructs a selection clause with a replaceable parameter
String mSelectionClause =  "var = ?";
```

Set up the array of selection arguments like this:

```
// Defines an array to contain the selection arguments
String[] selectionArgs = {""};
```

Put a value in the selection arguments array like this:

```
// Sets the selection argument to the user's input
selectionArgs[0] = mUserInput;
```

A selection clause that uses `?` as a replaceable parameter and an array of selection arguments array are preferred way to specify a selection, even if the provider isn't based on an SQL database.

Displaying query results

The `ContentResolver.query()` `(/reference/android/content/ContentResolver.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String))` client method always returns a `Cursor` `(/reference/android/database/Cursor.html)` containing the columns specified by the query's projection for the rows that match the query's selection criteria. A `Cursor (/reference/android/database/Cursor.html)` object provides random read access to the rows and columns it contains. Using `Cursor` `(/reference/android/database/Cursor.html)` methods, you can iterate over the rows in the results, determine the data type of each column, get the data out of a column, and examine other properties of the results. Some `Cursor (/reference/android/database/Cursor.html)` implementations automatically update the object when the provider's data changes, or trigger methods in an observer object when the `Cursor` `(/reference/android/database/Cursor.html)` changes, or both.

> **Note:** A provider may restrict access to columns based on the nature of the object making the query. For example, the Contacts Provider restricts access for some columns to sync adapters, so it won't return them to an activity or service.

If no rows match the selection criteria, the provider returns a `Cursor` `(/reference/android/database/Cursor.html)` object for which `Cursor.getCount()` `(/reference/android/database/Cursor.html#getCount())` is 0 (an empty cursor).

If an internal error occurs, the results of the query depend on the particular provider. It may choose to return `null`, or it may throw an `Exception (/reference/java/lang/Exception.html)`.

Since a `Cursor (/reference/android/database/Cursor.html)` is a "list" of rows, a good way to display the contents of a `Cursor (/reference/android/database/Cursor.html)` is to link it to a `ListView` `(/reference/android/widget/ListView.html)` via a `SimpleCursorAdapter` `(/reference/android/widget/SimpleCursorAdapter.html)`.

The following snippet continues the code from the previous snippet. It creates a `SimpleCursorAdapter` `(/reference/android/widget/SimpleCursorAdapter.html)` object containing the `Cursor`

(/reference/android/database/Cursor.html) retrieved by the query, and sets this object to be the adapter for a ListView (/reference/android/widget/ListView.html):

```
// Defines a list of columns to retrieve from the Cursor and load into an output row
String[] mWordListColumns =
{
    UserDictionary.Words.WORD,   // Contract class constant containing the word column
    UserDictionary.Words.LOCALE  // Contract class constant containing the locale colu
};

// Defines a list of View IDs that will receive the Cursor columns for each row
int[] mWordListItems = { R.id.dictWord, R.id.locale};

// Creates a new SimpleCursorAdapter
mCursorAdapter = new SimpleCursorAdapter(
    getApplicationContext(),               // The application's Context object
    R.layout.wordlistrow,                  // A layout in XML for one row in the ListV
    mCursor,                               // The result from the query
    mWordListColumns,                      // A string array of column names in the cu
    mWordListItems,                        // An integer array of view IDs in the row
    0);                                    // Flags (usually none are needed)

// Sets the adapter for the ListView
mWordList.setAdapter(mCursorAdapter);
```

Note: To back a ListView (/reference/android/widget/ListView.html) with a Cursor (/reference/android/database/Cursor.html), the cursor must contain a column named _ID. Because of this, the query shown previously retrieves the _ID column for the "words" table, even though the ListView (/reference/android/widget/ListView.html) doesn't display it. This restriction also explains why most providers have a _ID column for each of their tables.

Getting data from query results

Rather than simply displaying query results, you can use them for other tasks. For example, you can retrieve spellings from the user dictionary and then look them up in other providers. To do this, you iterate over the rows in the Cursor (/reference/android/database/Cursor.html):

```

// Determine the column index of the column named "word"
int index = mCursor.getColumnIndex(UserDictionary.Words.WORD);

/*
 * Only executes if the cursor is valid. The User Dictionary Provider returns null if
 * an internal error occurs. Other providers may throw an Exception instead of returni
 */

if (mCursor != null) {
    /*
     * Moves to the next row in the cursor. Before the first movement in the cursor, t
     * "row pointer" is -1, and if you try to retrieve data at that position you will
     * exception.
     */
    while (mCursor.moveToNext()) {

        // Gets the value from the column.
        newWord = mCursor.getString(index);

        // Insert code here to process the retrieved word.
```

```
            ...

            // end of while loop
        }
    } else {

        // Insert code here to report an error if the cursor is null or the provider threw
    }
```

Cursor (/reference/android/database/Cursor.html) implementations contain several "get" methods for retrieving different types of data from the object. For example, the previous snippet uses getString() (/reference/android/database/Cursor.html#getString(int)). They also have a getType() (/reference/android/database/Cursor.html#getType(int)) method that returns a value indicating the data type of the column.

Content Provider Permissions

A provider's application can specify permissions that other applications must have in order to access the provider's data. These permissions ensure that the user knows what data an application will try to access. Based on the provider's requirements, other applications request the permissions they need in order to access the provider. End users see the requested permissions when they install the application.

If a provider's application doesn't specify any permissions, then other applications have no access to the provider's data. However, components in the provider's application always have full read and write access, regardless of the specified permissions.

As noted previously, the User Dictionary Provider requires the `android.permission.READ_USER_DICTIONARY` permission to retrieve data from it. The provider has the separate `android.permission.WRITE_USER_DICTIONARY` permission for inserting, updating, or deleting data.

To get the permissions needed to access a provider, an application requests them with a <uses-permission> (/guide/topics/manifest/uses-permission-element.html) element in its manifest file. When the Android Package Manager installs the application, a user must approve all of the permissions the application requests. If the user approves all of them, Package Manager continues the installation; if the user doesn't approve them, Package Manager aborts the installation.

The following <uses-permission> (/guide/topics/manifest/uses-permission-element.html) element requests read access to the User Dictionary Provider:

```
    <uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

The impact of permissions on provider access is explained in more detail in the Security and Permissions (/guide/topics/security/security.html) guide.

Inserting, Updating, and Deleting Data

In the same way that you retrieve data from a provider, you also use the interaction between a provider client and the provider's ContentProvider (/reference/android/content/ContentProvider.html) to modify data. You call a method of ContentResolver (/reference/android/content/ContentResolver.html) with arguments that are passed to the corresponding method of ContentProvider (/reference/android/content/ContentProvider.html). The provider and provider client automatically handle security and inter-process communication.

Inserting data

To insert data into a provider, you call the ContentResolver.insert() (/reference/android/content/ContentResolver.html#insert(android.net.Uri, android.content.ContentValues)) method. This method inserts a new row into the provider and returns a content URI for that row. This snippet shows how to insert a new word into the User Dictionary Provider:

```
// Defines a new Uri object that receives the result of the insertion
Uri mNewUri;

...

// Defines an object to contain the new values to insert
ContentValues mNewValues = new ContentValues();

/*
 * Sets the values of each column and inserts the word. The arguments to the "put"
 * method are "column name" and "value"
 */
mNewValues.put(UserDictionary.Words.APP_ID, "example.user");
mNewValues.put(UserDictionary.Words.LOCALE, "en_US");
mNewValues.put(UserDictionary.Words.WORD, "insert");
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");

mNewUri = getContentResolver().insert(
    UserDictionary.Word.CONTENT_URI,   // the user dictionary content URI
    mNewValues                          // the values to insert
);
```

The data for the new row goes into a single ContentValues (/reference/android/content/ContentValues.html) object, which is similar in form to a one-row cursor. The columns in this object don't need to have the same data type, and if you don't want to specify a value at all, you can set a column to null using ContentValues.putNull() (/reference/android/content/ContentValues.html#putNull(java.lang.String)).

The snippet doesn't add the _ID column, because this column is maintained automatically. The provider assigns a unique value of _ID to every row that is added. Providers usually use this value as the table's primary key.

The content URI returned in newUri identifies the newly-added row, with the following format:

```
content://user_dictionary/words/<id_value>
```

The <id_value> is the contents of _ID for the new row. Most providers can detect this form of content URI automatically and then perform the requested operation on that particular row.

To get the value of _ID from the returned Uri (/reference/android/net/Uri.html), call ContentUris.parseId() (/reference/android/content/ContentUris.html#parseId(android.net.Uri)).

Updating data

To update a row, you use a ContentValues (/reference/android/content/ContentValues.html) object with the updated values just as you do with an insertion, and selection criteria just as you do with a query. The client method you use is ContentResolver.update() (/reference/android/content/ContentResolver.html#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[])). You only need to add values to the ContentValues (/reference/android/content/ContentValues.html) object for columns you're updating. If you want to clear the contents of a column, set the value to null.

The following snippet changes all the rows whose locale has the language "en" to a have a locale of null. The return value is the number of rows that were updated:

```
// Defines an object to contain the updated values
```

```
ContentValues mUpdateValues = new ContentValues();

// Defines selection criteria for the rows you want to update
String mSelectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";
String[] mSelectionArgs = {"en_%"};

// Defines a variable to contain the number of updated rows
int mRowsUpdated = 0;

...

/*
 * Sets the updated value and updates the selected words.
 */
mUpdateValues.putNull(UserDictionary.Words.LOCALE);

mRowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,   // the user dictionary content URI
    mUpdateValues                       // the columns to update
    mSelectionClause                    // the column to select on
    mSelectionArgs                      // the value to compare to
);
```

You should also sanitize user input when you call `ContentResolver.update()` `(/reference/android/content/ContentResolver.html#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[]))`. To learn more about this, read the section Protecting against malicious input (#Injection).

Deleting data

Deleting rows is similar to retrieving row data: you specify selection criteria for the rows you want to delete and the client method returns the number of deleted rows. The following snippet deletes rows whose appid matches "user". The method returns the number of deleted rows.

```

// Defines selection criteria for the rows you want to delete
String mSelectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] mSelectionArgs = {"user"};

// Defines a variable to contain the number of rows deleted
int mRowsDeleted = 0;

...

// Deletes the words that match the selection criteria
mRowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,   // the user dictionary content URI
    mSelectionClause                    // the column to select on
    mSelectionArgs                      // the value to compare to
);
```

You should also sanitize user input when you call `ContentResolver.delete()` `(/reference/android/content/ContentResolver.html#delete(android.net.Uri, java.lang.String, java.lang.String[]))`. To learn more about this, read the section Protecting against malicious input (#Injection).

Provider Data Types

Content providers can offer many different data types. The User Dictionary Provider offers only text, but providers can also offer the following formats:

- integer
- long integer (long)
- floating point
- long floating point (double)

Another data type that providers often use is Binary Large OBject (BLOB) implemented as a 64KB byte array. You can see the available data types by looking at the `Cursor` (/reference/android/database/Cursor.html) class "get" methods.

The data type for each column in a provider is usually listed in its documentation. The data types for the User Dictionary Provider are listed in the reference documentation for its contract class `UserDictionary.Words` (/reference/android/provider/UserDictionary.Words.html) (contract classes are described in the section Contract Classes (#ContractClasses)). You can also determine the data type by calling `Cursor.getType()` (/reference/android/database/Cursor.html#getType(int)).

Providers also maintain MIME data type information for each content URI they define. You can use the MIME type information to find out if your application can handle data that the provider offers, or to choose a type of handling based on the MIME type. You usually need the MIME type when you are working with a provider that contains complex data structures or files. For example, the `ContactsContract.Data` (/reference/android/provider/ContactsContract.Data.html) table in the Contacts Provider uses MIME types to label the type of contact data stored in each row. To get the MIME type corresponding to a content URI, call `ContentResolver.getType()` (/reference/android/content/ContentResolver.html#getType(android.net.Uri)).

The section MIME Type Reference (#MIMETypeReference) describes the syntax of both standard and custom MIME types.

Alternative Forms of Provider Access

Three alternative forms of provider access are important in application development:

- Batch access: You can create a batch of access calls with methods in the `ContentProviderOperation` class, and then apply them with `ContentResolver.applyBatch()`.
- Asynchronous queries: You should do queries in a separate thread. One way to do this is to use a `CursorLoader` object. The examples in the Loaders guide demonstrate how to do this.
- Data access via intents: Although you can't send an intent directly to a provider, you can send an intent to the provider's application, which is usually the best-equipped to modify the provider's data.

Batch access and modification via intents are described in the following sections.

Batch access

Batch access to a provider is useful for inserting a large number of rows, or for inserting rows in multiple tables in the same method call, or in general for performing a set of operations across process boundaries as a transaction (an atomic operation).

To access a provider in "batch mode", you create an array of `ContentProviderOperation` (/reference/android/content/ContentProviderOperation.html) objects and then dispatch them to a content provider with `ContentResolver.applyBatch()` (/reference/android/content/ContentResolver.html#applyBatch(java.lang.String, java.util.ArrayList<android.content.ContentProviderOperation>)). You pass the content provider's *authority* to this method, rather than a particular content URI. This allows each `ContentProviderOperation` (/reference/android/content/ContentProviderOperation.html) object in the array to work against a different table. A call to `ContentResolver.applyBatch()` (/reference/android/content/ContentResolver.html#applyBatch(java.lang.String, java.util.ArrayList<android.content.ContentProviderOperation>)) returns an array of results.

The description of the `ContactsContract.RawContacts` (/reference/android/provider/ContactsContract.RawContacts.html) contract class includes a code snippet that demonstrates batch insertion. The Contact Manager (/resources/samples/ContactManager/index.html) sample application contains an example of batch access in its `ContactAdder.java` source file.

Data access via intents

Intents can provide indirect access to a content provider. You allow the user to access data in a provider even if your application doesn't have access permissions, either by getting a result intent back from an application that has permissions, or by activating an application that has permissions and letting the user do work in it.

Displaying data using a helper app

If your application *does* have access permissions, you still may want to use an intent to display data in another application. For example, the Calendar application accepts an `ACTION_VIEW` (/reference/android/content/Intent.html#ACTION_VIEW) intent, which displays a particular date or event. This allows you to display calendar information without having to create your own UI. To learn more about this feature, see the Calendar Provider (/guide/topics/providers/calendar-provider.html) guide.

The application to which you send the intent doesn't have to be the application associated with the provider. For example, you can retrieve a contact from the Contact Provider, then send an `ACTION_VIEW` (/reference/android/content/Intent.html#ACTION_VIEW) intent containing the content URI for the contact's image to an image viewer.

Getting access with temporary permissions

You can access data in a content provider, even if you don't have the proper access permissions, by sending an intent to an application that does have the permissions and receiving back a result intent containing "URI" permissions. These are permissions for a specific content URI that last until the activity that receives them is finished. The application that has permanent permissions grants temporary permissions by setting a flag in the result intent:

- Read permission: `FLAG_GRANT_READ_URI_PERMISSION`
- Write permission: `FLAG_GRANT_WRITE_URI_PERMISSION`

> **Note:** These flags don't give general read or write access to the provider whose authority is contained in the content URI. The access is only for the URI itself.

A provider defines URI permissions for content URIs in its manifest, using the `android:grantUriPermission` `(/guide/topics/manifest/provider-element.html#gprmsn)` attribute of the `<provider>` `(/guide/topics/manifest/provider-element.html)` element, as well as the `<grant-uri-permission>` `(/guide/topics/manifest/grant-uri-permission-element.html)` child element of the `<provider>` `(/guide/topics/manifest/provider-element.html)` element. The URI permissions mechanism is explained in more detail in the Security and Permissions (/guide/topics/security/security.html) guide, in the section "URI Permissions".

For example, you can retrieve data for a contact in the Contacts Provider, even if you don't have the `READ_CONTACTS` `(/reference/android/Manifest.permission.html#READ_CONTACTS)` permission. You might want to do this in an application that sends e-greetings to a contact on his or her birthday. Instead of requesting `READ_CONTACTS` `(/reference/android/Manifest.permission.html#READ_CONTACTS)`, which gives you access to all of the user's contacts and all of their information, you prefer to let the user control which contacts are used by your application. To do this, you use the following process:

1. Your application sends an intent containing the action `ACTION_PICK` and the "contacts" MIME type `CONTENT_ITEM_TYPE`, using the method `startActivityForResult()`.
2. Because this intent matches the intent filter for the People app's "selection" activity, the activity will come to the foreground.
3. In the selection activity, the user selects a contact to update. When this happens, the selection activity calls `setResult(resultcode, intent)` to set up a intent to give back to your application. The intent contains the content URI of the contact the user selected, and the "extras" flags `FLAG_GRANT_READ_URI_PERMISSION`. These flags grant URI permission to your app to read data for the contact pointed to by the content URI. The selection activity then calls `finish()` to return control to your application.
4. Your activity returns to the foreground, and the system calls your activity's `onActivityResult()` method. This method receives the result intent created by the selection activity in the People app.
5. With the content URI from the result intent, you can read the contact's data from the Contacts Provider, even though you didn't request permanent read access permission to the provider in your manifest. You can then get the contact's birthday information or his or her email address and then send the e-greeting.

Using another application

A simple way to allow the user to modify data to which you don't have access permissions is to activate an application that has permissions and let the user do the work there.

For example, the Calendar application accepts an `ACTION_INSERT` (/reference/android/content/Intent.html#ACTION_INSERT) intent, which allows you to activate the application's insert UI. You can pass "extras" data in this intent, which the application uses to pre-populate the UI. Because recurring events have a complex syntax, the preferred way of inserting events into the Calendar Provider is to activate the Calendar app with an `ACTION_INSERT` (/reference/android/content/Intent.html#ACTION_INSERT) and then let the user insert the event there.

Contract Classes

A contract class defines constants that help applications work with the content URIs, column names, intent actions, and other features of a content provider. Contract classes are not included automatically with a provider; the provider's developer has to define them and then make them available to other developers. Many of the providers included with the Android platform have corresponding contract classes in the package `android.provider` (/reference/android/provider/package-summary.html).

For example, the User Dictionary Provider has a contract class `UserDictionary` (/reference/android/provider/UserDictionary.html) containing content URI and column name constants. The content URI for the "words" table is defined in the constant `UserDictionary.Words.CONTENT_URI` (/reference/android/provider/UserDictionary.Words.html#CONTENT_URI). The `UserDictionary.Words` (/reference/android/provider/UserDictionary.Words.html) class also contains column name constants, which are used in the example snippets in this guide. For example, a query projection can be defined as:

```
String[] mProjection =
{
    UserDictionary.Words._ID,
    UserDictionary.Words.WORD,
    UserDictionary.Words.LOCALE
};
```

Another contract class is `ContactsContract` (/reference/android/provider/ContactsContract.html) for the Contacts Provider. The reference documentation for this class includes example code snippets. One of its subclasses, `ContactsContract.Intents.Insert` (/reference/android/provider/ContactsContract.Intents.Insert.html), is a contract class that contains constants for intents and intent data.

MIME Type Reference

Content providers can return standard MIME media types, or custom MIME type strings, or both.

MIME types have the format

```
type/subtype
```

For example, the well-known MIME type `text/html` has the `text` type and the `html` subtype. If the provider returns this type for a URI, it means that a query using that URI will return text containing HTML tags.

Custom MIME type strings, also called "vendor-specific" MIME types, have more complex *type* and *subtype* values. The *type* value is always

```
vnd.android.cursor.dir
```

for multiple rows, or

```
vnd.android.cursor.item
```

for a single row.

The *subtype* is provider-specific. The Android built-in providers usually have a simple subtype. For example, the when the Contacts application creates a row for a telephone number, it sets the following MIME type in the row:

```
vnd.android.cursor.item/phone_v2
```

Notice that the subtype value is simply `phone_v2`.

Other provider developers may create their own pattern of subtypes based on the provider's authority and table names. For example, consider a provider that contains train timetables. The provider's authority is `com.example.trains`, and it contains the tables Line1, Line2, and Line3. In response to the content URI

```
content://com.example.trains/Line1
```

for table Line1, the provider returns the MIME type

```
vnd.android.cursor.dir/vnd.example.line1
```

In response to the content URI

```
content://com.example.trains/Line2/5
```

for row 5 in table Line2, the provider returns the MIME type

```
vnd.android.cursor.item/vnd.example.line2
```

Most content providers define contract class constants for the MIME types they use. The Contacts Provider contract class `ContactsContract.RawContacts` `(/reference/android/provider/ContactsContract.RawContacts.html)`, for example, defines the constant `CONTENT_ITEM_TYPE` `(/reference/android/provider/ContactsContract.RawContacts.html#CONTENT_ITEM_TYPE)` for the MIME type of a single raw contact row.

Content URIs for single rows are described in the section Content URIs (#ContentURIs).