Creating a Content Provider

A content provider manages access to a central repository of data. You implement a provider as one or more classes in an Android application, along with elements in the manifest file. One of your classes implements a subclass `ContentProvider` `(/reference/android/content/ContentProvider.html)`, which is the interface between your provider and other applications. Although content providers are meant to make data available to other applications, you may of course have activities in your application that allow the user to query and modify the data managed by your provider.

The rest of this topic is a basic list of steps for building a content provider and a list of APIs to use.

IN THIS DOCUMENT

KEY CLASSES

- `ContentProvider`
- `Cursor`
- `Uri`

RELATED SAMPLES

- Note Pad sample application

SEE ALSO

- Content Provider Basics
- Calendar Provider

Before You Start Building

Before you start building a provider, do the following:

1. **Decide if you need a content provider**. You need to build a content provider if you want to provide one or more of the following features:
   - You want to offer complex data or files to other applications.
   - You want to allow users to copy complex data from your app into other apps.
   - You want to provide custom search suggestions using the search framework.

   You *don't* need a provider to use an SQLite database if the use is entirely within your own application.
2. If you haven't done so already, read the topic Content Provider Basics to learn more about providers.

Next, follow these steps to build your provider:

1. Design the raw storage for your data. A content provider offers data in two ways:

   File data
   : Data that normally goes into files, such as photos, audio, or videos. Store the files in your application's private space. In response to a request for a file from another application, your provider can offer a handle to the file.

   "Structured" data
   : Data that normally goes into a database, array, or similar structure. Store the data in a form that's compatible with tables of rows and columns. A row represents an entity, such as a person or an item in inventory. A column represents some data for the entity, such a person's name or an item's price. A common way to store this type of data is in an SQLite database, but you can use any type of persistent storage. To learn more about the storage types available in the Android system, see the section Designing Data Storage.
2. Define a concrete implementation of the `ContentProvider` class and its required methods. This class is the interface between your data and the rest of the Android system. For more information about this class, see the section Implementing the ContentProvider Class.
3. Define the provider's authority string, its content URIs, and column names. If you want the provider's application to handle intents, also define intent actions, extras data, and flags. Also define the permissions that you will require for applications that want to access your data. You should consider

defining all of these values as constants in a separate contract class; later, you can expose this class to other developers. For more information about content URIs, see the section Designing Content URIs. For more information about intents, see the section Intents and Data Access.

4. Add other optional pieces, such as sample data or an implementation of `AbstractThreadedSyncAdapter` that can synchronize data between the provider and cloud-based data.

Designing Data Storage

A content provider is the interface to data saved in a structured format. Before you create the interface, you must decide how to store the data. You can store the data in any form you like, and then design the interface to read and write the data as necessary.

These are some of the data storage technologies that are available in Android:

- The Android system includes an SQLite database API that Android's own providers use to store table-oriented data. The `SQLiteOpenHelper` class helps you create databases, and the `SQLiteDatabase` class is the base class for accessing databases.

  Remember that you don't have to use a database to implement your repository. A provider appears externally as a set of tables, similar to a relational database, but this is not a requirement for the provider's internal implementation.

- For storing file data, Android has a variety of file-oriented APIs. To learn more about file storage, read the topic Data Storage. If you're designing a provider that offers media-related data such as music or videos, you can have a provider that combines table data and files.
- For working with network-based data, use classes in `java.net` and `android.net`. You can also synchronize network-based data to a local data store such as a database, and then offer the data as tables or files. The Sample Sync Adapter sample application demonstrates this type of synchronization.

Data design considerations

Here are some tips for designing your provider's data structure:

- Table data should always have a "primary key" column that the provider maintains as a unique numeric value for each row. You can use this value to link the row to related rows in other tables (using it as a "foreign key"). Although you can use any name for this column, using `BaseColumns._ID` is the best choice, because linking the results of a provider query to a `ListView` requires one of the retrieved columns to have the name `_ID`.
- If you want to provide bitmap images or other very large pieces of file-oriented data, store the data in a file and then provide it indirectly rather than storing it directly in a table. If you do this, you need to tell users of your provider that they need to use a `ContentResolver` file method to access the data.
- Use the Binary Large OBject (BLOB) data type to store data that varies in size or has a varying structure. For example, you can use a BLOB column to store a protocol buffer or JSON structure.

  You can also use a BLOB to implement a *schema-independent* table. In this type of table, you define a primary key column, a MIME type column, and one or more generic columns as BLOB. The meaning of the data in the BLOB columns is indicated by the value in the MIME type column. This allows you to store different row types in the same table. The Contacts Provider's "data" table `ContactsContract.Data` `(/reference/android/provider/ContactsContract.Data.html)` is an example of a schema-independent table.

Designing 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 or file (a **path**). The optional id part points to an individual row in a table. Every data access method of `ContentProvider` `(/reference/android/content/ContentProvider.html)` has a content URI as an argument; this allows you to determine the table, row, or file to access.

The basics of content URIs are described in the topic Content Provider Basics (/guide/topics/providers/content-provider-basics.html).

Designing an authority

A provider usually has a single authority, which serves as its Android-internal name. To avoid conflicts with other providers, you should use Internet domain ownership (in reverse) as the basis of your provider authority. Because this recommendation is also true for Android package names, you can define your provider authority as an extension of the name of the package containing the provider. For example, if your Android package name is `com.example.<appname>`, you should give your provider the authority `com.example.<appname>.provider`.

Designing a path structure

Developers usually create content URIs from the authority by appending paths that point to individual tables. For example, if you have two tables *table1* and *table2*, you combine the authority from the previous example to yield the content URIs `com.example.<appname>.provider/table1` and `com.example.<appname>.provider/table2`. Paths aren't limited to a single segment, and there doesn't have to be a table for each level of the path.

Handling content URI IDs

By convention, providers offer access to a single row in a table by accepting a content URI with an ID value for the row at the end of the URI. Also by convention, providers match the ID value to the table's `_ID` column, and perform the requested access against the row that matches.

This convention facilitates a common design pattern for apps accessing a provider. The app does a query against the provider and displays the resulting `Cursor` (/reference/android/database/Cursor.html) in a `ListView` (/reference/android/widget/ListView.html) using a `CursorAdapter` (/reference/android/widget/CursorAdapter.html). The definition of `CursorAdapter` (/reference/android/widget/CursorAdapter.html) requires one of the columns in the `Cursor` (/reference/android/database/Cursor.html) to be `_ID`

The user then picks one of the displayed rows from the UI in order to look at or modify the data. The app gets the corresponding row from the `Cursor` (/reference/android/database/Cursor.html) backing the `ListView` (/reference/android/widget/ListView.html), gets the `_ID` value for this row, appends it to the content URI, and sends the access request to the provider. The provider can then do the query or modification against the exact row the user picked.

Content URI patterns

To help you choose which action to take for an incoming content URI, the provider API includes the convenience class `UriMatcher` (/reference/android/content/UriMatcher.html), which maps content URI "patterns" to integer values. You can use the integer values in a `switch` statement that chooses the desired action for the content URI or URIs that match a particular pattern.

A content URI pattern matches content URIs using wildcard characters:

- `*`: Matches a string of any valid characters of any length.
- `#`: Matches a string of numeric characters of any length.

As an example of designing and coding content URI handling, consider a provider with the authority `com.example.app.provider` that recognizes the following content URIs pointing to tables:

- `content://com.example.app.provider/table1`: A table called `table1`.
- `content://com.example.app.provider/table2/dataset1`: A table called `dataset1`.
- `content://com.example.app.provider/table2/dataset2`: A table called `dataset2`.
- `content://com.example.app.provider/table3`: A table called `table3`.

The provider also recognizes these content URIs if they have a row ID appended to them, as for example `content://com.example.app.provider/table3/1` for the row identified by `1` in `table3`.

The following content URI patterns would be possible:

`content://com.example.app.provider/*`
    Matches any content URI in the provider.
`content://com.example.app.provider/table2/*`:

Matches a content URI for the tables `dataset1` and `dataset2`, but doesn't match content URIs for `table1` or `table3`.

`content://com.example.app.provider/table3/#`: Matches a content URI for single rows in `table3`, such as `content://com.example.app.provider/table3/6` for the row identified by `6`.

The following code snippet shows how the methods in `UriMatcher` (/reference/android/content/UriMatcher.html) work. This code handles URIs for an entire table differently from URIs for a single row, by using the content URI pattern `content://<authority>/<path>` for tables, and `content://<authority>/<path>/<id>` for single rows.

The method `addURI()` (/reference/android/content/UriMatcher.html#addURI(java.lang.String, java.lang.String, int)) maps an authority and path to an integer value. The method `match()` (/reference/android/content/UriMatcher.html#match(android.net.Uri)) returns the integer value for a URI. A `switch` statement chooses between querying the entire table, and querying for a single record:

```
public class ExampleProvider extends ContentProvider {
...
    // Creates a UriMatcher object.
    private static final UriMatcher sUriMatcher;
...
    /*
     * The calls to addURI() go here, for all of the content URI patterns that the pro
     * should recognize. For this snippet, only the calls for table 3 are shown.
     */
...
    /*
     * Sets the integer value for multiple rows in table 3 to 1. Notice that no wildca
     * in the path
     */
    sUriMatcher.addURI("com.example.app.provider", "table3", 1);

    /*
     * Sets the code for a single row to 2. In this case, the "#" wildcard is
     * used. "content://com.example.app.provider/table3/3" matches, but
     * "content://com.example.app.provider/table3 doesn't.
     */
    sUriMatcher.addURI("com.example.app.provider", "table3/#", 2);
...
    // Implements ContentProvider.query()
    public Cursor query(
        Uri uri,
        String[] projection,
        String selection,
        String[] selectionArgs,
        String sortOrder) {
...
        /*
         * Choose the table to query and a sort order based on the code returned for t
         * URI. Here, too, only the statements for table 3 are shown.
         */
        switch (sUriMatcher.match(uri)) {


            // If the incoming URI was for all of table3
            case 1:

                if (TextUtils.isEmpty(sortOrder)) sortOrder = "_ID ASC";
                break;

            // If the incoming URI was for a single row
            case 2:

                /*
                 * Because this URI was for a single row, the _ID value part is
                 * present. Get the last path segment from the URI; this is the _ID va
                 * Then, append the value to the WHERE clause for the query
                 */
                selection = selection + "_ID = " uri.getLastPathSegment();
                break;

            default:
            ...
                // If the URI is not recognized, you should do some error handling he
        }
        // call the code to actually do the query
    }
```

Another class, `ContentUris` (/reference/android/content/ContentUris.html), provides convenience methods for working with the `id` part of content URIs. The classes `Uri` (/reference/android/net/Uri.html) and `Uri.Builder` (/reference/android/net/Uri.Builder.html) include convenience methods for parsing existing `Uri` (/reference/android/net/Uri.html) objects and building new ones.

Implementing the ContentProvider Class

The `ContentProvider` (/reference/android/content/ContentProvider.html) instance manages access to a structured set of data by handling requests from other applications. All forms of access eventually call `ContentResolver` (/reference/android/content/ContentResolver.html), which then calls a concrete method of `ContentProvider` (/reference/android/content/ContentProvider.html) to get access.

Required methods

The abstract class `ContentProvider` (/reference/android/content/ContentProvider.html) defines six abstract methods that you must implement as part of your own concrete subclass. All of these methods except `onCreate()` (/reference/android/content/ContentProvider.html#onCreate()) are called by a client application that is attempting to access your content provider:

`query()`
: Retrieve data from your provider. Use the arguments to select the table to query, the rows and columns to return, and the sort order of the result. Return the data as a `Cursor` object.

`insert()`
: Insert a new row into your provider. Use the arguments to select the destination table and to get the column values to use. Return a content URI for the newly-inserted row.

`update()`
: Update existing rows in your provider. Use the arguments to select the table and rows to update and to get the updated column values. Return the number of rows updated.

`delete()`
: Delete rows from your provider. Use the arguments to select the table and the rows to delete. Return the number of rows deleted.

`getType()`
: Return the MIME type corresponding to a content URI. This method is described in more detail in the section Implementing Content Provider MIME Types.

`onCreate()`
: Initialize your provider. The Android system calls this method immediately after it creates your provider. Notice that your provider is not created until a `ContentResolver` object tries to access it.

Notice that these methods have the same signature as the identically-named `ContentResolver` (/reference/android/content/ContentResolver.html) methods.

Your implementation of these methods should account for the following:

- All of these methods except onCreate() can be called by multiple threads at once, so they must be thread-safe. To learn more about multiple threads, see the topic Processes and Threads.
- Avoid doing lengthy operations in onCreate(). Defer initialization tasks until they are actually needed. The section Implementing the onCreate() method discusses this in more detail.
- Although you must implement these methods, your code does not have to do anything except return the expected data type. For example, you may want to prevent other applications from inserting data into some tables. To do this, you can ignore the call to insert() and return 0.

Implementing the query() method

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 must return a Cursor (/reference/android/database/Cursor.html) object, or if it fails, throw an Exception (/reference/java/lang/Exception.html). If you are using an SQLite database as your data storage, you can simply return the Cursor (/reference/android/database/Cursor.html) returned by one of the query() methods of the SQLiteDatabase (/reference/android/database/sqlite/SQLiteDatabase.html) class. If the query does not match any rows, you should return a Cursor (/reference/android/database/Cursor.html) instance whose getCount() (/reference/android/database/Cursor.html#getCount()) method returns 0. You should return null only if an internal error occurred during the query process.

If you aren't using an SQLite database as your data storage, use one of the concrete subclasses of Cursor (/reference/android/database/Cursor.html). For example, the MatrixCursor (/reference/android/database/MatrixCursor.html) class implements a cursor in which each row is an array of Object (/reference/java/lang/Object.html). With this class, use addRow() (/reference/android/database/MatrixCursor.html#addRow(java.lang.Object[])) to add a new row.

Remember that the Android system must be able to communicate the Exception (/reference/java/lang/Exception.html) across process boundaries. Android can do this for the following exceptions that may be useful in handling query errors:

- IllegalArgumentException (You may choose to throw this if your provider receives an invalid content URI)
- NullPointerException

Implementing the insert() method

The insert() (/reference/android/content/ContentProvider.html#insert(android.net.Uri, android.content.ContentValues)) method adds a new row to the appropriate table, using the values in the ContentValues (/reference/android/content/ContentValues.html) argument. If a column name is not in the ContentValues (/reference/android/content/ContentValues.html) argument, you may want to provide a default value for it either in your provider code or in your database schema.

This method should return the content URI for the new row. To construct this, append the new row's _ID (or other primary key) value to the table's content URI, using withAppendedId() (/reference/android/content/ContentUris.html#withAppendedId(android.net.Uri, long)).

Implementing the delete() method

The delete() (/reference/android/content/ContentProvider.html#delete(android.net.Uri, java.lang.String, java.lang.String[])) method does not have to physically delete rows from your data storage. If you are using a sync adapter with your provider, you should consider marking a deleted row with a "delete" flag rather than removing the row entirely. The sync adapter can check for deleted rows and remove them from the server before deleting them from the provider.

Implementing the update() method

The update() (/reference/android/content/ContentProvider.html#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[])) method takes the same ContentValues (/reference/android/content/ContentValues.html) argument used by insert() (/reference/android/content/ContentProvider.html#insert(android.net.Uri, android.content.ContentValues)), and the same selection and selectionArgs arguments used by delete() (/reference/android/content/ContentProvider.html#delete(android.net.Uri, java.lang.String, java.lang.String[])) and ContentProvider.query() (/reference/android/content/ContentProvider.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String)). This may allow you to re-use code between these methods.

Implementing the onCreate() method

The Android system calls onCreate() (/reference/android/content/ContentProvider.html#onCreate()) when it starts up the provider. You should perform only fast-running initialization tasks in this method, and defer database creation and data loading until the provider actually receives a request for the data. If you do lengthy tasks in onCreate() (/reference/android/content/ContentProvider.html#onCreate()), you will slow down your provider's startup. In turn, this will slow down the response from the provider to other applications.

For example, if you are using an SQLite database you can create a new SQLiteOpenHelper (/reference/android/database/sqlite/SQLiteOpenHelper.html) object in ContentProvider.onCreate() (/reference/android/content/ContentProvider.html#onCreate()), and then create the SQL tables the first time you open the database. To facilitate this, the first time you call getWritableDatabase() (/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase()), it automatically calls the SQLiteOpenHelper.onCreate() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onCreate(android.database.sqlite.SQLiteDatabase)) method.

The following two snippets demonstrate the interaction between ContentProvider.onCreate() (/reference/android/content/ContentProvider.html#onCreate()) and SQLiteOpenHelper.onCreate() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onCreate(android.database.sqlite.SQLiteDatabase)). The first snippet is the implementation of ContentProvider.onCreate() (/reference/android/content/ContentProvider.html#onCreate()):

```
public class ExampleProvider extends ContentProvider

    /*
     * Defines a handle to the database helper object. The MainDatabaseHelper class is
     * in a following snippet.
     */
    private MainDatabaseHelper mOpenHelper;

    // Defines the database name
    private static final String DBNAME = "mydb";

    // Holds the database object
    private SQLiteDatabase db;

    public boolean onCreate() {

        /*
         * Creates a new helper object. This method always returns quickly.
         * Notice that the database itself isn't created or opened
         * until SQLiteOpenHelper.getWritableDatabase is called
         */
        mOpenHelper = new SQLiteOpenHelper(
            getContext(),        // the application context
            DBNAME,              // the name of the database)
            null,                // uses the default SQLite cursor
            1                    // the version number
        );

        return true;
    }
    ...

    // Implements the provider's insert method
    public Cursor insert(Uri uri, ContentValues values) {
        // Insert code here to determine which table to open, handle error-checking, a

        ...

        /*
         * Gets a writeable database. This will trigger its creation if it doesn't alr
         *
         */
        db = mOpenHelper.getWritableDatabase();
    }
}
```

The next snippet is the implementation of SQLiteOpenHelper.onCreate() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onCreate(android.database.sqlite.SQLiteDatabase)), including a helper class:

```
...
// A string that defines the SQL statement for creating a table
private static final String SQL_CREATE_MAIN = "CREATE TABLE " +
    "main " +                       // Table's name
    "(" +                           // The columns in the table
    " _ID INTEGER PRIMARY KEY, " +
    " WORD TEXT"
    " FREQUENCY INTEGER " +
    " LOCALE TEXT )";
...
/**
 * Helper class that actually creates and manages the provider's underlying data repos
 */
protected static final class MainDatabaseHelper extends SQLiteOpenHelper {

    /*
     * Instantiates an open helper for the provider's SQLite data repository
     * Do not do database creation and upgrade here.
     */
    MainDatabaseHelper(Context context) {
        super(context, DBNAME, null, 1);
    }

    /*
     * Creates the data repository. This is called when the provider attempts to open
     * repository and SQLite reports that it doesn't exist.
     */
    public void onCreate(SQLiteDatabase db) {

        // Creates the main table
        db.execSQL(SQL_CREATE_MAIN);
    }
}
```

Implementing ContentProvider MIME Types

The ContentProvider (/reference/android/content/ContentProvider.html) class has two methods for returning MIME types:

`getType()`
: One of the required methods that you must implement for any provider.

`getStreamTypes()`
: A method that you're expected to implement if your provider offers files.

MIME types for tables

The `getType()` (/reference/android/content/ContentProvider.html#getType(android.net.Uri)) method returns a `String` (/reference/java/lang/String.html) in MIME format that describes the type of data returned by the content URI argument. The `Uri` (/reference/android/net/Uri.html) argument can be a pattern rather than a specific URI; in this case, you should return the type of data associated with content URIs that match the pattern.

For common types of data such as as text, HTML, or JPEG, `getType()` (/reference/android/content/ContentProvider.html#getType(android.net.Uri)) should return the standard MIME type for that data. A full list of these standard types is available on the IANA MIME Media Types (http://www.iana.org/assignments/media-types/index.htm) website.

For content URIs that point to a row or rows of table data, `getType()` (/reference/android/content/ContentProvider.html#getType(android.net.Uri)) should return a MIME type in Android's vendor-specific MIME format:

- Type part: `vnd`
- Subtype part:
  - If the URI pattern is for a single row: `android.cursor.item/`
  - If the URI pattern is for more than one row: `android.cursor.dir/`
- Provider-specific part: `vnd.<name>.<type>`

You supply the `<name>` and `<type>`. The `<name>` value should be globally unique, and the `<type>` value should be unique to the corresponding URI pattern. A good choice for `<name>` is your company's name or some part of your application's Android package name. A good choice for the `<type>` is a string that identifies the table associated with the URI.

For example, if a provider's authority is `com.example.app.provider`, and it exposes a table named `table1`, the MIME type for multiple rows in `table1` is:

```
vnd.android.cursor.dir/vnd.com.example.provider.table1
```

For a single row of `table1`, the MIME type is:

```
vnd.android.cursor.item/vnd.com.example.provider.table1
```

MIME types for files

If your provider offers files, implement `getStreamTypes()` (/reference/android/content/ContentProvider.html#getStreamTypes(android.net.Uri, java.lang.String)). The method returns a `String` (/reference/java/lang/String.html) array of MIME types for the files your provider can return for a given content URI. You should filter the MIME types you offer by the MIME type filter argument, so that you return only those MIME types that the client wants to handle.

For example, consider a provider that offers photo images as files in `.jpg`, `.png`, and `.gif` format. If an application calls `ContentResolver.getStreamTypes()` (/reference/android/content/ContentResolver.html#getStreamTypes(android.net.Uri, java.lang.String)) with the filter string `image/*` (something that is an "image"), then the `ContentProvider.getStreamTypes()` (/reference/android/content/ContentProvider.html#getStreamTypes(android.net.Uri, java.lang.String)) method should return the array:

```
{ "image/jpeg", "image/png", "image/gif"}
```

If the app is only interested in `.jpg` files, then it can call `ContentResolver.getStreamTypes()` (/reference/android/content/ContentResolver.html#getStreamTypes(android.net.Uri, java.lang.String)) with the filter string `*\/jpeg`, and `ContentProvider.getStreamTypes()` (/reference/android/content/ContentProvider.html#getStreamTypes(android.net.Uri, java.lang.String)) should return:

```
{"image/jpeg"}
```

If your provider doesn't offer any of the MIME types requested in the filter string, `getStreamTypes()` (/reference/android/content/ContentProvider.html#getStreamTypes(android.net.Uri, java.lang.String)) should return `null`.

Implementing a Contract Class

A contract class is a `public final` class that contains constant definitions for the URIs, column names, MIME types, and other meta-data that pertain to the provider. The class establishes a contract between the provider and other applications by ensuring that the provider can be correctly accessed even if there are changes to the actual values of URIs, column names, and so forth.

A contract class also helps developers because it usually has mnemonic names for its constants, so developers are less likely to use incorrect values for column names or URIs. Since it's a class, it can contain Javadoc documentation. Integrated development environments such as Eclipse can auto-complete constant names from the contract class and display Javadoc for the constants.

Developers can't access the contract class's class file from your application, but they can statically compile it into their application from a `.jar` file you provide.

The `ContactsContract` (/reference/android/provider/ContactsContract.html) class and its nested classes are examples of contract classes.

Implementing Content Provider Permissions

Permissions and access for all aspects of the Android system are described in detail in the topic Security and Permissions (/guide/topics/security/security.html). The topic Data Storage (/guide/topics/data/data-storage.html) also described the security and permissions in effect for various types of storage. In brief, the important points are:

- By default, data files stored on the device's internal storage are private to your application and provider.
- `SQLiteDatabase` databases you create are private to your application and provider.
- By default, data files that you save to external storage are *public* and *world-readable*. You can't use a content provider to restrict access to files in external storage, because other applications can use other API calls to read and write them.
- The method calls for opening or creating files or SQLite databases on your device's internal storage can potentially give both read and write access to all other applications. If you use an internal file or database as your provider's repository, and you give it "world-readable" or "world-writeable" access, the permissions you set for your provider in its manifest won't protect your data. The default access for files and databases in internal storage is "private", and for your provider's repository you shouldn't change this.

If you want to use content provider permissions to control access to your data, then you should store your data in internal files, SQLite databases, or the "cloud" (for example, on a remote server), and you should keep files and databases private to your application.

Implementing permissions

All applications can read from or write to your provider, even if the underlying data is private, because by default your provider does not have permissions set. To change this, set permissions for your provider in your manifest file, using attributes or child elements of the <provider> (/guide/topics/manifest/provider-element.html) element. You can set permissions that apply to the entire provider, or to certain tables, or even to certain records, or all three.

You define permissions for your provider with one or more <permission> (/guide/topics/manifest/permission-element.html) elements in your manifest file. To make the permission unique to your provider, use Java-style scoping for the android:name (/guide/topics/manifest/permission-element.html#nm) attribute. For example, name the read permission com.example.app.provider.permission.READ_PROVIDER.

The following list describes the scope of provider permissions, starting with the permissions that apply to the entire provider and then becoming more fine-grained. More fine-grained permissions take precedence over ones with larger scope:

Single read-write provider-level permission
: One permission that controls both read and write access to the entire provider, specified with the android:permission attribute of the <provider> element.

Separate read and write provider-level permission
: A read permission and a write permission for the entire provider. You specify them with the android:readPermission and android:writePermission attributes of the <provider> element. They take precedence over the permission required by android:permission.

Path-level permission
: Read, write, or read/write permission for a content URI in your provider. You specify each URI you want to control with a <path-permission> child element of the <provider> element. For each content URI you specify, you can specify a read/write permission, a read permission, or a write permission, or all three. The read and write permissions take precedence over the read/write permission. Also, path-level permission takes precedence over provider-level permissions.

Temporary permission
: A permission level that grants temporary access to an application, even if the application doesn't have the permissions that are normally required. The temporary access feature reduces the number of permissions an application has to request in its manifest. When you turn on temporary permissions, the only applications that need "permanent" permissions for your provider are ones that continually access all your data.

  Consider the permissions you need to implement an email provider and app, when you want to allow an outside image viewer application to display photo attachments from your provider. To give the image viewer the necessary access without requiring permissions, set up temporary permissions for content URIs for photos. Design your email app so that when the user wants to display a photo, the app sends an intent containing the photo's content URI and permission flags to the image viewer. The image viewer can then query your email provider to retrieve the photo, even though the viewer doesn't have the normal read permission for your provider.

  To turn on temporary permissions, either set the android:grantUriPermissions (/guide/topics/manifest/provider-element.html#gprmsn) attribute of the <provider> (/guide/topics/manifest/provider-element.html) element, or add one or more <grant-uri-permission> (/guide/topics/manifest/grant-uri-permission-element.html) child elements to your <provider> (/guide/topics/manifest/provider-element.html) element. If you use temporary permissions, you have to call Context.revokeUriPermission() (/reference/android/content/Context.html#revokeUriPermission(android.net.Uri, int)) whenever you remove support for a content URI from your provider, and the content URI is associated with a temporary permission.

  The attribute's value determines how much of your provider is made accessible. If the attribute is set to true, then the system will grant temporary permission to your entire provider, overriding any other permissions that are required by your provider-level or path-level permissions.

  If this flag is set to false, then you must add <grant-uri-permission> (/guide/topics/manifest/grant-uri-permission-element.html) child elements to your <provider> (/guide/topics/manifest/provider-element.html) element. Each child element specifies the content URI or URIs for which temporary access is granted.

  To delegate temporary access to an application, an intent must contain the

`FLAG_GRANT_READ_URI_PERMISSION` (/reference/android/content/Intent.html#FLAG_GRANT_READ_URI_PERMISSION) or the `FLAG_GRANT_WRITE_URI_PERMISSION` (/reference/android/content/Intent.html#FLAG_GRANT_WRITE_URI_PERMISSION) flags, or both. These are set with the `setFlags()` (/reference/android/content/Intent.html#setFlags(int)) method.

If the `android:grantUriPermissions` (/guide/topics/manifest/provider-element.html#gprmsn) attribute is not present, it's assumed to be `false`.

The <provider> Element

Like `Activity` (/reference/android/app/Activity.html) and `Service` (/reference/android/app/Service.html) components, a subclass of `ContentProvider` (/reference/android/content/ContentProvider.html) must be defined in the manifest file for its application, using the `<provider>` (/guide/topics/manifest/provider-element.html) element. The Android system gets the following information from the element:

Authority (`android:authorities`)
: Symbolic names that identify the entire provider within the system. This attribute is described in more detail in the section Designing Content URIs.

Provider class name (`android:name`)
: The class that implements `ContentProvider`. This class is described in more detail in the section Implementing the ContentProvider Class.

Permissions
: Attributes that specify the permissions that other applications must have in order to access the provider's data:

- `android:grantUriPermssions`: Temporary permission flag.
- `android:permission`: Single provider-wide read/write permission.
- `android:readPermission`: Provider-wide read permission.
- `android:writePermission`: Provider-wide write permission.

Permissions and their corresponding attributes are described in more detail in the section Implementing Content Provider Permissions (#Permissions).

Startup and control attributes
: These attributes determine how and when the Android system starts the provider, the process characteristics of the provider, and other run-time settings:

- `android:enabled`: Flag allowing the system to start the provider.
- `android:exported`: Flag allowing other applications to use this provider.
- `android:initOrder`: The order in which this provider should be started, relative to other providers in the same process.
- `android:multiProcess`: Flag allowing the system to start the provider in the same process as the calling client.
- `android:process`: The name of the process in which the provider should run.
- `android:syncable`: Flag indicating that the provider's data is to be sync'ed with data on a server.

The attributes are fully documented in the dev guide topic for the `<provider>` (/guide/topics/manifest/provider-element.html) element.

Informational attributes
: An optional icon and label for the provider:

- `android:icon`: A drawable resource containing an icon for the provider. The icon appears next to the provider's label in the list of apps in *Settings* > *Apps* > *All*.
- `android:label`: An informational label describing the provider or its data, or both. The label appears in the list of apps in *Settings* > *Apps* > *All*.

The attributes are fully documented in the dev guide topic for the `<provider>` (/guide/topics/manifest/provider-element.html) element.

Intents and Data Access

Applications can access a content provider indirectly with an `Intent` (/reference/android/content/Intent.html). The application does not call any of the methods of `ContentResolver` (/reference/android/content/ContentResolver.html) or `ContentProvider` (/reference/android/content/ContentProvider.html). Instead, it sends an intent that starts an activity, which is often part of the provider's own application. The destination activity is in charge of retrieving and displaying the data in its UI. Depending on the action in the intent, the destination activity may also prompt the user to make modifications to the provider's data. An intent may also contain "extras" data that the destination activity displays in the UI; the user then has the option of changing this data before using it to modify the data in the provider.

You may want to use intent access to help ensure data integrity. Your provider may depend on having data inserted, updated, and deleted according to strictly defined business logic. If this is the case, allowing other applications to directly modify your data may lead to invalid data. If you want developers to use intent access, be sure to document it thoroughly. Explain to them why intent access using your own application's UI is better than trying to modify the data with their code.

Handling an incoming intent that wishes to modify your provider's data is no different from handling other intents. You can learn more about using intents by reading the topic Intents and Intent Filters (/guide/components/intents-filters.html).