Creating a New Content Provider

Create a new Content Provider by extending the abstract ContentProvider class. Override the onCreate method to open or initialize the underlying data source you're exposing with this new provider. The skeleton code for a new Content Provider is shown below:

```
import android.content.*;
import android.database.Cursor;
import android.net.Uri;
import android.database.SQLException;
public class MyProvider extends ContentProvider {
@Override
public boolean onCreate() {
// TODO: Construct the underlying database.
return true;
}
}
```

You should also expose a public static CONTENT_URI variable that returns the full URI to this provider. Content URIs must be unique between providers, so it's good practice to base the URI path on your package name. The general form for defi ning a Content Provider's URI is content://com.<CompanyName>.provider.<ApplicationName>/<DataPath>

For example:

content://com.paad.provider.myapp/items
Content URIs can represent either of two forms. The previous URI represents a request for all values of that type (e.g., all items).

Appending a trailing /<rownumber>, as shown below, represents a request for a single record (e.g., "the fifth item").
content://com.paad.provider.myapp/items/5

It's good form to support access to your provider using both these forms.

The simplest way to do this is using a UriMatcher. Configure the UriMatcher to parse URIs to determine their form when the provider is being accessed through a Content Resolver. The following snippet shows the skeleton code for this pattern:

```
public class MyProvider extends ContentProvider {
private static final String myURI =
"content://com.paad.provider.myapp/items";
public static final Uri CONTENT_URI = Uri.parse(myURI);
@Override
public boolean onCreate() {
// TODO: Construct the underlying database.
return true;
}
// Create the constants used to differentiate between the different
// URI requests.
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;
private static final UriMatcher uriMatcher;
// Populate the UriMatcher object, where a URI ending in 'items' will
// correspond to a request for all items, and 'items/[rowID]'
// represents a single row.
static {
uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
uriMatcher.addURI("com.paad.provider.myApp", "items", ALLROWS);
uriMatcher.addURI("com.paad.provider.myApp", "items/#",
SINGLE_ROW);
}
}
```

You can use the same technique to expose alternative URIs for different subsets of data, or different tables within your database from within the same Content Provider. It's also good practice to expose the names and indexes of the columns available in your provider to simplify extracting information from Cursor results.

Exposing Access to the Data Source

You can expose queries and transactions with your Content Provider by implementing the delete, insert, update, and query methods.

These methods act as a generic interface to the underlying data source, allowing Android applications to share data across application boundaries without having to publish separate interfaces for each application. The most common scenario is to use a Content Provider to expose a private SQLite Database, but within these methods you can access any source of data (including fi les or application instance variables).

The following code snippet shows the skeleton code for implementing queries and transactions for a Content Provider. Notice that the UriMatcher object is used to refi ne the transaction and query requests.

```
@Override
public Cursor query(Uri uri,
String[] projection,
String selection,
String[] selectionArgs,
String sort) {
// If this is a row query, limit the result set to the passed in row.
switch (uriMatcher.match(uri)) {
case SINGLE_ROW :
// TODO: Modify selection based on row id, where:
// rowNumber = uri.getPathSegments().get(1));
}
return null;
}
@Override
public Uri insert(Uri _uri, ContentValues _initialValues) {
long rowID = [ ... Add a new item ... ]
// Return a URI to the newly added item.
if (rowID > 0) {
return ContentUris.withAppendedId(CONTENT_URI, rowID);
}
throw new SQLException("Failed to add new item into " + _uri);
}
@Override
public int delete(Uri uri, String where, String[] whereArgs) {
switch (uriMatcher.match(uri)) {
case ALLROWS:
case SINGLE_ROW:
default: throw new IllegalArgumentException("Unsupported URI:" +
uri);
}
}
@Override
public int update(Uri uri, ContentValues values, String where,
String[] whereArgs) {
switch (uriMatcher.match(uri)) {
case ALLROWS:
case SINGLE_ROW:

default: throw new IllegalArgumentException("Unsupported URI:" +
uri);
}
}
```

The final step in creating a Content Provider is defining the MIME type that identifies the data the provider returns. Override the getType method to return a String that uniquely describes your data type. The type returned should include two forms, one for a single entry and another for all the entries, following the forms:

- **Single Item**

vnd.<companyname>.cursor.item/<contenttype>

- **All Items**

vnd.<companyName>.cursor.dir/<contenttype>

The following code snippet shows how to override the get Type method to return the correct MIME type based on the URI passed in:

```
@Override
public String getType(Uri _uri) {
switch (uriMatcher.match(_uri)) {
case ALLROWS: return “vnd.paad.cursor.dir/myprovidercontent”;
case SINGLE_ROW: return “vnd.paad.cursor.item/myprovidercontent”;
default: throw new IllegalArgumentException(“Unsupported URI: “ +
_uri);
}
}
```

Registering Your Provider

Once you have completed your Content Provider, it must be added to the application manifest. Use the authorities tag to specify its address, as shown in the following XML snippet:

```
<provider android:name=”MyProvider”
android:authorities=”com.paad.provider.myapp”/>
```