

59. Implementing an Android Content Provider in Android Studio

As outlined in the previous chapter, content providers provide a mechanism through which the data stored by one Android application can be made accessible to other applications. Having provided a theoretical overview of content providers, this chapter will continue the coverage of content providers by extending the Database project created in the chapter entitled [*“An Android TableLayout and TableRow Tutorial”*](#) to implement content provider based access to the database.

59.1 Copying the Database Project

In order to keep the original Database project intact, we will make a backup copy of the project before modifying it to implement content provider support for the application. If the Database project is currently open within Android Studio, close it using the *File -> Close Project* menu option.

Using the file system explorer for your operating system type, navigate to the directory containing your Android Studio projects (typically this will be a folder named *AndroidStudioProjects* located in your home directory). Within this folder, copy the Database project folder to a new folder named *DatabaseOriginal*.

Within the Android Studio welcome screen, select the *Open an existing Android Studio project* option from the Quick Start list and navigate to and select the original *Database* project so that it loads into the main window.

59.2 Adding the Content Provider Package

The next step is to add a new package to the Database project into which the content provider class will be created. Add this new package by navigating within the Project tool window to *app -> java*, right-clicking on the *java* entry and selecting the *New -> Package* menu option. When the *Choose Destination Directory* dialog appears, select the *..\app\src\main\java* option from the *Directory Structure* panel and click on OK.

In the *New Package* dialog, enter the following package name into the name field before clicking on the OK button:

```
com.ebookfrenzy.database.provider
```

The new package should now be listed within the Project tool window as illustrated in [Figure 59-1](#):

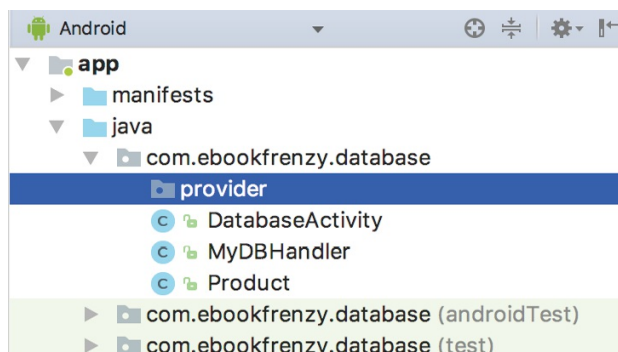


Figure 59-1

59.3 Creating the Content Provider Class

As discussed in [“Understanding Android Content Providers”](#), content providers are created by subclassing the `android.content.ContentProvider` class. Consequently, the next step is to add a class to the new *provider* package to serve as the content provider for this application. Locate the new package in the Project tool window, right-click on it and select the *New -> Other -> Content Provider* menu option. In the *Configure Component* dialog, enter *MyContentProvider* into the *Class Name* field and the following into the *URI Authorities* field:

```
com.ebookfrenzy.database.provider.MyContentProvider
```

Make sure that the new content provider class is both exported and enabled before clicking on *Finish* to create the new class.

Once the new class has been created, the *MyContentProvider.java* file should be listed beneath the *provider* package in the Project tool window and automatically loaded into the editor where it should appear as outlined in the following listing:

```
package com.ebookfrenzy.database.provider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;

public class MyContentProvider extends ContentProvider {
```

```

    public MyContentProvider() {
    }

    @Override
    public int delete(Uri uri, String selection, String[]
selectionArgs) {
        // Implement this to handle requests to delete one or more
rows.
        throw new UnsupportedOperationException("Not yet
implemented");
    }

    @Override
    public String getType(Uri uri) {
        // TODO: Implement this to handle requests for the MIME type
of the data
        // at the given URI.
        throw new UnsupportedOperationException("Not yet
implemented");
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        // TODO: Implement this to handle requests to insert a new
row.
        throw new UnsupportedOperationException("Not yet
implemented");
    }

    @Override
    public boolean onCreate() {
        // TODO: Implement this to initialize your content provider
on startup.
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String
selection,
        String[] selectionArgs, String sortOrder) {
        // TODO: Implement this to handle query requests from
clients.
        throw new UnsupportedOperationException("Not yet
implemented");
    }

```

```

    }

    @Override
    public int update(Uri uri, ContentValues values, String
selection,
        String[] selectionArgs) {
        // TODO: Implement this to handle requests to update one or
more rows.
        throw new UnsupportedOperationException("Not yet
implemented");
    }
}

```

As is evident from a quick review of the code in this file, Android Studio has already populated the class with stubs for each of the methods that a subclass of `ContentProvider` is required to implement. It will soon be necessary to begin implementing these methods, but first some constants relating to the provider's content authority and URI need to be declared.

59.4 Constructing the Authority and Content URI

As outlined in the previous chapter, all content providers must have associated with them an *authority* and a *content uri*. In practice, the authority is typically the full package name of the content provider class itself, in this case `com.ebookfrenzy.database.database.provider.MyContentProvider` as declared when the new Content Provider class was created in the previous section.

The content URI will vary depending on application requirements, but for the purposes of this example it will comprise the authority with the name of the database table appended at the end. Within the `MyContentProvider.java` file, make the following modifications:

```

package com.ebookfrenzy.database.provider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.content.UriMatcher;

public class MyContentProvider extends ContentProvider {

    private static final String AUTHORITY =

```

```

        "com.ebookfrenzy.database.provider.MyContentProvider";
    private static final String PRODUCTS_TABLE = "products";
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/" +
            PRODUCTS_TABLE);

    public MyContentProvider() {
    }
}

```

The above statements begin by creating a new String object named *AUTHORITY* and assigning the authority string to it. Similarly, a second String object named *PRODUCTS_TABLE* is created and initialized with the name of our database table (products).

Finally, these two string elements are combined, prefixed with *content://* and converted to a Uri object using the *parse()* method of the Uri class. The result is assigned to a variable named *CONTENT_URI*.

59.5 Implementing URI Matching in the Content Provider

When the methods of the content provider are called, they will be passed as an argument a URI indicating the data on which the operation is to be performed. This URI may take the form of a reference to a specific row in a specific table. It is also possible that the URI will be more general, for example specifying only the database table. It is the responsibility of each method to identify the *Uri type* and to act accordingly. This task can be eased considerably by making use of a UriMatcher instance. Once a UriMatcher instance has been created, it can be configured to return a specific integer value corresponding to the type of URI it detects when asked to do so. For the purposes of this tutorial, we will be configuring our UriMatcher instance to return a value of 1 when the URI references the entire products table, and a value of 2 when the URI references the ID of a specific row in the products table. Before working on creating the UriMatcher instance, we will first create two integer variables to represent the two URI types:

```
package com.ebookfrenzy.database.provider;
```

```

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.content.UriMatcher;

public class MyContentProvider extends ContentProvider {

    private static final String AUTHORITY =
        "com.ebookfrenzy.database.provider.MyContentProvider";
    private static final String PRODUCTS_TABLE = "products";
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/" +
            PRODUCTS_TABLE);

    public static final int PRODUCTS = 1;
    public static final int PRODUCTS_ID = 2;
    .
    .
}

```

With the Uri type variables declared, it is now time to add code to create a UriMatcher instance and configure it to return the appropriate variables:

```

public class MyContentProvider extends ContentProvider {

    private static final String AUTHORITY =
        "com.ebookfrenzy.database.provider.MyContentProvider";
    private static final String PRODUCTS_TABLE = "products";
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/" +
            PRODUCTS_TABLE);

    public static final int PRODUCTS = 1;
    public static final int PRODUCTS_ID = 2;

    private static final UriMatcher sURIMatcher =
        new UriMatcher(UriMatcher.NO_MATCH);

    static {
        sURIMatcher.addURI(AUTHORITY, PRODUCTS_TABLE, PRODUCTS);
        sURIMatcher.addURI(AUTHORITY, PRODUCTS_TABLE + "/#",
            PRODUCTS_ID);
    }

```

```
.  
.   
}
```

The UriMatcher instance (named sURIMatcher) is now primed to return the value of PRODUCTS when just the products table is referenced in a URI, and PRODUCTS_ID when the URI includes the ID of a specific row in the table.

59.6 Implementing the Content Provider onCreate() Method

When the content provider class is created and initialized, a call will be made to the *onCreate()* method of the class. It is within this method that any initialization tasks for the class need to be performed. For the purposes of this example, all that needs to be performed is for an instance of the MyDBHandler class implemented in [“An Android SQLite Database Tutorial”](#) to be created. Once this instance has been created, it will need to be accessible from the other methods in the class, so a declaration for the database handler also needs to be declared, resulting in the following code changes to the *MyContentProvider.java* file:

```
package com.ebookfrenzy.database.provider;  
  
import com.ebookfrenzy.database.MyDBHandler;  
  
import android.content.ContentProvider;  
import android.content.ContentValues;  
import android.database.Cursor;  
import android.net.Uri;  
import android.content.UriMatcher;  
import android.database.sqlite.SQLiteDatabase;  
import android.database.sqlite.SQLiteQueryBuilder;  
import android.text.TextUtils;  
  
public class MyContentProvider extends ContentProvider {  
  
    private MyDBHandler myDB;  
  
    .  
    .  
    .  
  
    @Override  
    public boolean onCreate() {
```

```

        myDB = new MyDBHandler(getContext(), null, null, 1);
        return false;
    }
}

```

59.7 Implementing the Content Provider insert() Method

When a client application or activity requests that data be inserted into the underlying database, the *insert()* method of the content provider class will be called. At this point, however, all that exists in the *MyContentProvider.java* file of the project is a stub method, which reads as follows:

```

@Override
public Uri insert(Uri uri, ContentValues values) {
    // TODO: Implement this to handle requests to insert a new row.
    throw new UnsupportedOperationException("Not yet implemented");
}

```

Passed as arguments to the method are a URI specifying the destination of the insertion and a ContentValues object containing the data to be inserted.

This method now needs to be modified to perform the following tasks:

- Use the *sUriMatcher* object to identify the URI type.
- Throw an exception if the URI is not valid.
- Obtain a reference to a writable instance of the underlying SQLite database.
- Perform a SQL insert operation to insert the data into the database table.
- Notify the corresponding content resolver that the database has been modified.
- Return the URI of the newly added table row.

Bringing these requirements together results in a modified *insert()* method, which reads as follows:

```

@Override
public Uri insert(Uri uri, ContentValues values) {

    int uriType = sURIMatcher.match(uri);

```



```

    SQLiteDatabase sqlDB = myDB.getWritableDatabase();

    long id = 0;
    switch (uriType) {
        case PRODUCTS:
            id = sqlDB.insert(MyDBHandler.TABLE_PRODUCTS,
                             null, values);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI: "
                                             + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return Uri.parse(PRODUCTS_TABLE + "/" + id);
}

```

59.8 Implementing the Content Provider query() Method

When a content provider is called upon to return data, the *query()* method of the provider class will be called. When called, this method is passed some or all of the following arguments:

- **URI** – The URI specifying the data source on which the query is to be performed. This can take the form of a general query with multiple results, or a specific query targeting the ID of a single table row.
- **Projection** – A row within a database table can comprise multiple columns of data. In the case of this application, for example, these correspond to the ID, product name and product quantity. The projection argument is simply a String array containing the name for each of the columns that is to be returned in the result data set.
- **Selection** – The “where” element of the selection to be performed as part of the query. This argument controls which rows are selected from the specified database. For example, if the query was required to select only products named “Cat Food” then the selection string passed to the *query()* method would read *productname = “Cat Food”*.
- **Selection Args** – Any additional arguments that need to be passed to the SQL query operation to perform the selection.

- **Sort Order** – The sort order for the selected rows.

When called, the *query()* method is required to perform the following operations:

- Use the *sUriMatcher* to identify the Uri type.
- Throw an exception if the URI is not valid.
- Construct a SQL query based on the criteria passed to the method. For convenience, the *SQLiteQueryBuilder* class can be used in construction of the query.
- Execute the query operation on the database.
- Notify the content resolver of the operation.
- Return a *Cursor* object containing the results of the query.

With these requirements in mind, the code for the *query()* method in the *MyContentProvider.java* file should now read as outlined in the following listing:

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder) {

    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
    queryBuilder.setTables(MyDBHandler.TABLE_PRODUCTS);

    int uriType = sURIMatcher.match(uri);

    switch (uriType) {
        case PRODUCTS_ID:
            queryBuilder.appendWhere(MyDBHandler.COLUMN_ID + "="
                + uri.getLastPathSegment());
            break;
        case PRODUCTS:
            break;
        default:
            throw new IllegalArgumentException("Unknown URI");
    }

    Cursor cursor = queryBuilder.query(myDB.getReadableDatabase(),
```

```

        projection, selection, selectionArgs, null, null,
        sortOrder);
    cursor.setNotificationUri(getContext().getContentResolver(),
        uri);
    return cursor;
}

```

59.9 Implementing the Content Provider update() Method

The *update()* method of the content provider is called when changes are being requested to existing database table rows. The method is passed a URI with the new values in the form of a ContentValues object and the usual selection argument strings.

When called, the *update()* method would typically perform the following steps:

- Use the sUriMatcher to identify the URI type.
- Throw an exception if the URI is not valid.
- Obtain a reference to a writable instance of the underlying SQLite database.
- Perform the appropriate update operation on the database, depending on the selection criteria and the URI type.
- Notify the content resolver of the database change.
- Return a count of the number of rows that were changed as a result of the update operation.

A general-purpose *update()* method, and the one we will use for this project, would read as follows:

```

@Override
public int update(Uri uri, ContentValues values, String selection,
    String[] selectionArgs) {

    int uriType = sURIMatcher.match(uri);
    SQLiteDatabase sqlDB = myDB.getWritableDatabase();
    int rowsUpdated = 0;

    switch (uriType) {

```

```

        case PRODUCTS:
            rowsUpdated =
                sqlDB.update(MyDBHandler.TABLE_PRODUCTS,
                            values,
                            selection,
                            selectionArgs);

            break;
        case PRODUCTS_ID:
            String id = uri.getLastPathSegment();
            if (TextUtils.isEmpty(selection)) {
                rowsUpdated =
                    sqlDB.update(MyDBHandler.TABLE_PRODUCTS,
                                values,
                                MyDBHandler.COLUMN_ID + "=" + id,
                                null);
            } else {
                rowsUpdated =
                    sqlDB.update(MyDBHandler.TABLE_PRODUCTS,
                                values,
                                MyDBHandler.COLUMN_ID + "=" + id
                                    + " and "
                                    + selection,
                                selectionArgs);
            }
            break;
        default:
            throw new IllegalArgumentException("Unknown URI: "
                + uri);
    }
    getContext().getContentResolver().notifyChange(uri,
        null);
    return rowsUpdated;
}

```

59.10

Implementing the Content Provider delete() Method

In common with a number of other content provider methods, the *delete()* method is passed a URI, a selection string and an optional set of selection arguments. A typical *delete()* method will also perform the following, and by now largely familiar, tasks when called:

- Use the `sUriMatcher` to identify the URI type.
- Throw an exception if the URI is not valid.
- Obtain a reference to a writable instance of the underlying SQLite database.
- Perform the appropriate delete operation on the database depending on the selection criteria and the Uri type.
- Notify the content resolver of the database change.
- Return the number of rows deleted as a result of the operation.

A typical *delete()* method is in many ways similar to the *update()* method and may be implemented as follows:

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs)
{
    int uriType = sURIMatcher.match(uri);
    SQLiteDatabase sqlDB = myDB.getWritableDatabase();
    int rowsDeleted = 0;

    switch (uriType) {
        case PRODUCTS:
            rowsDeleted = sqlDB.delete(MyDBHandler.TABLE_PRODUCTS,
                                      selection,
                                      selectionArgs);
            break;

        case PRODUCTS_ID:
            String id = uri.getLastPathSegment();
            if (TextUtils.isEmpty(selection)) {
                rowsDeleted =
sqlDB.delete(MyDBHandler.TABLE_PRODUCTS,
              MyDBHandler.COLUMN_ID + "=" + id,
              null);
            } else {
                rowsDeleted =
sqlDB.delete(MyDBHandler.TABLE_PRODUCTS,
              MyDBHandler.COLUMN_ID + "=" + id
                + " and " + selection,
              selectionArgs);
            }
            break;
    }
```

```

        default:
            throw new IllegalArgumentException("Unknown URI: " +
                uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return rowsDeleted;
}

```

With these methods implemented, the content provider class, in terms of the requirements for this example at least, is complete. The next step is to make sure that the content provider is declared in the project manifest file so that it is visible to any content resolvers seeking access to it.

59.1 Declaring the Content Provider in the Manifest File

Unless a content provider is declared in the manifest file of the application to which it belongs, it will not be possible for a content resolver to locate and access it. As outlined, content providers are declared using the <provider> tag and the manifest entry must correctly reference the content provider authority and content URI.

For the purposes of this project, therefore, locate the *AndroidManifest.xml* file for the DatabaseProvider project within the Project tool window and double-click on it to load it into the editing panel. Within the editing panel, make sure that the content provider declaration has already been added by Android Studio when the MyContentProvider class was added to the project:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.database" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".DatabaseActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

```

```

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

    <provider android:name=".provider.MyContentProvider"
        android:authorities=
            "com.ebookfrenzy.database.provider.MyContentProvider"
        android:enabled="true"
        android:exported="true" >
    </provider>
</application>

</manifest>

```

All that remains before testing the application is to modify the database handler class to use the content provider instead of directly accessing the database.

59.12 Modifying the Database Handler

When this application was originally created, it was designed to use a database handler to access the underlying database directly. Now that a content provider has been implemented, the database handler needs to be modified so that all database operations are performed using the content provider via a content resolver.

The first step is to modify the *MyDBHandler.java* class so that it obtains a reference to a *ContentResolver* instance. This can be achieved in the constructor method of the class:

```

package com.ebookfrenzy.database;

import com.ebookfrenzy.database.provider.MyContentProvider;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;
import android.content.ContentResolver;

public class MyDBHandler extends SQLiteOpenHelper {

    private ContentResolver myCR;

```

```

private static final int DATABASE_VERSION = 1;
private static final String DATABASE_NAME = "productDB.db";
private static final String TABLE_PRODUCTS = "products";

private static final String COLUMN_ID = "_id";
private static final String COLUMN_PRODUCTNAME = "productname";
private static final String COLUMN_QUANTITY = "quantity";

public MyDBHandler(Context context, String name,
                    SQLiteDatabase.CursorFactory factory, int version)
{
    super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    myCR = context.getContentResolver();
}
.
.
.
}

```

Next, the *addProduct()*, *findProduct()* and *removeProduct()* methods need to be rewritten to use the content resolver and content provider for data management purposes:

```

public void addProduct(Product product) {

    ContentValues values = new ContentValues();
    values.put(COLUMN_PRODUCTNAME, product.getProductName());
    values.put(COLUMN_QUANTITY, product.getQuantity());

    myCR.insert(MyContentProvider.CONTENT_URI, values);
}

public Product findProduct(String productname) {
    String[] projection = {COLUMN_ID,
        COLUMN_PRODUCTNAME, COLUMN_QUANTITY };

    String selection = "productname = \"" + productname +
        "\"";

    Cursor cursor = myCR.query(MyContentProvider.CONTENT_URI,
        projection, selection, null,
        null);
}

```



```

        Product product = new Product();

        if (cursor.moveToFirst()) {
            cursor.moveToFirst();
            product.setID(Integer.parseInt(cursor.getString(0)));
            product.setProductName(cursor.getString(1));
            product.setQuantity(
                Integer.parseInt(cursor.getString(2)));
            cursor.close();
        } else {
            product = null;
        }
        return product;
    }

    public boolean deleteProduct(String productname) {

        boolean result = false;

        String selection = "productname = \"" + productname +
            "\"";

        int rowsDeleted =
            myCR.delete(MyContentProvider.CONTENT_URI,
                selection, null);

        if (rowsDeleted > 0)
            result = true;

        return result;
    }

```

With the database handler class updated to use a content resolver and content provider, the application is now ready to be tested. Compile and run the application and perform some operations to add, find and remove product entries. In terms of operation and functionality, the application should behave exactly as it did when directly accessing the database, except that it is now using the content provider.

With the content provider now implemented and declared in the manifest file, any other applications can potentially access that data (since no permissions were declared, the default full access is in effect). The only information that the other applications need to know to gain access is the

content URI and the names of the columns in the products table.

59.13 Summary

The goal of this chapter was to provide a more detailed overview of the exact steps involved in implementing an Android content provider with a particular emphasis on the structure and implementation of the query, insert, delete and update methods of the content provider class. Practical use of the content resolver class to access data in the content provider was also covered, and the Database project was modified to make use of both a content provider and content resolver.