Android Insights - 3

Content Providers

Topics to be Covered

- Content Provider Basics
- Accessing a Content Provider
- Developing a Custom Content Provider

Content Provider

- A content provider manages access to a central repository of data..
- The 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.

Accessing a Content Provider

- A content provider offers methods which correspond to the basic CRUD functions of persistent storage.
- An application accesses the data from a content provider with a ContentResolver client object. This object has methods that call identically-named methods in the provider object.
- A content provider is identified by a content URI.

Accessing a Content Provider...

- Example of getting a list of words from the User Dictionary provider:

```
// 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
```

- The content URI of the words table is: `content://user_dictionary/words`
- Read permission for accessing the content provider is also needed in the manifest file:

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

Developing a Custom Content Provider

1. Extend the `ContentProvider` class.
2. In the `onCreate()` method, create a new instance of the database helper class.

```
public class MyContentProvider extends ContentProvider {
    private DbHelper db;

    @Override
    public boolean onCreate() {
        db = new DbHelper(getContext());
        return true;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
        return null;
    }

    @Override
    public String getType(Uri uri) {
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
                      String[] selectionArgs) {
        return 0;
    }
}
```

Developing a Custom Content Provider...

Suppose, we need to provide access to 2 tables through this single content provider. As we have only one method per CRUD operation, we need a way to differentiate between accesses to these two tables.

3. We need to define content URI paths to each table. These are defined in a public final class which can be used by both provider and user as a contract: (see next slide)

Developing a Custom Content Provider...

```
import android.net.Uri;

public final class MyContract {
    private static final String SCHEME = "content://";
    public static final String AUTHORITY = "com.example.contentprovidertest.provider";

    public static final class Table1 {
        public static final String TABLE_NAME = "table1";
        private static final String PATH = "/" + TABLE_NAME;
        public static final Uri CONTENT_URI = Uri.parse(SCHEME + AUTHORITY + PATH);

        private Table1() {}
    }

    public static final class Table2 {
        public static final String TABLE_NAME = "table2";
        private static final String PATH = "/" + TABLE_NAME;
        public static final Uri CONTENT_URI = Uri.parse(SCHEME + AUTHORITY + PATH);

        private Table2() {}
    }

    private MyContract() {}
}
```

Developing a Custom Content Provider...

Now comes the issue of differentiating between paths. The idea is to match a URI and then taking appropriate actions for the corresponding table path.

4. Add a `UriMatcher` to the provider and add expected URI patterns to it.
5. In the `query()` method, get the appropriate table name from the URI.

Developing a Custom Content Provider...

```
public class MyContentProvider extends ContentProvider {
    private DbHelper db;
    private static final UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
    private static final int TABLE1 = 1;
    private static final int TABLE2 = 2;
    private static final int TABLE2_SINGLE_ROW = 3;

    static {
        matcher.addURI(MyContract.AUTHORITY, MyContract.Table1.TABLE_NAME, TABLE1);
        matcher.addURI(MyContract.AUTHORITY, MyContract.Table2.TABLE_NAME, TABLE2);
        matcher.addURI(MyContract.AUTHORITY, MyContract.Table2.TABLE_NAME + "/#", TABLE2_SINGLE_ROW);
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
        String table = "";

        switch (matcher.match(uri)) {
            case TABLE1:
                table = TrafficSignsContract.SignSet.TABLE_NAME;
                break;
            case TABLE2:
                table = TrafficSignsContract.Sign.TABLE_NAME;
                break;
            case TABLE2_SINGLE_ROW:
                table = TrafficSignsContract.Sign.TABLE_NAME;
                String idSelection = TrafficSignsContract.Sign._ID + " = " + uri.getLastPathSegment();
                selection = selection == null ? idSelection : selection + " AND " + idSelection;
                break;
            default:
                throw new IllegalArgumentException("Wrong URI: " + uri.toString());
        }

        return null;
    }
```

Developing a Custom Content Provider...

6. Now write the actual query method:

```
try {
    Cursor cursor = db.getReadableDatabase().query(table, projection, selection,
            selectionArgs, null, null, sortOrder);
    cursor.setNotificationUri(getContext().getContentResolver(), uri);
    return cursor;
} catch (SQLiteException e) {
    Log.e(getClass().getCanonicalName(), "Query Exception", e);
    return null;
}
```

- You should add this URI to notification observables by calling `setNotificationUri()` so that if this cursor is directly used in a ListView, updating or inserting or deleting data in the table represented by this URI would notify the ListView of this data change.

Developing a Custom Content Provider...

7. insert, update and delete methods are similar.
   - `insert()` returns the Uri with the newly inserted ID appended.
   - `update()` and `delete()` returns the number of rows affected.
   - You should call `notifyChangeToContentObservers(uri);` before returning from these methods.

Developing a Custom Content Provider...

We need to provide MIME type of the data returned by a URI.

8. The overridden method `getType(Uri uri)` needs to be filled-in.
   - For common types of data such as as text, HTML, or JPEG, getType() should return the standard MIME type for that data.
   - For content URIs that point to a row or rows of table data, getType() 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.

Developing a Custom Content Provider...

- Content type defined in the contract class:

```
public final class MyContract {
    private static final String SCHEME = "content://";
    public static final String AUTHORITY = "com.example.contentprovidertest.provider";

    public static final class Table1 {
        public static final String TABLE_NAME = "table1";
        private static final String PATH = "/" + TABLE_NAME;
        public static final Uri CONTENT_URI = Uri.parse(SCHEME + AUTHORITY + PATH);

        public static final String CONTENT_TYPE =
                "vnd.android.cursor.dir/vnd." + AUTHORITY + "." + TABLE_NAME;

        private Table1() {}
    }

    public static final class Table2 {
        public static final String TABLE_NAME = "table2";
        private static final String PATH = "/" + TABLE_NAME;
        public static final Uri CONTENT_URI = Uri.parse(SCHEME + AUTHORITY + PATH);

        public static final String CONTENT_TYPE =
                "vnd.android.cursor.dir/vnd." + AUTHORITY + "." + TABLE_NAME;
        public static final String CONTENT_TYPE_SINGLE_ROW =
                "vnd.android.cursor.item/vnd." + AUTHORITY + "." + TABLE_NAME;

        private Table2() {}
    }

    private MyContract() {}
}
```

Developing a Custom Content Provider...

- `getType()` method in the provider class:

```
@Override
public String getType(Uri uri) {
    switch (matcher.match(uri)) {
        case TABLE1:
            return MyContract.Table1.CONTENT_TYPE;
        case TABLE2:
            return MyContract.Table2.CONTENT_TYPE;
        case TABLE2_SINGLE_ROW:
            return MyContract.Table2.CONTENT_TYPE_SINGLE_ROW;
        default:
            return null;
    }
}
```

Developing a Custom Content Provider...

9. We need to declare the provider in the manifest.xml file:

```
<provider android:name="com.example.contentprovidertest.provider.MyContentProvider"
          android:authorities="com.example.contentprovidertest.provider"/>
```

Developing a Custom Content Provider...

10. Finally, we need to define permissions for applications who wish to access the provider.

Different forms of permissions:

- 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 in manifest.xml.
- Separate read and write provider-level permission
  - A read permission and a write permission for the entire provider.
  - Specified 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.
- 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.

Developing a Custom Content Provider...

- Permission defined in manifest.xml of the *provider*:

```
<provider
    android:name="com.example.contentprovidertest.provider.MyContentProvider"
    android:authorities="com.example.contentprovidertest.provider"
    android:permission="com.example.contentprovidertest.provider.permission.READ_WRITE_PROVIDER"/>
```

- Permission defined in manifest.xml of the *user*:

```
<uses-permission
    android:name="com.example.contentprovidertest.provider.permission.READ_WRITE_PROVIDER"/>
```