Introduction to ContentProviders

Consume and publish data in Mono for Android

Xamarin Inc.

BRIEF

A *ContentProvider* encapsulates a data repository and provides an API to access it. The provider exists as part of an Android application that usually also provides a UI for displaying/managing the data. The key benefit of using a content provider is enabling other applications to easily access the encapsulated data using a provider client object (called a *ContentResolver*). Together a content provider and content resolver offer a consistent inter-application API for data access that is simple to build and consume.

The Android operating system uses content providers to facilitate access to shared data such as media files, contacts and calendar information. Any application can choose to use ContentProviders to manage data internally and also to expose it to other applications. A ContentProvider is also required for your application to provide custom search suggestions or if you want to provide the ability to copy complex data from your application to paste into other applications.

This document shows how to access and build ContentProviders with Mono for Android.

Sample Code:

[ContactsAdapterDemo.zip]

[SimpleContentProvider.zip]

[SearchableDictionary.zip]

Related Articles:

Related Android Documentation:

Content Providers Developers Guide

ContentProvider Class Reference

ContentResolver Class Reference

ListView Class Reference

CursorAdapter Class Reference

UriMatcher Class Reference

android.provider Package Summary

ContactsContract Class Reference

Other Related Documentation:

[ListViews and Adapters Document]

Overview

This article introduces the `ContentProvider` class. The discussion will begin with an overview of the `ContentProvider` class itself before introducing two examples of how to use it. The document structure is as follows:

- **How it works** – An overview of what the ContentProvider is designed for and how it works.
- **Consuming a Content Provider** – An example accessing the Contacts list.
- **Using ContentProvider to share data** – Writing and consuming a ContentProvider in the same application.

`ContentProviders` and the cursors that operate on their data are often used to populate ListViews. Refer to the [ListViews and Adapters document] for more information on how to use those classes.

How It Works

There are two classes involved in a ContentProvider interaction:

- **ContentProvider** – Implements an API that exposes a set of data in a standard way. The main methods are Query, Insert, Update and Delete.
- **ContentResolver** – A static proxy that communicates with a ContentProvider to access its data, either from within the same application or from another application.

A content provider is normally backed by an SQLite database, but the API means that consuming code does not need to know anything about the underlying SQL. Queries are done via a Uri using constants to reference column names (to reduce dependencies on the underlying data structure), and an ICursor is returned for the consuming code to iterate over.

Consuming a ContentProvider

ContentProviders expose their functionality through a Uri that is registered in the `AndroidManifest.xml` of the application that publishes the data. There is a convention where the Uri and the data columns that are exposed should be available as constants to make it easy to bind to the data. Android's built-in ContentProviders all provide convenience classes with constants that reference the data structure in the Android.Providers namespace.

BUILT-IN PROVIDERS

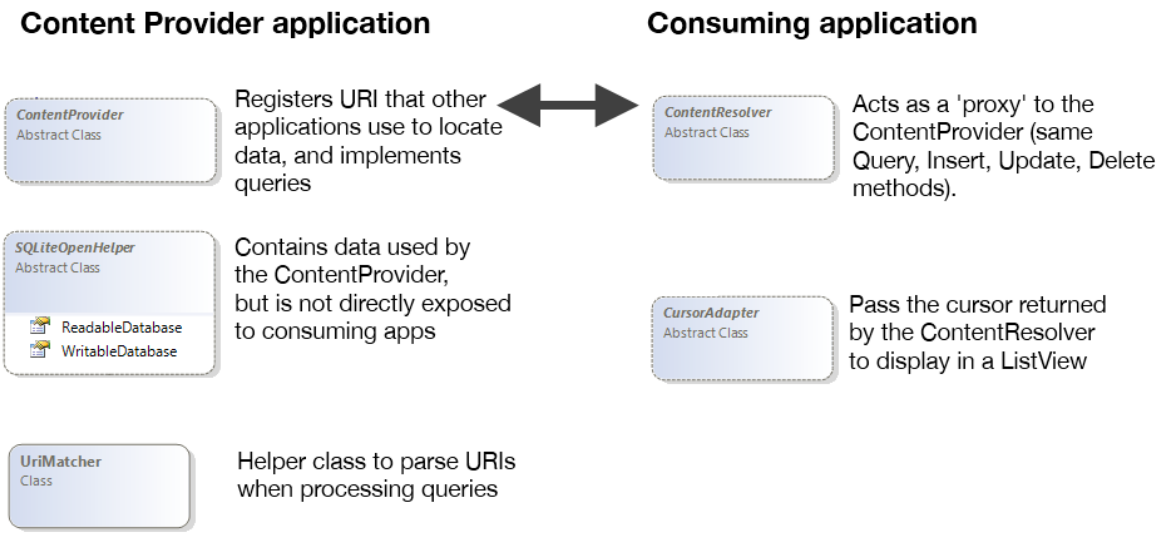
Android offers access to a wide range of system and user data using ContentProviders.

- Browser – bookmarks and browser history (requires permission READ_HISTORY_BOOKMARKS and/or WRITE_HISTORY_BOOKMARKS).

- ➔ CallLog – recent calls made or received with the device.
- ➔ Contacts – detailed information from the user's contact list, including people, phones, photos & groups.
- ➔ MediaStore – contents of the user's device: audio (albums, artists, genres, playlists), images (including thumbnails) & video.
- ➔ Settings – system-wide device settings and preferences.
- ➔ UserDictionary – contents of the user-defined dictionary used for predictive text input.
- ➔ Voicemail – history of voicemail messages.

Classes Overview

The primary classes used when working with a `ContentProvider` are shown here:



The purpose of each class is described below:

- ➔ **ContentProvider** – Implement this abstract class's methods to expose data. The API is made available to other classes and applications via the Uri attribute that is added to the class definition.
- ➔ **SQLiteOpenHelper** – Helps implement the SQLite datastore that is exposed by the `ContentProvider`.
- ➔ **UriMatcher** – Use UriMatcher in your `ContentProvider` implementation to help manage Uris that are used to query the content.
- ➔ **ContentResolver** – Consuming code uses a ContentResolver to access a ContentProvider instance. The two classes together take care of the inter-process communication issues, allowing data to be easily shared between applications. Consuming code never creates a ContentProvider class explicity, instead the data is accessed by creating a cursor based on a Uri exposed by the ContentProvider application.

➔ **CursorAdapter** – Use `CursorAdapter` or `SimpleCursorAdapter` to display data accessed via a `ContentProvider`.

The ContentProvider API allows consumers to perform a variety of operations on the data, such as:

➔ Querying data to return lists or individual records.

➔ Modifying individual records.

➔ Adding new records.

➔ Deleting records.

This document contains an example that use a system-provided ContentProvider as well as a simple read-only example that implements a custom ContentProvider.

Using the Contacts ContentProvider

Writing code to use access data exposed by a ContentProvider doesn't require a reference to the ContentProvider class at all. Instead a Uri is used to create a cursor over the data exposed by the ContentProvider. Android uses the Uri to search the system for the application that has exposed a ContentProvider with that identifier. The Uri is a string, typically in a reverse-DNS format such as this `content://com.android.contacts/data`.

Rather than having to remember this string, the Android *Contacts* provider exposes its metadata in the `android.provider.ContactsContract` class. This class is used to determine both the Uri of the ContentProvider and also the names of the tables and columns that can be queried.

Some data types also require special permission to access. The built-in contacts list requires the `android.permission.READ_CONTACTS` permission in the AndroidManifest.xml file.

There are three ways to create a cursor from the Uri:

➔ **ManagedQuery()** – The preferred approach in Android 2.3 (API Level 10) and earlier, a ManagedQuery returns a cursor and also automatically manages refreshing the data and closing the cursor. This method is deprecated in Android 3.0 (API Level 11).

➔ **ContentResolver.Query()** – Returns an unmanaged cursor, which means it must be refreshed and closed explicitly in code.

➔ **CursorLoader().LoadInBackground()** – Introduced in Android 3.0 (API Level 11), CursorLoader is now the preferred way to consume a ContentProvider. CursorLoader queries a ContentResolver on a background thread so the UI isn't blocked. This class can be accessed in older versions of Android using the v4 compatibility library.

Each of these methods has the same basic set of inputs:

➔ **Uri** – The fully qualified name of the ContentProvider.

➔ **Projection** – Specification of which columns to select for the cursor.

- ➔ **Selection** – Similar to a SQL WHERE clause.
- ➔ **SelectionArgs** – Parameters to be substituted in the Selection.
- ➔ **SortOrder** – Columns to sort by.

Creating Inputs for a query

The ContactsProvider sample code performs a very simple query against Android's built-in Contacts provider. You do not need to know the actual Uri or column names, all the information required to query the Contacts ContentProvider is available as constants exposed by the `ContactsContract` class.

Regardless of which method is used to retrieve the cursor, these same objects are used as parameters as shown in the `ContactsProvider/ContactsAdapter.cs` file:

```
var uri = ContactsContract.Contacts.ContentUri;
string[] projection = {
    ContactsContract.Contacts.InterfaceConsts.Id,
    ContactsContract.Contacts.InterfaceConsts.DisplayName,
    ContactsContract.Contacts.InterfaceConsts.PhotoId,
};
```

For this example, the `selection`, `selectionArgs` and `sortOrder` will be ignored by setting them to `null`.

Creating a Cursor from a Content Provider Uri

Once the parameter objects have been created, they can be used in one of the following three ways:

USING A MANAGED QUERY

Applications targeting Android 2.3 (API Level 10) or earlier should use this method:

```
var cursor = activity.ManagedQuery(uri, projection, null, null, null);
```

This cursor will be managed by Android so you do not need to close it.

USING CONTENTRESOLVER

Accessing ContentResolver directly to get a cursor against a ContentProvider can be done like this:

```
var cursor = activity.ContentResolver(uri, projection, null, null, null);
```

This cursor is unmanaged, so it must be closed when no longer required. Ensure that the code closes a cursor that is open, otherwise an error will occur.

```
cursor.Close();
```

Alternatively you can call `StartManagingCursor()` and `StopManagingCursor()` to 'manage' the cursor. Managed cursors are automatically deactivated and re-queried when activities are stopped and restarted.

USING CURSORLOADER

Applications built for Android 3.0 (API Level 11) or newer should use this method:

```
var loader = new CursorLoader (activity, uri, projection, null, null,
null);
var cursor = (ICursor)loader.LoadInBackground();
```

The `CursorLoader` ensures that all cursor operations are done on a background thread, and can intelligently re-use an existing cursor across activity instances when an activity is restarted (eg. due to a configuration change) rather that reload the data again.

Earlier Android versions can also use the CursorLoader class by using the v4 support libraries [doc ref?].

Displaying the Cursor Data with a Custom Adapter

To display the contact image we'll use a custom adapter, so that we can manually resolve the PhotoId reference to an image file path.

To display data with a custom adapter, the example uses a CursorLoader to retrieve all the Contact data into a local collection in the `FillContacts` method from `ContactsProvider/ContactsAdapter.cs`:

```
void FillContacts ()
{
    var uri = ContactsContract.Contacts.ContentUri;
    string[] projection = {
        ContactsContract.Contacts.InterfaceConsts.Id,
        ContactsContract.Contacts.InterfaceConsts.DisplayName,
        ContactsContract.Contacts.InterfaceConsts.PhotoId
    };
    // CursorLoader introduced in Honeycomb (3.0, API11)
    var loader = new CursorLoader(activity, uri, projection, null, null,
null);
    var cursor = (ICursor)loader.LoadInBackground();
    contactList = new List<Contact> ();
    if (cursor.MoveToFirst ()) {
        do {
            contactList.Add (new Contact{
                Id = cursor.GetLong (cursor.GetColumnIndex (projection
[0])),
                DisplayName = cursor.GetString (cursor.GetColumnIndex
(projection [1])),
                PhotoId = cursor.GetString (cursor.GetColumnIndex
(projection [2]))
            });
        } while (cursor.MoveToNext());
    }
}
```

Then implement the BaseAdapter's methods using the `contactList` collection. The adapter is implemented just as it would be with any other collection – there is no 'special handling' here because the data is sourced from a ContentProvider:

```
Activity activity;
public ContactsAdapter (Activity activity)
{
    this.activity = activity;
    FillContacts ();
```

```
}
public override int Count {
    get { return contactList.Count; }
}
public override Java.Lang.Object GetItem (int position)
{
   return null; // could wrap a Contact in a Java.Lang.Object to return it
here if needed
}
public override long GetItemId (int position)
{
    return contactList [position].Id;
}
public override View GetView (int position, View convertView, ViewGroup
parent)
{
    var view = convertView ?? activity.LayoutInflater.Inflate
(Resource.Layout.ContactListItem, parent, false);
    var contactName = view.FindViewById<TextView>
(Resource.Id.ContactName);
    var contactImage = view.FindViewById<ImageView>
(Resource.Id.ContactImage);
    contactName.Text = contactList [position].DisplayName;
    if (contactList [position].PhotoId == null) {
        contactImage = view.FindViewById<ImageView>
(Resource.Id.ContactImage);
        contactImage.SetImageResource (Resource.Drawable.ContactImage);
    } else {
        var contactUri = ContentUris.WithAppendedId
(ContactsContract.Contacts.ContentUri, contactList [position].Id);
        var contactPhotoUri = Android.Net.Uri.WithAppendedPath (contactUri,
Contacts.Photos.ContentDirectory);
        contactImage.SetImageURI (contactPhotoUri);
    }
    return view;
}
```

The image is displayed (if it exists) using the Uri to the image file on the device. The application looks like this:

Using a similar code pattern your application can access a wide variety of system data including the user's photos, videos and music. Some data types require special permissions to be requested in the project's Properties (which are stored in AndroidManifest.xml).

Displaying the Cursor Data with a SimpleCursorAdapter

The cursor could also be displayed with a `SimpleCursorAdapter` (although only the name will be displayed, not the photo). This code demonstrates how to use a ContentProvider with SimpleCursorAdapter (this code does not appear in the sample):

```
var uri = ContactsContract.Contacts.ContentUri;
string[] projection = {
    ContactsContract.Contacts.InterfaceConsts.Id,
    ContactsContract.Contacts.InterfaceConsts.DisplayName
};
var loader = new CursorLoader (this, uri, projection, null, null, null);
var cursor = (ICursor)loader.LoadInBackground();
var fromColumns = new string[]
{ContactsContract.Contacts.InterfaceConsts.DisplayName};
var toControlIds = new int[] {Android.Resource.Id.Text1};
adapter = new SimpleCursorAdapter (this,
Android.Resource.Layout.SimpleListItem1, cursor, fromColumns,
toControlsIds);
listView.Adapter = adapter;
```

Refer to the [ListViews and Adapters document] for further information on implementing `SimpleCursorAdapter`.

Creating a Custom ContentProvider

The previous section demonstrated how to consume data from a built-in ContentProvider implementation. This section will explain how to build a custom ContentProvider and then consume its data.

About ContentProviders

A content provider class must inherit from `ContentProvider`. It should consist of an internal data store that is used to respond to queries and it should expose Uris and MIME Types as constants to help consuming code make valid requests for data.

URI (AUTHORITY)

ContentProviders are accessed in Android using a Uri. An application that exposes a ContentProvider sets the Uris that it will respond to in its AndroidManifest.xml file. When the application is installed, these Uris are registered so that other applications can access them.

In Mono for Android, the content provider class should have a `[ContentProvider]` attribute to specify the Uri (or Uris) that should be added to AndroidManifest.xml.

MIME TYPE

The typical format for MIME Types consists of two parts. Android ContentProviders commonly use these two strings for the first part of the MIME Type:

- ➔ "vnd.android.cursor.item" – to represent a single row, use the `ContentResolver.CursorItemBaseType` constant in code.
- ➔ "vnd.android.cursor.dir" – for multiple rows, use the `ContentResolver.CursorDirBaseType` constant in code.

The second part of the MIME Type is specific to your application, and should use a reverse-DNS standard with a '`vnd`.' prefix. The sample code uses "`vnd.com.xamarin.sample.Vegetables`".

DATA MODEL METADATA

Consuming applications need to construct Uri 'queries' to access different types of data. The base Uri can be expanded to refer to a particular table of data and may also include parameters to filter the results. Further, the columns and clauses used with the resulting cursor to display data must be declared.

To ensure that only valid Uri queries are constructed it is customary to provide the valid strings as constant values. This makes it easier to access the ContentProvider because it makes the values 'discoverable' via code-completion and prevents typos in the strings.

In the previous example the `android.provider.ContactsContract` class exposed the metadata for the Contacts data. For our custom ContentProvider we will just expose the constants on the class itself.

Implementation

There are three steps to creating and consuming a custom `ContentProvider`:

- ➔ **Create a database class** – Implement `SQLiteOpenHelper`.
- ➔ **Create a ContentProvider class** – Implement `ContentProvider` with an instance of the database, metadata exposed as constant values and methods to access the data.
- ➔ **Access the ContentProvider via its Uri** – Populate a CursorAdapter using the ContentProvider, accessed via its Uri.

As previously discussed, ContentProviders can be consumed from applications other than where they are defined. In this example the data is consumed in the same application, but keep in mind that other applications can also access it (as long as they know the Uri and information about the schema, which is usually exposed as constant values).

Create a database

Most ContentProvider implementations will be based on a SQLite database. The example database code in SimpleContentProvider/VegetableDatabase.cs creates a very simple two-column database, as shown:

```
class VegetableDatabase  : SQLiteOpenHelper {
    public static readonly string create_table_sql =
        "CREATE TABLE [vegetables] ([_id] INTEGER PRIMARY KEY
AUTOINCREMENT NOT NULL UNIQUE, [name] TEXT NOT NULL UNIQUE)";
    public static readonly string DatabaseName = "vegetables.db";
    public static readonly int DatabaseVersion = 1;
    public VegetableDatabase(Context context) : base(context, DatabaseName,
null, DatabaseVersion) { }
    public override void OnCreate(SQLiteDatabase db)
    {
        db.ExecSQL(create_table_sql);
        // seed with data
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Vegetables')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Fruits')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Flower
Buds')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Legumes')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Bulbs')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Tubers')");
    }
    public override void OnUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion)
    {   // not required for this example
        throw new NotImplementedException();
    }
}
```

The database implementation itself does not need any special considerations to be exposed with a ContentProvider, however if the ContentProvider's data is intended to be bound to a `ListView` control then a unique integer column named

`_id` must be part of the result set. See the [ListViews and Adapters document] for more details on using the `ListView` control.

Create the ContentProvider

The rest of this section gives step-by-step instructions on how the `SimpleContentProvider/VegetableProvider.cs` example class was built.

INITIALIZE THE DATABASE

The first step is to subclass `ContentProvider` and add the database that it will use.

```
public class VegetableProvider : ContentProvider {
    VegetableDatabase vegeDB;
    public override bool OnCreate()
    {
        vegeDB = new VegetableDatabase(Context);
        return true;
    }
}
```

The rest of the code will form the actual content provider implementation that allows the data to be discovered and queried.

ADD METADATA FOR CONSUMERS

There are four different types of metadata that we are going to expose on the ContentProvider class. Only the authority is required, the rest are done by convention.

- ➔ **Authority** – The ContentProvider attribute *must* be added to the class so that it is registered with the Android when the application is installed.
- ➔ **Uri** – The CONTENT_URI is exposed as a constant so that it is easy to use in code. It should match the Authority, but include the scheme and base path.
- ➔ **MIME Types** – Lists of results and single results are treated as different content types, so we define two MIME Types to represent them.
- ➔ **InterfaceConsts** – Provide a constant value for each data column name, so that consuming code can easily discover and refer to them without risking typographical errors.

This code shows how each of these items is implemented, adding to the database definition from the previous step:

```
[ContentProvider(new string[] { "com.xamarin.sample.VegetableProvider" })]
public class VegetableProvider : ContentProvider {
    public static readonly String AUTHORITY =
"com.xamarin.sample.VegetableProvider";
    static string BASE_PATH = "vegetables";
    public static readonly Android.Net.Uri CONTENT_URI =
Android.Net.Uri.Parse("content://" + AUTHORITY + "/" + BASE_PATH);
    // MIME types used for getting a list, or a single vegetable
    public static readonly String VEGETABLES_MIME_TYPE =
ContentResolver.CursorDirBaseType + "/vnd.com.xamarin.sample.Vegetables";
```

```
    public static readonly String VEGETABLE_MIME_TYPE =
ContentResolver.CursorItemBaseType + "/vnd.com.xamarin.sample.Vegetables";
    // Column names
    public static class InterfaceConsts {
        public static readonly string Id = "_id";
        public static readonly string Name = "name";
    }
    VegetableDatabase vegeDB;
    public override bool OnCreate()
    {
        vegeDB = new VegetableDatabase(Context);
        return true;
    }
}
```

IMPLEMENT URI PARSING HELPER

Because consuming code uses Uris to make requests of a ContentProvider, we need to be able to parse those requests to determine what data to return. The `UriMatcher` class can help to parse Uris, once it has been initialized with the Uri patterns that the ContentProvider supports. When adding Uris to the UriMatcher

The UriMatcher in the example will be initialized with two Uris:

- ➔ "com.xamarin.sample.VegetableProvider/vegetables" – request to return the full list of vegetables.
- ➔ "com.xamarin.sample.VegetableProvider/vegetables/#" – where the # is a placeholder for a numeric parameter (the _id of the row in the database). An asterisk placeholder ("*") can also be used to match a text parameter.

In the code we use the constants to refer to metadata values like the AUTHORITY and BASE_PATH. The return codes will be used in methods that do Uri parsing, to determine what data to return.

```
const int GET_ALL = 0; // return code when list of Vegetables requested
const int GET_ONE = 1; // return code when a single Vegetable is requested
by ID
static UriMatcher uriMatcher = BuildUriMatcher();
static UriMatcher BuildUriMatcher()
{
    var matcher = new UriMatcher(UriMatcher.NoMatch);
    // Uris to match, and the code to return when matched
    matcher.AddURI(AUTHORITY, BASE_PATH, GET_ALL); // all vegetables
    matcher.AddURI(AUTHORITY, BASE_PATH + "/#", GET_ONE); // specific
vegetable by numeric ID
    return matcher;
}
```

This code is all private to the ContentProvider class. Refer to Google's UriMatcher documentation for further information.

IMPLEMENT THE QUERY METHOD

The simplest ContentProvider method to implement is the `Query` method. The implementation below uses the `UriMatcher` to parse the `uri` parameter and call

the correct database method. If the `uri` contains an ID parameter then the integer is parsed out (using `LastPathSegment`) and used in the database query.

```
public override Android.Database.ICursor Query(Android.Net.Uri uri,
string[] projection, string selection, string[] selectionArgs, string
sortOrder)
{
    switch (uriMatcher.Match(uri)) {
    case GET_ALL:
        return GetFromDatabase();
    case GET_ONE:
        var id = uri.LastPathSegment;
        return GetFromDatabase(id); // the ID is the last part of the Uri
    default:
        throw new Java.Lang.IllegalArgumentException("Unknown Uri: " +
uri);
    }
}
Android.Database.ICursor GetFromDatabase()
{
    return vegeDB.ReadableDatabase.RawQuery("SELECT _id, name FROM
vegetables", null);
}
Android.Database.ICursor GetFromDatabase(string id)
{
    return vegeDB.ReadableDatabase.RawQuery("SELECT _id, name FROM
vegetables WHERE _id = " + id, null);
}
```

The GetType method must also be overridden. This method may be called to determine the content type that will be returned for a given Uri. This might tell the consuming application how to handle that data.

```
public override String GetType(Android.Net.Uri uri)
{
    switch (uriMatcher.Match(uri)) {
    case GET_ALL:
        return VEGETABLES_MIME_TYPE; // list
    case GET_ONE:
        return VEGETABLE_MIME_TYPE; // single item
    default:
        throw new Java.Lang.IllegalArgumentExceptoin ("Unknown Uri: " +
uri);
    }
}
```

IMPLEMENT THE OTHER OVERRIDES

Our simple example does not allow for editing or deletion of data, but the Insert, Update and Delete methods must be implemented so add them without an implementation:

```
public override int Delete(Android.Net.Uri uri, string selection, string[]
selectionArgs)
{
    throw new Java.Lang.UnsupportedOperationException();
}
```

```
public override Android.Net.Uri Insert(Android.Net.Uri uri, ContentValues
values)
{
    throw new Java.Lang.UnsupportedOperationException();
}
public override int Update(Android.Net.Uri uri, ContentValues values,
string selection, string[] selectionArgs)
{
    throw new Java.Lang.UnsupportedOperationException();
}
```

That completes the basic ContentProvider implementation. Once the application has been installed, the data it exposes will be available both inside the application but also to any other application that knows the Uri to reference it.

Access the ContentProvider

Once the `VegetableProvider` has been implemented, accessing it is done the same way as the Contacts provider at the start of this document: obtain a cursor using the specified Uri and then use an adapter to access the data.

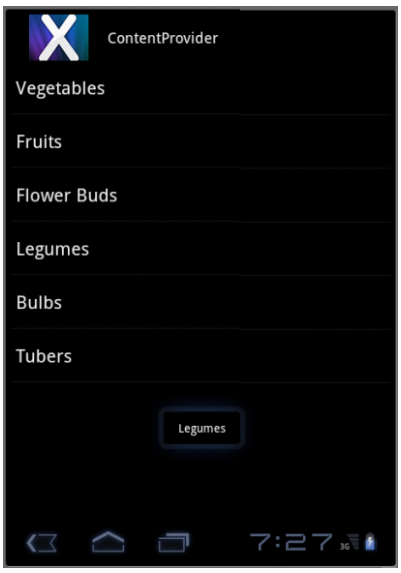
BIND A LISTVIEW TO THE CONTENTPROVIDER

To populate a ListView with data we use the Uri that corresponds to the unfiltered list of vegetables. In the code we use the constant value `VegetableProvider.CONTENT_URI`, which we know resolves to "`content://com.xamarin.sample.VegetableProvider/vegetables`". Our `VegetableProvider.Query` implementation will return a cursor that can then be bound to the ListView.

The code in `SimpleContentProvider/HomeScreen.cs` shows how simple it is to display data from a `ContentProvider`:

```
listView = FindViewById<ListView>(Resource.Id.List);
string[] projection = new string[] { VegetableProvider.InterfaceConsts.Id,
VegetableProvider.InterfaceConsts.Name} ;
string[] fromColumns = new string[]
{ VegetableProvider.InterfaceConsts.Name };
int[] toControlIds = new int[] { Android.Resource.Id.Text1 };
// CursorLoader introduced in Honeycomb (3.0, API11)
var loader = new CursorLoader(this,
    VegetableProvider.CONTENT_URI, projection, null, null, null);
cursor = (ICursor)loader.LoadInBackground();
// create a SimpleCursorAdapter
adapter = new SimpleCursorAdapter(this,
Android.Resource.Layout.SimpleListItem1, cursor, fromColumns,
toControlIds);
listView.Adapter = adapter;
```

The resulting application looks like this:



RETRIEVE A SINGLE ITEM FROM THE CONTENT PROVIDER

A consuming application might also want to access single rows of data, which can be done by constructing a different Uri that refers to a specific row (for example).

Use ContentResolver directly to access a single item, by building up a Uri with the required Id.

```
Uri.WithAppendedPath(VegetableProvider.CONTENT_URI, id.ToString());
```

The complete method looks like this:

```
protected void OnListItemClick(object sender,
AdapterView.ItemClickEventArgs e)
{
    var id = e.Id;
    string[] projection = new string[] { "name" };
    var uri = Uri.WithAppendedPath(VegetableProvider.CONTENT_URI,
id.ToString());
    ICursor vegeCursor = ContentResolver.Query(uri, projection, null, new
string[] { id.ToString() }, null);
    string text = "";
    if (vegeCursor.MoveToFirst()) {
        text = vegeCursor.GetInt(0) + " " + vegeCursor.GetString(1);
        Android.Widget.Toast.MakeText(this, text,
Android.Widget.ToastLength.Short).Show();
    }
    vegeCursor.Close();
}
```

Summary

ContentProviders that are exposed by Android (or other applications) are an easy way to include data from other sources in your application. They allow you to

access and present data such as the Contacts list, photos or calendar events from within your application and let the user interact with that data.

Custom ContentProviders are a convenient way to package your data for use inside your own app, but also so that other applications can consume it as well (including special uses like custom search and copy/paste).

This document has provided some simple examples of consuming and writing ContentProvider code.