Content Providers

Introduction

- The Android platform allows to expose the data sources (e.g. the SQLite database) through a REST like abstraction, known as a 'Content Provider'.
- In order to retrieve data from a content provider or in order to save data into it we use REST-like URIs.
- The Android platform treats every URI address that starts with `content//` as a URI address that represents a data resource served by a content provider.

Introduction

- The content provider Uri allows us to perform basic CRUD (create, read, update, delete) operations.
- Each Uri instance represents either a collection of data or a specific individual one.

Introduction

- Assuming a database named 'school' on our android we might use the following URI:

```
content://com.abelski.schoolprovider/student
```

- If we want to access a specific student we might use the following URL:

```
content://com.abelski.schoolprovider/student/34234
```

Introduction

- The URI access mechanism simplifies our access to the content providers we use.
- Content providers allow different applications on the android platform to share data with each other.

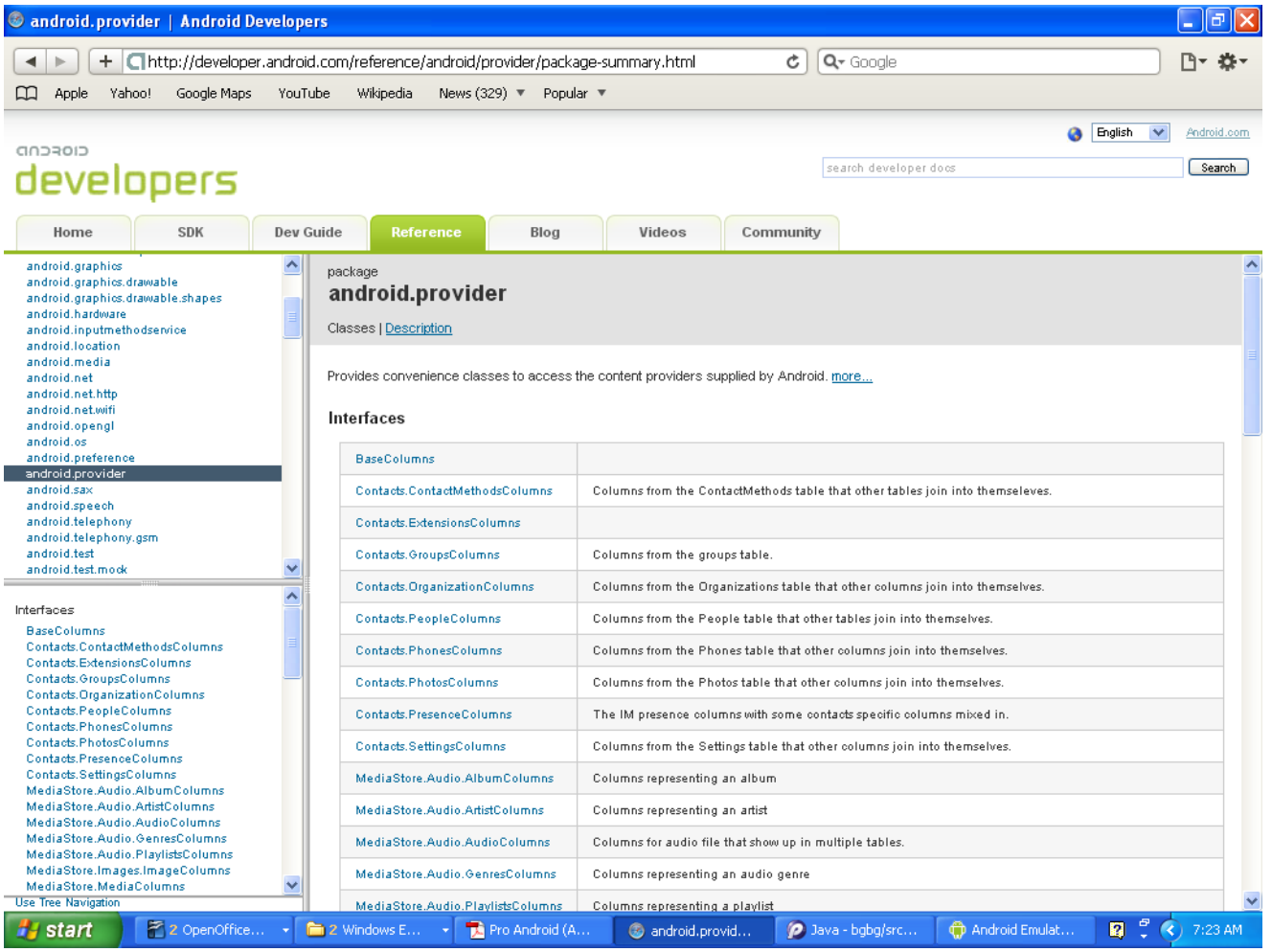
Android Built-In Content Providers

- The android platform comes with a pre defined list of content providers.

  You can find the list of these predefined content providers browsing at

  http://developer.android.com/reference/android/provider/package-summary.html

Android Built-In Content Providers



Content Providers Architecture

- Each content provider exposes specific data through a URI similarly to the way a web site exposes its content through URL and similarly to the way a web service provides us with its service.

Content Providers Registration

- Each content provider is registered (similarly to a web site) with a name (similarly to the domain name concept) and a set of URIs (similarly to the URL concept).

Content Providers Registration

- The `AndroidManifext.xml` file includes the registration of the content providers we make available for others.
- Each content provider is registered using the `<provider>` XML element.

```
<provider android:name="StudentsProvider"
android:authorities="com.zindell.studentsprovider" />
```

Content Providers Registration

- The '`authorities`' attribute in each content provider registration defines the URI through which the content provider will be accessed.
- The URI should be composed of small letters only.
- The previous code sample defines a content provider available for access using the following URI:

```
content://com.zindell.studentsprovider/student/
```

Content Providers REST Access

- The content providers' URL functions in a similar way to a REST-like URL address through which data is retrieved or being manipulated.

  Using `content://com.zindell.studentsprovider/student` URL we can get the details of all students.

  Using `content://com.zindell.studentsprovider/student/2342` we can get a specific student identified by 2342.

- The content provider returns the data as a set of rows and columns.

Content Providers URL Structure

- The content provider URIs has the following structure:

  content:// (1) / (2) / (3)

- The (1) part is known as the authorities. The (2) part is known as the data type part and it might point at a collection or a directory. The (3) part is known as the instance identifier and it points at a specific item.

- The following are few examples for possible URIs:

```
content://com.zindell.provider.booksprovider/book/3234
content://com.abelski.provider.coursesprovider/course/2534
```

Content Providers URL Structure

- The data type path can be empty, if the content provider handles only one type of content, a single path segment (e.g. student) or even a chain of path segments (employees/managers).
- The instance identifier is an integer identifying a specific piece of content. The content Uri without the instance identifier refers the whole collection of the content represented by the Uri.

Content Providers Mime Types

- Just as with a response we get from a web server that includes the MIME type describing the response, a content provider returns a MIME type as well.
- The content provider MIME Type is different when getting a collection of records and when getting a specific record.
- The android platform uses the content provider MIME type as a mean for identifying it.

Content Providers Mime Types

- When the content provider returns a single record the mime type should be of the following pattern:

```
vnd.android.cursor.item/ourcompany.contenttype
```

- The following are few possible examples:

```
vnd.android.cursor.item/abelski.course
vnd.android.cursor.item/abelski.student
vnd.android.cursor.item/abelski.topic
```

Content Providers Mime Types

- When the content provider returns a collection of records the mime type should be of the following pattern:

```
vnd.android.cursor.dir/ourcompany.contenttype
```

- The following are few possible examples:

```
vnd.android.cursor.dir/abelski.course
vnd.android.cursor.dir/abelski.student
vnd.android.cursor.dir/abelski.topic
```

Using The Content Provider

- Using a content provider `URIs` it is possible to execute the various state change methods, such as insert, update and delete.

The `Cursor` Object

- When querying a content provider asking for a collection of records we work with a `Cursor` object.

The `Cursor` Object

```
...

String[] vec = new String[] { Contacts.People.NAME};
contacts = Contacts.People.CONTENT_URI;

cursor = managedQuery(
    contacts,
    vec, // specifying which columns return
    null,// specifying which rows to return... in this case all rows
    null,// selection arguments... in this case none
    Contacts.People.NAME + " ASC"); // ascending order by name

int i = 0;
if (cursor.moveToFirst()) {
    String name = null;
    int nameColumn = cursor.getColumnIndex(Contacts.People.NAME);
    do {
        name = cursor.getString(nameColumn);
        builder.append("i="+i+" "+name+"\n");
        i++;
    }
    while (cursor.moveToNext());
}
...
```

The `ContentValues` Object

- When executing methods such as insert,delete and update we work with a `ContentValues` object.

The `ContentValues` Object

```
...
values = new ContentValues();
values.put(Contacts.People.NAME, "Haim Michael");
uri = getContentResolver().insert(Contacts.People.CONTENT_URI, values);
values = new ContentValues();
values.put(Contacts.People.NAME, "Moshe Israeli");
uri = getContentResolver().insert(Contacts.People.CONTENT_URI, values);
...
```

Address Book Content Provider Demo

- The android platform already includes several predefined content providers we can access from within our code.
- One of them is a content provider through which we can interact with the mobile telephone address book. Its URI address is defined at `Contacts.People.CONTENT_URI`.

Address Book Content Provider Demo

```
package com.abelski;

import android.app.Activity;
import android.net.Uri;
import android.os.Bundle;
import android.content.ContentValues;
import android.provider.Contacts;
import android.widget.TextView;
import android.database.Cursor;

public class ContactsContentProviderInteraction extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        StringBuilder builder = new StringBuilder();
        try
        {
            // declaring required local variables
            ContentValues values = null;
            Uri uri = null;
            Cursor cursor = null;
            Uri contacts = null;
```

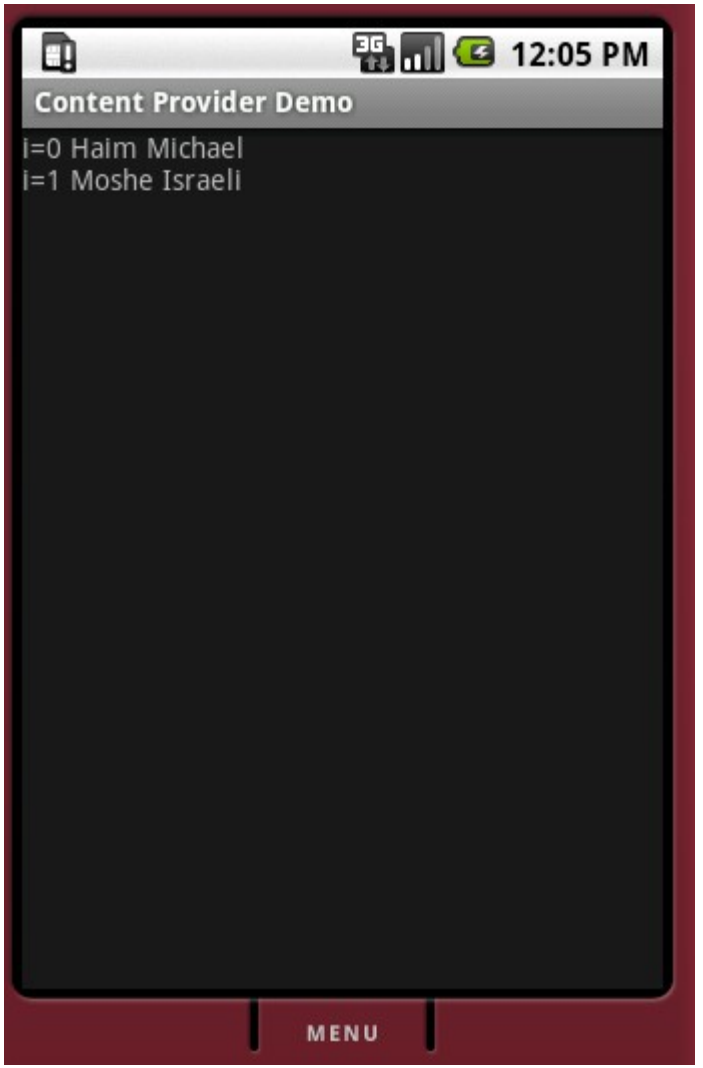
Content Provider Demo

```
// restoring state
super.onCreate(savedInstanceState);
// setting view
setContentView(R.layout.main);

// adding records
values = new ContentValues();
values.put(Contacts.People.NAME, "Haim Michael");
uri = getContentResolver().
    insert(Contacts.People.CONTENT_URI, values);
values = new ContentValues();
values.put(Contacts.People.NAME, "Moshe Israeli");
uri = getContentResolver().insert(
    Contacts.People.CONTENT_URI, values);

// getting all records
String[] vec = new String[] { Contacts.People.NAME};
contacts = Contacts.People.CONTENT_URI;
cursor = managedQuery(
    contacts,
    vec, // specifying return columns
    null,// specifying which rows to return.. this case all rows
    null,// selection arguments... in this case none
    Contacts.People.NAME + " ASC"); // ascending order by name
int i = 0;
```

Address Book Content Provider Demo

```
            if (cursor.moveToFirst())
            {
                String name = null;
                int nameColumn = cursor.getColumnIndex(Contacts.People.NAME);
                do
                {
                    name = cursor.getString(nameColumn);
                    builder.append("i="+i+" "+name+"\n");
                    i++;
                }
                while (cursor.moveToNext());
            }
        }
        catch (Exception e)
        {
            builder.append(e.getMessage());
        }
        TextView text = (TextView)findViewById(R.id.tf);
        text.setText(builder.toString());
    }
}
```

Address Book Content Provider Demo



The `SimpleCursorAdapter` Class

- This class bridges between a `Cursor` object and a selection widget such as the list view.

The `SimpleCursorAdapter` Class

```
public class SimpleContentProviderDemoActivity extends Activity
{
    @Override
    public void onCreate(Bundle bndl)
    {
        super.onCreate(bndl);
        ContentResolver resolver = getContentResolver();
        Cursor cursor =
            resolver.query(Settings.System.CONTENT_URI,
            null, null,null, null);
        startManagingCursor(cursor);
        ListView list = new ListView(this);
        String[] from = new String[]
            { Settings.System.NAME, Settings.System.VALUE };
        int[] to = new int[]
            { R.id.thename, R.id.thevalue };
        SimpleCursorAdapter adapter = new SimpleCursorAdapter(
            this,R.layout.row, cursor, from, to);
        list.setAdapter(adapter);
        setContentView(list);
    }
}
```

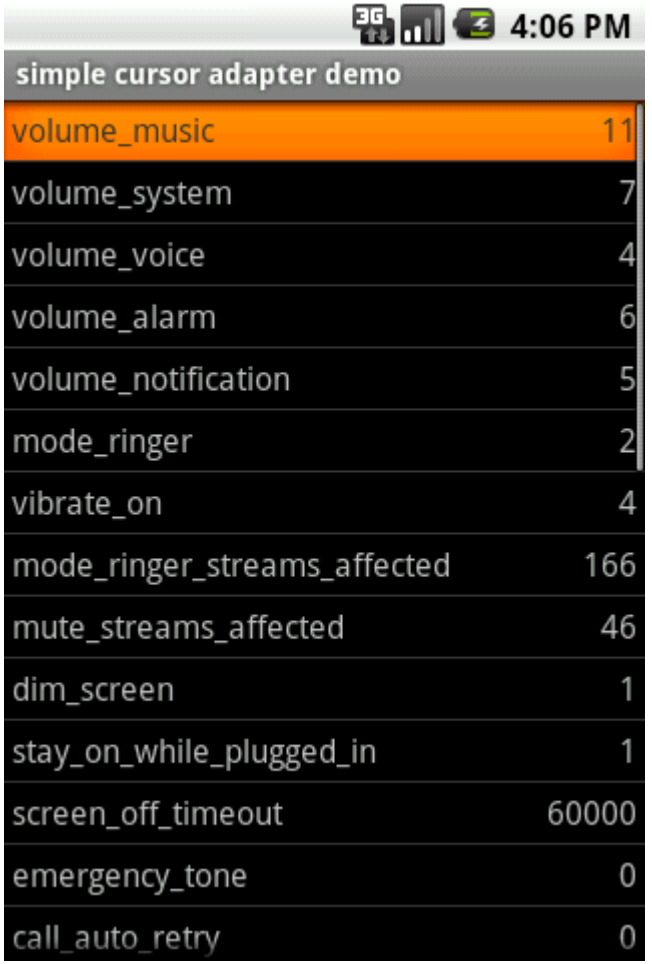
The Activity Source Code

The `SimpleCursorAdapter` Class

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="wrap_content"
    android:padding="5sp"
    android:layout_width="fill_parent">
    <TextView
        android:layout_height="wrap_content"
        android:id="@+id/thename"
        android:textSize="22sp"
        android:layout_width="fill_parent"
        android:layout_weight="1"></TextView>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/thevalue"
        android:textSize="22sp"
        android:gravity="right"></TextView>
</LinearLayout>
```

The row.xml Layout File

The `SimpleCursorAdapter` Class



Create Our Own Content Provider

- The drawback in using a database directly is its access limitation. Once a database is created it is visible for the application that created it only. The SQLite database created on Android by one application is usable by that application only. Other applications cannot use it.
- Creating a content provider will allow our data to be accessible from other applications.

Create Our Own Content Provider

- In order to create a new content provide we should define a class that extends the `ContentProvider` abstract class.
- It is a common practice to include within the definition of the new content provider an inner class through which the content provider will access the database.

Create Our Own Content Provider

- Within the class we define as a class that extends `ContentProvider` we are responsible for implementing six methods: `onCreate()`, `query()`, `insert()`, `update()`, `delete()` and `getType()`.

The `onCreate()` Method

- This method is the entry point to the content provider. Within this method we will initialize our content provider.
- This method should return true in order to indicate that the content provider is ready for use.

```
@Override
public boolean onCreate()
{
    db=(new DatabaseHelper(getContext())).
        GetWritableDatabase();
    return (db == null) ? false : true;
}
```

The `query()` Method

- When the `query()` method is been called by the `ContentResolver` object our content provider gets the details of the query the activity wants to perform.

```
...
public abstract Cursor query(
    Uri  uri,
    String[]  projection,
    String  selection,
    String[]  selectionArgs,
    String  sortOrder)
...
```

The `query()` Method

- The returned `Cursor` will be used to iterate through the returned records.

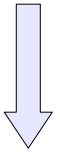
```
...
if (cursor.moveToFirst()) {
    ...
    do {
        ...
    }
    while(cursor.moveToNext());
}
...
```

The `query()` Method

- The query method is indirectly been called when calling the `managedQuery` method. The parameters of these two methods match with each other.

The `query()` Method

```
public final Cursor managedQuery (    Uri  uri,
                                      String[] projection,
                                      String selection,
                                      String[] selectionArgs,
                                      String sortOrder)
{
        ...
}
```



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

The `query()` Method

- The `Uri` parameter represents the collection or the instance being queried.
- The `projection` String[] parameter array represents the list of properties the content provider should return.
- The `selection` String parameter is the where clause of the query.

The `query()` Method

- The `selectionArgs` String[] parameter is an array of values that substitute ? marks the selection string includes.
- The `sortOrder` parameter is the sort clause of the query.

The `insert()` Method

- The `insert()` method receives a `Uri` representing the collection to which we want to add the new values and a `ContentValues` object that includes the initial data of our new instance.

```
...
public Uri  insert (Uri  uri, ContentValues  values)
{
    ...
}
...
```

The `insert()` Method

- Our implementation for the `insert()` method is responsible for creating the new data instance (e.g. new table raw) and returning a `Uri` for the new instance.

The `update()` Method

- The `update()` method gets the `Uri` of the collection or of the instance that needs to be changed.

```
...
public int update ( Uri uri,
                    ContentValues values,
                    String selection,
                    String[] selectionArgs)
{
    ...
}
...
```

The `update()` Method

- The `uri` parameter represents the collection or the instance that we want to change.
- The `values` parameter includes the new values we want to set. This parameter is of the `ContentValues` type.
- The `selection` string holds the where clause and the `selectionArgs` array of strings holds the parameters to use for replacing the ? Characters the where clause includes.

The `update()` Method

- Our implementation for this method should identify the instance or the instances to be modified based on the `uri` and the `selection` where clause and then replace the instance(s) current property values with the ones supplied.

The `delete()` Method

- Our definition for the `delete()` method receives a `uri` that represents the instance or the collection that we need to delete and a `selection` where clause together with the `selectionArgs` parameters.

```
...
public int delete(
    Uri uri, String selection, String[] selectionArgs)
{
    ...
}
...
```

The `getType()` Method

- This method receives a `uri` object and returns its MIME type. The `uri` can represent either a collection or an instance.
- Our implementation for this method should determine which type of `uri` was received and return the corresponding MIME type.

```
...
public String getType(Uri url){
    ...
}
...
```

Create Content Provider Code Sample

```
public class MemoProvider extends ContentProvider
{
    private static final String LOGGER_TAG = "MemoProvider";
    private static final String DATABASE_NAME = "memos.db";
    private static final int DATABASE_VERSION = 2;
    private static final String MEMOS_TABLE_NAME = "memos";
    private static HashMap<String, String> map;
    private static final int MEMOS = 1;
    private static final int SPECIFIC_MEMO = 2;
    private static final UriMatcher matcher;

    static
    {
        matcher = new UriMatcher(UriMatcher.NO_MATCH);
        matcher.addURI(Memos.AUTHORITY, "memos", MEMOS);
        matcher.addURI(Memos.AUTHORITY, "memos/#", SPECIFIC_MEMO);
        map = new HashMap<String, String>();
        map.put(Memos._ID, Memos._ID);
        map.put(Memos.TITLE, Memos.TITLE);
        map.put(Memos.MEMO, Memos.MEMO);
        map.put(Memos.CREATION_DATE, Memos.CREATION_DATE);
        map.put(Memos.MODIFIED_DATE, Memos.MODIFIED_DATE);
    }
```

Create Content Provider Code Sample

```
public static class Memos implements BaseColumns
{
    public static final String AUTHORITY = "com.abelski.provider.memodb";
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/memos");
    public static final String CONTENT_TYPE =
        "vnd.android.cursor.dir/vnd.abelski.memo";
    public static final String CONTENT_ITEM_TYPE =
        "vnd.android.cursor.item/vnd.abelski.memo";
    public static final String DEFAULT_SORT_ORDER = "modified DESC";
    public static final String TITLE = "title";
    public static final String MEMO = "memo";
    public static final String CREATION_DATE = "created";
    public static final String MODIFIED_DATE = "modified";
}
```

Create Content Provider Code Sample

```
private static class DatabaseHelper extends SQLiteOpenHelper
{
    DatabaseHelper(Context context)
    {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db)
    {
        db.execSQL("CREATE TABLE " + MEMOS_TABLE_NAME + " (" + Memos._ID
                + " INTEGER PRIMARY KEY," + Memos.TITLE + " TEXT,"
                + Memos.MEMO + " TEXT," + Memos.CREATION_DATE + " INTEGER,"
                + Memos.MODIFIED_DATE + " INTEGER" + ");");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    {
        Log.w(LOGGER_TAG, "Upgrading the database from version "
                + oldVersion + " to " + newVersion
                + ", old data is deleted");
        db.execSQL("DROP TABLE IF EXISTS " + MEMOS_TABLE_NAME);
        onCreate(db);
    }
}
```

Create Content Provider Code Sample

```
private DatabaseHelper dbHelper;

@Override
public boolean onCreate()
{
    dbHelper = new DatabaseHelper(getContext());
    return (dbHelper!=null);
}
```

Create Content Provider Code Sample

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder)
{
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

    switch (matcher.match(uri))
    {
        case MEMOS:
            qb.setTables(MEMOS_TABLE_NAME);
            qb.setProjectionMap(map);
            break;

        case SPECIFIC_MEMO:
            qb.setTables(MEMOS_TABLE_NAME);
            qb.setProjectionMap(map);
            qb.appendWhere(Memos._ID + "=" + uri.getPathSegments().get(1));
            break;

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

Create Content Provider Code Sample

```
        String orderBy;
        if (TextUtils.isEmpty(sortOrder))
        {
            orderBy = Memos.DEFAULT_SORT_ORDER;
        } else
        {
            orderBy = sortOrder;
        }

        SQLiteDatabase db = dbHelper.getReadableDatabase();
        Cursor cursor = qb.query(db, projection, selection, selectionArgs,
                null, null, orderBy);

        cursor.setNotificationUri(getContext().getContentResolver(), uri);
        return cursor;
    }
```

Create Content Provider Code Sample

```
@Override
public Uri insert(Uri uri, ContentValues initialValues)
{
    if (matcher.match(uri) != MEMOS)
    {
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    ContentValues values;
    if (initialValues != null)
    {
        values = new ContentValues(initialValues);
    }
    else
    {
        values = new ContentValues();
    }

    Long now = Long.valueOf(System.currentTimeMillis());

    if (values.containsKey(Memos.CREATION_DATE) == false)
    {
        values.put(Memos.CREATION_DATE, now);
    }
```

Create Content Provider Code Sample

```
        if (values.containsKey(Memos.MODIFIED_DATE) == false)
        {
            values.put(Memos.MODIFIED_DATE, now);
        }

        if (values.containsKey(Memos.TITLE) == false)
        {
            values.put(Memos.TITLE, "no title");
        }

        if (values.containsKey(Memos.MEMO) == false)
        {
            values.put(Memos.MEMO, "empty memo");
        }

        SQLiteDatabase db = dbHelper.getWritableDatabase();
        long rowId = db.insert(MEMOS_TABLE_NAME, Memos.MEMO, values);
        if (rowId > 0)
        {
            Uri uriNotify = ContentUris.withAppendedId(Memos.CONTENT_URI,rowId);
            getContext().getContentResolver().notifyChange(uriNotify, null);
            return uriNotify;
        }

        throw new SQLException("Failed to insert row into " + uri);
    }
```

Create Content Provider Code Sample

```
@Override
public int delete(Uri uri, String where, String[] whereArgs)
{
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    int count;
    switch (matcher.match(uri))
    {
        case MEMOS:
        count = db.delete(MEMOS_TABLE_NAME, where, whereArgs);
        break;

        case SPECIFIC_MEMO:
        String id = uri.getPathSegments().get(1);
        count = db.delete(MEMOS_TABLE_NAME, Memos._ID + "=" + id
            + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""),
            whereArgs);
        break;

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

    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
```

Create Content Provider Code Sample

```
    @Override
    public int update(Uri uri, ContentValues values,
        String where, String[] whereArgs)
    {
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        int count;
        switch (matcher.match(uri))
        {
            case MEMOS:
                count = db.update(MEMOS_TABLE_NAME, values, where, whereArgs);
                break;

            case SPECIFIC_MEMO:
                String id = uri.getPathSegments().get(1);
                count = db.update(MEMOS_TABLE_NAME, values, Memos._ID + "=" + id
                + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""),
                whereArgs);
                break;

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

        getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }
}
```

Create Content Provider Code Sample

- The new content provider should be registered within the manifest file of our application.

Create Content Provider Code Sample

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.abelski.android"
      android:versionCode="1"
      android:versionName="1.0">
    <application    android:icon="@drawable/icon"
                    android:label="@string/app_name">
        <activity android:name=".MemoActivity"
                  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="MemoProvider"
            android:authorities="com.abelski.provider.memodb" />

    </application>
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```

Create Content Provider Code Sample

Changes Notification

- We can develop our content provider in such a way that its clients will be notified when the data changes.
- This way, when a client uses a content provider it will be notified of any data changes and will be able to refresh the data it retrieves.

Changes Notification

- We can call the `notifyChange()` method on the `ContentResolver` instance we are working with in order to notify about the change.

```
...
getContext().getContentResolver().notifyChange(uri, null);
...
```

Changes Notification

- On the content consumer side we can place a call to the `registerContentObserver()` on the `ContentResolver` we are working with. In order to untie the connection we can call the `unregisterContentObserver()` method.

```
...
getContext().getContentResolver().
    registerContentObserver(uri,true,observer);
...
getContext().getContentResolver().
    unregisterContentObserver(observer);
...
```

Content Providers

Introduction

- The Android platform allows to expose the data sources (e.g. the SQLite database) through a REST like abstraction, known as a 'Content Provider'.
- In order to retrieve data from a content provider or in order to save data into it we use REST-like URIs.
- The Android platform treats every URI address that starts with `content//` as a URI address that represents a data resource served by a content provider.

Instances of Uri serve as handles for content providers. From a developer perspective it doesn't matter where the data is stored. It can be stored in a database, a flat file or even on another server. The developer neither needs to know or care where the data comes from as long as it is available when needed.

Introduction

- The content provider Uri allows us to perform basic CRUD (create, read, update, delete) operations.
- Each Uri instance represents either a collection of data or a specific individual one.

Introduction

- Assuming a database named 'school' on our android we might use the following URI:

```
content://com.abelski.schoolprovider/student
```

- If we want to access a specific student we might use the following URL:

```
content://com.abelski.schoolprovider/student/34234
```

Introduction

- The URI access mechanism simplifies our access to the content providers we use.
- Content providers allow different applications on the android platform to share data with each other.

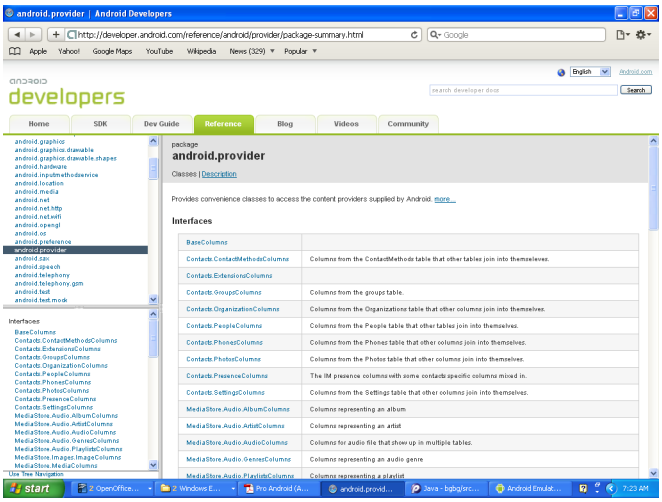
The content providers mechanism is just an abstraction. Unless we want to share data externally or between applications there is no need to use this mechanism.

Android Built-In Content Providers

- The android platform comes with a pre defined list of content providers.

  You can find the list of these predefined content providers browsing at http://developer.android.com/reference/android/provider/package-summary.html

Android Built-In Content Providers



Content Providers Architecture

- Each content provider exposes specific data through a URI similarly to the way a web site exposes its content through URL and similarly to the way a web service provides us with its service.

Content Providers Registration

- Each content provider is registered (similarly to a web site) with a name (similarly to the domain name concept) and a set of URIs (similarly to the URL concept).

The '`android:name`' attribute specifies the name of the class that functions as a content provider. That class must extend the `ContentProvider` class. The `StudentsProvider` class should extend `ContentProvider`.

Content Providers Registration

- The `AndroidManifext.xml` file includes the registration of the content providers we make available for others.
- Each content provider is registered using the `<provider>` XML element.

```
<provider android:name="StudentsProvider"
android:authorities="com.zindell.studentsprovider" />
```


The '`android:name`' attribute specifies the name of the class that functions as a content provider. That class must extend the `ContentProvider` class. The `StudentsProvider` class should extend `ContentProvider`.

Content Providers Registration

- The '`authorities`' attribute in each content provider registration defines the URI through which the content provider will be accessed.
- The URI should be composed of small letters only.
- The previous code sample defines a content provider available for access using the following URI:
  `content://com.zindell.studentsprovider/student/`

Content Providers REST Access

- The content providers' URL functions in a similar way to a REST-like URL address through which data is retrieved or being manipulated.

  Using `content://com.zindell.studentsprovider/student` URL we can get the details of all students.

  Using `content://com.zindell.studentsprovider/student/2342` we can get a specific student identified by 2342.

- The content provider returns the data as a set of rows and columns.

The content providers' URLs also resemble a call to a stored procedure on our database. The small difference is having the input embedded within the URL call instead of having it being sent.

Content Providers URL Structure

- The content provider URIs has the following structure:
  content:// (1) / (2) / (3)
- The (1) part is known as the authorities. The (2) part is known as the data type part and it might point at a collection or a directory. The (3) part is known as the instance identifier and it points at a specific item.
- The following are few examples for possible URIs:

```
content://com.zindell.provider.booksprovider/book/3234
content://com.abelski.provider.coursesprovider/course/2534
```

Content Providers URL Structure

- The data type path can be empty, if the content provider handles only one type of content, a single path segment (e.g. student) or even a chain of path segments (employees/managers).
- The instance identifier is an integer identifying a specific piece of content. The content Uri without the instance identifier refers the whole collection of the content represented by the Uri.

Content Providers Mime Types

- Just as with a response we get from a web server that includes the MIME type describing the response, a content provider returns a MIME type as well.
- The content provider MIME Type is different when getting a collection of records and when getting a specific record.
- The android platform uses the content provider MIME type as a mean for identifying it.

Content Providers Mime Types

- When the content provider returns a single record the mime type should be of the following pattern:

```
vnd.android.cursor.item/ourcompany.contenttype
```

- The following are few possible examples:

```
vnd.android.cursor.item/abelski.course
vnd.android.cursor.item/abelski.student
vnd.android.cursor.item/abelski.topic
```

Content Providers Mime Types

- When the content provider returns a collection of records the mime type should be of the following pattern:

```
vnd.android.cursor.dir/ourcompany.contenttype
```

- The following are few possible examples:

```
vnd.android.cursor.dir/abelski.course
vnd.android.cursor.dir/abelski.student
vnd.android.cursor.dir/abelski.topic
```

Using The Content Provider

- Using a content provider `URIs` it is possible to execute the various state change methods, such as insert, update and delete.

The `Cursor` Object

- When querying a content provider asking for a collection of records we work with a `Cursor` object.

The `Cursor` Object

```
...

String[] vec = new String[] { Contacts.People.NAME};
contacts = Contacts.People.CONTENT_URI;

cursor = managedQuery(
    contacts,
    vec, // specifying which columns return
    null,// specifying which rows to return... in this case all rows
    null,// selection arguments... in this case none
    Contacts.People.NAME + " ASC"); // ascending order by name

int i = 0;
if (cursor.moveToFirst()) {
    String name = null;
    int nameColumn = cursor.getColumnIndex(Contacts.People.NAME);
    do {
        name = cursor.getString(nameColumn);
        builder.append("i="+i+" "+name+"\n");
        i++;
    }
    while (cursor.moveToNext());
}
...
```

The `ContentValues` Object

- When executing methods such as insert,delete and update we work with a `ContentValues` object.

The `ContentValues` Object

```
...
values = new ContentValues();
values.put(Contacts.People.NAME, "Haim Michael");
uri = getContentResolver().insert(Contacts.People.CONTENT_URI, values);
values = new ContentValues();
values.put(Contacts.People.NAME, "Moshe Israeli");
uri = getContentResolver().insert(Contacts.People.CONTENT_URI, values);
...
```

Address Book Content Provider Demo

- The android platform already includes several predefined content providers we can access from within our code.
- One of them is a content provider through which we can interact with the mobile telephone address book. Its URI address is defined at `Contacts.People.CONTENT_URI`.

You can download the code of this application from the samples folder of this topic.

Address Book Content Provider Demo

```
package com.abelski;

import android.app.Activity;
import android.net.Uri;
import android.os.Bundle;
import android.content.ContentValues;
import android.provider.Contacts;
import android.widget.TextView;
import android.database.Cursor;

public class ContactsContentProviderInteraction extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        StringBuilder builder = new StringBuilder();
        try
        {
            // declaring required local variables
            ContentValues values = null;
            Uri uri = null;
            Cursor cursor = null;
            Uri contacts = null;
```


You can download the code of this application from the samples folder of this topic.

Content Provider Demo

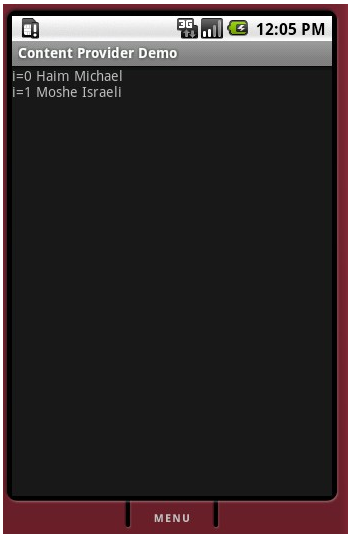
```
// restoring state
super.onCreate(savedInstanceState);
// setting view
setContentView(R.layout.main);

// adding records
values = new ContentValues();
values.put(Contacts.People.NAME, "Haim Michael");
uri = getContentResolver().
    insert(Contacts.People.CONTENT_URI, values);
values = new ContentValues();
values.put(Contacts.People.NAME, "Moshe Israeli");
uri = getContentResolver().insert(
    Contacts.People.CONTENT_URI, values);

// getting all records
String[] vec = new String[] { Contacts.People.NAME};
contacts = Contacts.People.CONTENT_URI;
cursor = managedQuery(
    contacts,
    vec, // specifying return columns
    null,// specifying which rows to return.. this case all rows
    null,// selection arguments... in this case none
    Contacts.People.NAME + " ASC"); // ascending order by name
int i = 0;
```


The list of the available possible properties for a given content provider is provided by the documentation or the source code where the content provider was defined.

Address Book Content Provider Demo

```
            if (cursor.moveToFirst())
            {
                String name = null;
                int nameColumn = cursor.getColumnIndex(Contacts.People.NAME);
                do
                {
                    name = cursor.getString(nameColumn);
                    builder.append("i="+i+" "+name+"\n");
                    i++;
                }
                while (cursor.moveToNext());
            }
        }
        catch (Exception e)
        {
            builder.append(e.getMessage());
        }
        TextView text = (TextView)findViewById(R.id.tf);
        text.setText(builder.toString());
    }
}
```

Address Book Content Provider Demo



The `SimpleCursorAdapter` Class

- This class bridges between a `Cursor` object and a selection widget such as the list view.

When defining a new content provider it doesn't necessarily need to base its work on a database.

The `SimpleCursorAdapter` Class

```
public class SimpleContentProviderDemoActivity extends Activity
{
    @Override
    public void onCreate(Bundle bndl)
    {
        super.onCreate(bndl);
        ContentResolver resolver = getContentResolver();
        Cursor cursor =
            resolver.query(Settings.System.CONTENT_URI,
            null, null,null, null);
        startManagingCursor(cursor);
        ListView list = new ListView(this);
        String[] from = new String[]
            { Settings.System.NAME, Settings.System.VALUE };
        int[] to = new int[]
            { R.id.thename, R.id.thevalue };
        SimpleCursorAdapter adapter = new SimpleCursorAdapter(
            this,R.layout.row, cursor, from, to);
        list.setAdapter(adapter);
        setContentView(list);
    }
}
```

The Activity Source Code

When defining a new content provider it doesn't necessarily need to base its work on a database.

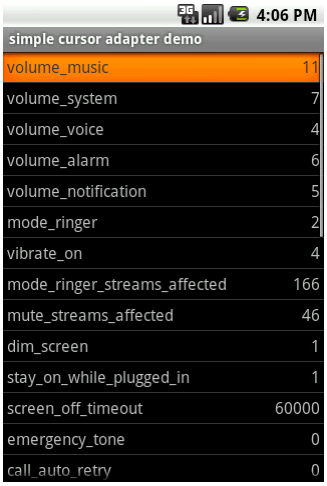
The `SimpleCursorAdapter` Class

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="wrap_content"
    android:padding="5sp"
    android:layout_width="fill_parent">
    <TextView
        android:layout_height="wrap_content"
        android:id="@+id/thename"
        android:textSize="22sp"
        android:layout_width="fill_parent"
        android:layout_weight="1"></TextView>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/thevalue"
        android:textSize="22sp"
        android:gravity="right"></TextView>
</LinearLayout>
```

The row.xml Layout File

When defining a new content provider it doesn't necessarily need to base its work on a database.

The `SimpleCursorAdapter` Class



When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Our Own Content Provider

- The drawback in using a database directly is its access limitation. Once a database is created it is visible for the application that created it only. The SQLite database created on Android by one application is usable by that application only. Other applications cannot use it.
- Creating a content provider will allow our data to be accessible from other applications.

Create Our Own Content Provider

- In order to create a new content provide we should define a class that extends the `ContentProvider` abstract class.
- It is a common practice to include within the definition of the new content provider an inner class through which the content provider will access the database.

When defining a new content provider it doesn't necessarily need to base its work on a database. We can define a new content provider that bases its work on a flat file or even on a web service or networking with another application running on another computer.

Create Our Own Content Provider

- Within the class we define as a class that extends `ContentProvider` we are responsible for implementing six methods: `onCreate()`, `query()`, `insert()`, `update()`, `delete()` and `getType()`.

The `onCreate()` Method

- This method is the entry point to the content provider. Within this method we will initialize our content provider.
- This method should return true in order to indicate that the content provider is ready for use.

```
@Override
public boolean onCreate()
{
    db=(new DatabaseHelper(getContext())).
        GetWritableDatabase();
    return (db == null) ? false : true;
}
```

The `query()` Method

- When the `query()` method is been called by the `ContentResolver` object our content provider gets the details of the query the activity wants to perform.

```
...
public abstract Cursor query(
    Uri  uri,
    String[]  projection,
    String  selection,
    String[]  selectionArgs,
    String  sortOrder)
...
```

The `query()` Method

- The returned `Cursor` will be used to iterate through the returned records.

```
...
if (cursor.moveToFirst()) {
    ...
    do {
        ...
    }
    while(cursor.moveToNext());
}
...
```

The `query()` Method

- The query method is indirectly been called when calling the `managedQuery` method. The parameters of these two methods match with each other.

The `query()` Method

```
public final Cursor managedQuery (    Uri  uri,
                                      String[] projection,
                                      String selection,
                                      String[] selectionArgs,
                                      String sortOrder)
{
        ...
}
```

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

The `query()` Method

- The `Uri` parameter represents the collection or the instance being queried.
- The `projection` String[] parameter array represents the list of properties the content provider should return.
- The `selection` String parameter is the where clause of the query.

The `query()` Method

- The `selectionArgs` String[] parameter is an array of values that substitute ? marks the selection string includes.
- The `sortOrder` parameter is the sort clause of the query.

The `insert()` Method

- The `insert()` method receives a `Uri` representing the collection to which we want to add the new values and a `ContentValues` object that includes the initial data of our new instance.

```
...
public Uri  insert (Uri  uri, ContentValues  values)
{
    ...
}
...
```

The `insert()` Method

- Our implementation for the `insert()` method is responsible for creating the new data instance (e.g. new table raw) and returning a `Uri` for the new instance.

The `update()` Method

- The `update()` method gets the `Uri` of the collection or of the instance that needs to be changed.

```
...
public int update ( Uri uri,
                    ContentValues values,
                    String selection,
                    String[] selectionArgs)
{
    ...
}
...
```

The `update()` Method

- The `uri` parameter represents the collection or the instance that we want to change.
- The `values` parameter includes the new values we want to set. This parameter is of the `ContentValues` type.
- The `selection` string holds the where clause and the `selectionArgs` array of strings holds the parameters to use for replacing the ? Characters the where clause includes.

The `update()` Method

- Our implementation for this method should identify the instance or the instances to be modified based on the `uri` and the `selection` where clause and then replace the instance(s) current property values with the ones supplied.

The `delete()` Method

- Our definition for the `delete()` method receives a `uri` that represents the instance or the collection that we need to delete and a `selection` where clause together with the `selectionArgs` parameters.

```
...
public int delete(
    Uri uri, String selection, String[] selectionArgs)
{
    ...
}
...
```

The `getType()` Method

- This method receives a `uri` object and returns its MIME type. The `uri` can represent either a collection or an instance.
- Our implementation for this method should determine which type of `uri` was received and return the corresponding MIME type.

```
...
public String getType(Uri url){
    ...
}
...
```

Create Content Provider Code Sample

```
public class MemoProvider extends ContentProvider
{
    private static final String LOGGER_TAG = "MemoProvider";
    private static final String DATABASE_NAME = "memos.db";
    private static final int DATABASE_VERSION = 2;
    private static final String MEMOS_TABLE_NAME = "memos";
    private static HashMap<String, String> map;
    private static final int MEMOS = 1;
    private static final int SPECIFIC_MEMO = 2;
    private static final UriMatcher matcher;

    static
    {
        matcher = new UriMatcher(UriMatcher.NO_MATCH);
        matcher.addURI(Memos.AUTHORITY, "memos", MEMOS);
        matcher.addURI(Memos.AUTHORITY, "memos/#", SPECIFIC_MEMO);
        map = new HashMap<String, String>();
        map.put(Memos._ID, Memos._ID);
        map.put(Memos.TITLE, Memos.TITLE);
        map.put(Memos.MEMO, Memos.MEMO);
        map.put(Memos.CREATION_DATE, Memos.CREATION_DATE);
        map.put(Memos.MODIFIED_DATE, Memos.MODIFIED_DATE);
    }
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
public static class Memos implements BaseColumns
{
    public static final String AUTHORITY = "com.abelski.provider.memodb";
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/memos");
    public static final String CONTENT_TYPE =
        "vnd.android.cursor.dir/vnd.abelski.memo";
    public static final String CONTENT_ITEM_TYPE =
        "vnd.android.cursor.item/vnd.abelski.memo";
    public static final String DEFAULT_SORT_ORDER = "modified DESC";
    public static final String TITLE = "title";
    public static final String MEMO = "memo";
    public static final String CREATION_DATE = "created";
    public static final String MODIFIED_DATE = "modified";
}
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
private static class DatabaseHelper extends SQLiteOpenHelper
{
    DatabaseHelper(Context context)
    {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db)
    {
        db.execSQL("CREATE TABLE " + MEMOS_TABLE_NAME + " (" + Memos._ID
                + " INTEGER PRIMARY KEY," + Memos.TITLE + " TEXT,"
                + Memos.MEMO + " TEXT," + Memos.CREATION_DATE + " INTEGER,"
                + Memos.MODIFIED_DATE + " INTEGER" + ");");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    {
        Log.w(LOGGER_TAG, "Upgrading the database from version "
                + oldVersion + " to " + newVersion
                + ", old data is deleted");
        db.execSQL("DROP TABLE IF EXISTS " + MEMOS_TABLE_NAME);
        onCreate(db);
    }
}
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
private DatabaseHelper dbHelper;

@Override
public boolean onCreate()
{
    dbHelper = new DatabaseHelper(getContext());
    return (dbHelper!=null);
}
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder)
{
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

    switch (matcher.match(uri))
    {
        case MEMOS:
            qb.setTables(MEMOS_TABLE_NAME);
            qb.setProjectionMap(map);
            break;

        case SPECIFIC_MEMO:
            qb.setTables(MEMOS_TABLE_NAME);
            qb.setProjectionMap(map);
            qb.appendWhere(Memos._ID + "=" + uri.getPathSegments().get(1));
            break;

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


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
        String orderBy;
        if (TextUtils.isEmpty(sortOrder))
        {
            orderBy = Memos.DEFAULT_SORT_ORDER;
        } else
        {
            orderBy = sortOrder;
        }

        SQLiteDatabase db = dbHelper.getReadableDatabase();
        Cursor cursor = qb.query(db, projection, selection, selectionArgs,
                null, null, orderBy);

        cursor.setNotificationUri(getContext().getContentResolver(), uri);
        return cursor;
    }
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
@Override
public Uri insert(Uri uri, ContentValues initialValues)
{
    if (matcher.match(uri) != MEMOS)
    {
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    ContentValues values;
    if (initialValues != null)
    {
        values = new ContentValues(initialValues);
    }
    else
    {
        values = new ContentValues();
    }

    Long now = Long.valueOf(System.currentTimeMillis());

    if (values.containsKey(Memos.CREATION_DATE) == false)
    {
        values.put(Memos.CREATION_DATE, now);
    }
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
        if (values.containsKey(Memos.MODIFIED_DATE) == false)
        {
            values.put(Memos.MODIFIED_DATE, now);
        }

        if (values.containsKey(Memos.TITLE) == false)
        {
            values.put(Memos.TITLE, "no title");
        }

        if (values.containsKey(Memos.MEMO) == false)
        {
            values.put(Memos.MEMO, "empty memo");
        }

        SQLiteDatabase db = dbHelper.getWritableDatabase();
        long rowId = db.insert(MEMOS_TABLE_NAME, Memos.MEMO, values);
        if (rowId > 0)
        {
            Uri uriNotify = ContentUris.withAppendedId(Memos.CONTENT_URI,rowId);
            getContext().getContentResolver().notifyChange(uriNotify, null);
            return uriNotify;
        }

        throw new SQLException("Failed to insert row into " + uri);
    }
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
@Override
public int delete(Uri uri, String where, String[] whereArgs)
{
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    int count;
    switch (matcher.match(uri))
    {
        case MEMOS:
        count = db.delete(MEMOS_TABLE_NAME, where, whereArgs);
        break;

        case SPECIFIC_MEMO:
        String id = uri.getPathSegments().get(1);
        count = db.delete(MEMOS_TABLE_NAME, Memos._ID + "=" + id
            + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""),
            whereArgs);
        break;

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

    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
    @Override
    public int update(Uri uri, ContentValues values,
        String where, String[] whereArgs)
    {
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        int count;
        switch (matcher.match(uri))
        {
            case MEMOS:
                count = db.update(MEMOS_TABLE_NAME, values, where, whereArgs);
                break;

            case SPECIFIC_MEMO:
                String id = uri.getPathSegments().get(1);
                count = db.update(MEMOS_TABLE_NAME, values, Memos._ID + "=" + id
                + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""),
                whereArgs);
                break;

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

        getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }
}
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

- The new content provider should be registered within the manifest file of our application.

When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.abelski.android"
      android:versionCode="1"
      android:versionName="1.0">
    <application    android:icon="@drawable/icon"
                    android:label="@string/app_name">
        <activity android:name=".MemoActivity"
                  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="MemoProvider"
            android:authorities="com.abelski.provider.memodb" />

    </application>
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```


When defining a new content provider it doesn't necessarily need to base its work on a database.

Create Content Provider Code Sample

Changes Notification

- We can develop our content provider in such a way that its clients will be notified when the data changes.
- This way, when a client uses a content provider it will be notified of any data changes and will be able to refresh the data it retrieves.

Changes Notification

- We can call the `notifyChange()` method on the `ContentResolver` instance we are working with in order to notify about the change.

```
...
getContext().getContentResolver().notifyChange(uri, null);
...
```


The second parameter is a reference for a `ContentObserver` object that represents the one who originated the change. It can be null.

Changes Notification

- On the content consumer side we can place a call to the `registerContentObserver()` on the `ContentResolver` we are working with. In order to untie the connection we can call the `unregisterContentObserver()` method.

```
...
getContext().getContentResolver().
    registerContentObserver(uri,true,observer);
...
getContext().getContentResolver().
    unregisterContentObserver(observer);
...
```