Understanding Android local data store APIs

Take advantage of preferences, SQLite, and the internal and external memory APIs

Skill Level: Intermediate

C. Enrique Ortiz
Developer and author
About Mobility Weblog

30 Nov 2010

The ability to store data locally on the mobile device is a critical function for mobile applications that are required to maintain essential information across application-executions or the lifetime of the application. As a developer, you constantly need to store information such as user preferences or application configurations. You must also decide if you need to tap internal or external storage, depending on characteristics, such as access visibility, or if you need to handle more complex, structured types of data. Follow along in this article to learn about Android data storage APIs, specifically the preferences, SQLite, and the internal and external memory APIs.

Frequently used acronyms

- ADT: Android Development Tools
- API: Application program interface
- IDE: Integrated development environment
- JDK: Java Development Kit
- JSON: JavaScript Object Notation
- SDK: Software Development Kit
- SQL: Structured Query Language
- UI: User interface
- XML: Extensible Markup Language
Prerequisites

To follow along with this article, you need the following skills and tools:

- Basic knowledge of Java™ technology and how to use Eclipse (or your favorite IDE)
- Java Development Kit (version 5 or 6 required)
- Eclipse (version 3.4 or 3.5)
- Android SDK and ADT plug-in

For download and setup information, see Resources at the end of this article.

The sample application

To highlight the local store aspects of Android application development, I cover a sample application that allows you to test the execution of various types of APIs. The source code is available for download. The application supports the actions in Figure 1.

Figure 1. The use cases
Figure 1 lists the following use cases:

- Manage and store preferences
- Load information from application assets
- Export information to internal memory, external memory, and the local database
- Read information from internal memory and the local database
- Clear stored information
- See information on the screen

You make use of the local store in the application throughout this article, as follows:

- Preferences are captured from the user, stored locally, and used
throughout the application.

- A picture of the user is retrieved from internal application assets, stored in local internal memory and external memory, and rendered on the screen.
- A list of friends in JSON is retrieved from the application's assets. It is parsed and stored in local internal memory, external memory, and the relational database, and rendered on the screen.

The sample application defines the classes in Table 1.

Table 1. Sample application classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainActivity</td>
<td>The Main Activity; where most of the sample code resides</td>
</tr>
<tr>
<td>Friend</td>
<td>Represents a Friend</td>
</tr>
<tr>
<td>AppPreferenceActivity</td>
<td>Preferences Activity and screen</td>
</tr>
<tr>
<td>DBHelper</td>
<td>A helper class for the management of the SQLite database</td>
</tr>
</tbody>
</table>

The example application uses two types of data. The first one is the application preferences that are stored as name-value pairs. For preferences, define the following information:

- A filename, used to load and store the list of the friends' names
- A filename, used to load and store a picture for the user
- A flag, if set, indicates to automatically delete all stored data at application startup

The second type of data is a friends list. The friends list is initially represented in a Facebook Graph API JSON format, which consists of an array of name and friend objects (see Listing 1).

Listing 1. The friends list (in Facebook Graph API JSON format)

```json
{
  "data": [
    {
      "name": "Edmund Troche",
      "id": "5000676799"
    }
  ]
}
```

The simple format above makes the Friend object and database schema simple. Listing 2 shows the Friend class.
Listing 2. Friend class

```java
package com.cenriqueortiz.tutorials.datastore;
import android.graphics.Bitmap;
/**
 * Represents a Friend
 */
public class Friend {
    public String id;
    public String name;
    public byte[] picture;
    public Bitmap pictureBitmap;
}
```

In addition to the ID and name, the sample application also keeps references to the friend's picture. While the sample application is not using those, you can easily extend the sample application to retrieve the picture from Facebook and show it in the main screen.

The database schema consists of a single table to store the Friend's information. It has three columns:

- Unique ID or key
- Facebook ID
- The Friend's name

Listing 3 shows the SQL statement for the corresponding relational table declaration.

Listing 3. Friend database table

```
db.execSQL("create table " + TABLE_NAME + " (_id integer primary key autoincrement, " + " fid text not null, name text not null) ");
```

Based on this information, you can display the name on the main screen, and using the ID, you can retrieve additional details for the selected user. In the sample application, you just display the names. It is left to you to experiment with retrieving additional information. Note that you can easily change the code to go directly to Facebook.

Storing application preferences

This section covers the Preferences API and screens. The Android API provides a number of ways to deal with preferences. One way is to use SharedPreferences directly and use your own screen design and management of the preferences. The second approach is to use a PreferenceActivity. A PreferenceActivity
automatically takes care of how the preferences are rendered on the screen (by
default they look like the system's preferences) and automatically stores or saves
the preferences as the user interacts with each preference through the use of
SharedPreferences.

To simplify the sample application, use a PreferenceActivity to manage the
preferences and the preference screen (see Figure 2). The preferences screen
displays two sections: Assets and Auto Settings. Under Assets, you can enter
filenames for both the Friends List and Picture options. Under Auto Settings, you can
select a check box to delete information at startup.

Figure 2. The Preferences screen as implemented

In Figure 2, the layout was defined using the declarative approach through XML
(instead of programmatically); declarative XML is preferred as it keeps the source
code clean and readable. Listing 4 shows the XML declaration for the Preferences
UI.

Listing 4. The Preferences screen XML declaration

```xml
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
```
The PreferenceScreen consists of two instances of EditTextPreference, a CheckBoxPreference, and the two category groups as defined by PreferenceCategory (one for Asset and the other for Auto Settings).

In the sample application, the design calls for the Preference screen to be invoked using a menu item. For this, use an Intent message to invoke the Preference Screen Activity called AppPreferenceActivity (see Listing 5). Note that I do not cover how Intent works in detail. See Resources for more information on Intents.

Listing 5. The AppPreferenceActivity

```java
/*
 * AppPreferenceActivity is a basic PreferenceActivity
 * C. Enrique Ortiz | http://CEnriqueOrtiz.com
 */
package com.cenriqueortiz.tutorials.datastore;

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class AppPreferenceActivity extends PreferenceActivity {

    /**
     * Default Constructor
     */
    public AppPreferenceActivity() {}
    
    /**
     * Called when the activity is first created.
     * Inflate the Preferences Screen XML declaration.
     */
    @Override
```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.prefs); // Inflate the XML declaration
}

In the sample application, invoke the Intent as in Listing 6, from inside the Menu item handler.

Listing 6. Invoking the Preference activity using an Intent

/**
 * Invoked when a menu item has been selected
 */
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        // Case: Bring up the Preferences Screen
        case R.id.menu_prefs: // Preferences
            // Launch the Preference Activity
            Intent i = new Intent(this, AppPreferenceActivity.class);
            startActivity(i);
            break;
        case R.id.menu...:
            break;
    }
    return true;
}

In addition, you must define all Intents in the AndroidManifest XML file as in Listing 7.

Listing 7. Defining the Intent in AndroidManifest.xml

Recall that PreferenceActivity uses SharedPreferences to automatically store the preferences as the user interacts with the preferences screen. The
application then uses these preferences when it executes to perform its various tasks. Listing 8 shows how to use SharedPreferences directly to load the stored preferences; you can refer to the companion sample code on how the loaded preferences are used throughout the sample code. In addition, Listing 8 also shows how to store preferences directly with SharedPreferences using an Editor in case you prefer to manage the preferences yourself (and not through PreferenceActivity).

Listing 8 shows how to use SharedPreferences to load the stored preferences and how to make changes to the stored preferences using an Editor.

Listing 8. Using SharedPreferences

```java
public boolean prefsGetAutoDelete() {
    boolean v = false;
    String key = appContext.getString(R.string.prefs_autodelete_key);
    try {
        v = sprefs.getBoolean(key, false);
    } catch (ClassCastException e) {
    }
    return v;
}

public void prefsSetAutoDelete(boolean v) {
    String key = appContext.getString(R.string.prefs_autodelete_key);
    Editor e = sprefs.edit();
    e.putBoolean(key, v);
    e.commit();
}
```

Next, you will see how to use a database to store data.

Using SQLite databases

Android provides support to local relational databases through SQLite. The table (defined in the following listings) summarizes the important database classes used in the sample application.

For the sample application, a DBHelper class is used to encapsulate some of the
database operations (see Listing 9).

**Listing 9. The DBHelper**

```java
package com.cenriqueortiz.tutorials.datastore;
import java.util.ArrayList;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
public class DBHelper extends SQLiteOpenHelper {
    A number of constants are defined for the database version, database name, and table name (see Listing 10).

**Listing 10. Initializing the DBHelper**

```java
private SQLiteDatabase db;
private static final int DATABASE_VERSION = 1;
private static final String DB_NAME = "sample.db";
private static final String TABLE_NAME = "friends";
/**
 * Constructor
 * @param context the application context
 */
public DBHelper(Context context) {
    super(context, DB_NAME, null, DATABASE_VERSION);
    db = getWritableDatabase();
}
```

The `onCreate()` method is invoked when the database is ready to be created. In this method, the tables are created (see Listing 11).

**Listing 11. Creating a database table**

```java
/**
 * Called at the time to create the DB.
 * The create DB statement
 * @param the SQLite DB
 */
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(
            "create table " + TABLE_NAME + " (_id integer primary key autoincrement,
" + " fid text not null, name text not null) ");
}
```

The `insert()` method is invoked by MainActivity when exporting information to the database (see Listing 12).

**Listing 12. Inserting a row**
The `insert(String id, String name)` method is invoked by `MainActivity` when adding a new friend. It inserts the friend's ID and name into the database (see Listing 13).

Listing 13. Deleting the database table

```java
/**
 * Wipe out the DB
 */
public void clearAll() {
    db.delete(TABLE_NAME, null, null);
}
```

Two `SELECT ALL` methods are provided: `cursorSelectAll()`, which returns a cursor, and `listSelectAll()`, which returns an `ArrayList` of `Friend` objects. These methods are invoked by `MainActivity` when loading information from the database (see Listing 14).

Listing 14. Running a Select All that returns a cursor

```java
/**
 * Select All returns a cursor
 */
public Cursor cursorSelectAll() {
    Cursor cursor = this.db.query(
            TABLE_NAME, // Table Name
            new String[] { "fid", "name" }, // Columns to return
            null, // SQL WHERE
            null, // Selection Args
            null, // SQL GROUP BY
            null, // SQL HAVING
            "name"); // SQL ORDER BY
    return cursor;
}
```

The `listSelectAll()` method returns the selected row inside an `ArrayList` container that is used by `MainActivity` to bind it to the `MainScreen` `ListView` (see Listing 15).

Listing 15. Running a Select All that returns a cursor

```java
```
/**
   * Select All that returns an ArrayList
   * @return the ArrayList for the DB selection
   */
public ArrayList<Friend> listSelectAll() {
    ArrayList<Friend> list = new ArrayList<Friend>();
    Cursor cursor = this.db.query(TABLE_NAME, new String[] { "fid", "name" },
null, null, null, null, "name");
    if (cursor.moveToFirst()) {
        do {
            Friend f = new Friend();
            f.id = cursor.getString(0);
            f.name = cursor.getString(1);
            list.add(f);
        } while (cursor.moveToNext());
    }
    if (cursor != null && !cursor.isClosed()) {
        cursor.close();
    }
    return list;
}

The onUpgrade() method is invoked if a database version change is detected (see
Listing 16).

Listing 16. Detecting if a database version changes

/**
   * Invoked if a DB upgrade (version change) has been detected
   */
@Override
/**
   * Invoked if a DB upgrade (version change) has been detected
   */
@Override
public void onUpgrade(SQLiteDatabase db,
int oldVersion, int newVersion) {
    // Here add any steps needed due to version upgrade
    // for example, data format conversions, old tables
    // no longer needed, etc
}

Throughout MainActivity, DBHelper is used when you export information to the
database, load information from the database, and when you clear the database.
The first thing is to instantiate the DBHelper when MainActivity is created.
Other tasks performed at onCreate() include initializing the different screen views
(see Listing 17).

Listing 17. MainActivity onCreate() initializing the database

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    appContext = this;
    setContentView(R.layout.main);
    dbHelper = new DBHelper(this);
    listView = (ListView) findViewById(R.id.friendsview);
    friendsArrayAdapter = new FriendsArrayAdapter(
Listing 18 shows how to load the friends list from the assets and how to parse and insert it into the database.

**Listing 18. MainActivity inserting into the database**

```java
String fname = prefsGetFilename();
if (fname != null && fname.length() > 0) {
    buffer = getAsset(fname);
    // Parse the JSON file
    String friendslist = new String(buffer);
    final JSONObject json = new JSONObject(friendslist);
    JSONArray d = json.getJSONArray("data");
    int l = d.length();
    for (int i2=0; i2<l; i2++) {
        JSONObject o = d.getJSONObject(i2);
        String n = o.getString("name");
        String id = o.getString("id");
        dbHelper.insert(id, n);
    }
    // Only the original owner thread can touch its views
    MainActivity.this.runOnUiThread(new Runnable() {
        public void run() {
            friendsArrayAdapter.notifyDataSetChanged();
        }
    });
}
```

Listing 19 shows how to perform a SELECT ALL and how to bind the data to the main screen ListView.

**Listing 19. MainActivity Select All and binding the data to ListView**

```java
final ArrayList<Friend> dbFriends = dbHelper.listSelectAll();
if (dbFriends != null) {
    // Only the original owner thread can touch its views
    MainActivity.this.runOnUiThread(new Runnable() {
        public void run() {
            friendsArrayAdapter = new FriendsArrayAdapter(
                MainActivity.this, R.layout.rowlayout, dbFriends);
            listView.setAdapter(friendsArrayAdapter);
            friendsArrayAdapter.notifyDataSetChanged();
        }
    });
}
```

Next, take a look at using the Internal Storage API with the example application.

**Using the device's internal storage for private data**
With the data storage API, you can store data using the internal storage. The information can be private, and you have the option of letting other applications have read or write access to it. This section covers the API to store private data using `android.content.Context.openFileInput`, `openFileOutput`, and `getCacheDir()` to cache data rather than store it persistently.

The snippet shown in Listing 20 shows how to read from the internal private store. What makes the storage private is the use of `openFileOutput()` with `MODE_PRIVATE`.

**Listing 20. Reading from the local private store**

```java
/**
 * Writes content to internal storage making the content private to
 * the application. The method can be easily changed to take the MODE
 * as argument and let the caller dictate the visibility:
 * MODE_PRIVATE, MODE_WORLD_WRITEABLE, MODE_WORLD_READABLE, etc.
 *
 * @param filename - the name of the file to create
 * @param content - the content to write
 */
public void writeInternalStoragePrivate(
    String filename, byte[] content) {
    try {
        //MODE_PRIVATE creates/replaces a file and makes
        // it private to your application. Other modes:
        // MODE_WORLD_WRITEABLE
        // MODE_WORLD_READABLE
        // MODE_APPEND
        FileOutputStream fos =
            openFileOutput(filename, Context.MODE_PRIVATE);
        fos.write(content);
        fos.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

The snippet in Listing 21 shows how to read from the internal private store; see the use of `openFileInput()`.

**Listing 21. Reading from the local private store**

```java
/**
 * Reads a file from internal storage
 * @param filename the file to read from
 * @return the file content
 */
public byte[] readInternalStoragePrivate(String filename) {
    int len = 1024;
    byte[] buffer = new byte[len];
    try {
        FileInputStream fis = openFileInput(filename);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        int nrb = fis.read(buffer, 0, len); // read up to len bytes
        while (nrb != -1) {
```
Listing 22 shows how to delete from the internal private store.

Listing 22. Deleting from the local private store

```java
/**
 * Delete internal private file
 * @param filename - the filename to delete
 */
public void deleteInternalStoragePrivate(String filename) {
    File file = getFileStreamPath(filename);
    if (file != null) {
        file.delete();
    }
}
```

Now you can see how to use external storage for public data.

Using the device's external storage for public data

With the data storage API, you can store data using the external storage. The information can be private, and you have the option of letting other applications have read or write access to it. In this section, you code the API to store public data using a number of APIs including `getExternalStorageState()`, `getExternalFilesDir()`, `getExternalStorageDirectory()`, and `getExternalStoragePublicDirectory()`. You use the following path for public data: `/Android/data/<package_name>/files/`.

Before using the external storage, you must see if it is available, and if it is writable. The following two code snippets show helper methods to test for such conditions. Listing 23 tests whether the external storage is available.

Listing 23. Testing if external storage is available

```java
/**
 * Helper Method to Test if external Storage is Available
 */
public boolean isExternalStorageAvailable() {
    boolean state = false;
    String extStorageState = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(extStorageState)) {
    state = true;
} return state;

Listing 24 tests whether the external storage is read-only.

Listing 24. Testing if external storage is read-only

/**
 * Helper Method to Test if external Storage is read only
 */
public boolean isExternalStorageReadOnly() {
    boolean state = false;
    String extStorageState = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(extStorageState)) {
        state = true;
    }
    return state;
}

Listing 25 shows how to write to external storage to store public data.

Listing 25. Writing to external memory

/**
 * Write to external public directory
 * @param filename - the filename to write to
 * @param content - the content to write
 */
public void writeToExternalStoragePublic(String filename, byte[] content) {
    // API Level 7 or lower, use getExternalStorageDirectory()
    // to open a File that represents the root of the external
    // storage, but writing to root is not recommended, and instead
    // application should write to application-specific directory, as shown below.
    String packageName = this.getPackageName();
    String path = "/Android/data/" + packageName + "/files/";
    if (isExternalStorageAvailable() &&
        !isExternalStorageReadOnly()) {
        try {
            File file = new File(path, filename);
            file.mkdirs();
            FileOutputStream fos = new FileOutputStream(file);
            fos.write(content);
            fos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

Listing 26 shows how to read from external storage.
Listing 26. Reading from external memory

```java
/**
 * Reads a file from internal storage
 * @param filename - the filename to read from
 * @return the file contents
 */
public byte[] readExternallStoragePublic(String filename) {
    int len = 1024;
    byte[] buffer = new byte[len];
    String packageName = this.getPackageName();
    String path = "/Android/data/" + packageName + "/files/";
    if (!isExternalStorageReadOnly()) {
        try {
            File file = new File(path, filename);
            FileInputStream fis = new FileInputStream(file);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int nrb = fis.read(buffer, 0, len); //read up to len bytes
            while (nrb != -1) {
                baos.write(buffer, 0, nrb);
                nrb = fis.read(buffer, 0, len);
            }
            buffer = baos.toByteArray();
            fis.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return buffer;
}
```

The Listing 27 snippet shows how to delete a file from external memory.

Listing 27. Deleting a file from external memory

```java
/**
 * Delete external public file
 * @param filename - the filename to write to
 */
void deleteExternalStoragePublicFile(String filename) {
   String packageName = this.getPackageName();
   String path = "/Android/data/" + packageName + "/files/"+filename;
   File file = new File(path, filename);
   if (file != null) {
       file.delete();
   }
}
```

Working with external storage requires a special permission, WRITE_EXTERNAL_STORAGE, to be requested through the AndroidManifest.xml (see Listing 28).

Listing 28. The WRITE_EXTERNAL_STORAGE

```xml
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```
The external storage API allows you to store files publicly by storing the files in predefined directories based on their types such as Pictures, Ringtones, and so on. This approach is not covered in this article, but you should become familiar with it. In addition, remember that files in the external storage can disappear at any time.

Related methods

If you have temporary files that do not need long-term persistence, you can store those files in a cache. Cache is a special memory useful for storing mid-sized or small data (less than a megabyte), but you must be aware that contents of the cache can be purged at any time depending on how much memory is available.

Listing 29 shows a helper method that returns the path to the cache in internal memory.

Listing 29. Retrieving the path to the internal memory cache

```java
/**
 * Helper method to retrieve the absolute path to the application specific internal cache directory on the file system. These files will be ones that get deleted when the application is uninstalled or when the device runs low on storage. There is no guarantee when these files will be deleted.
 * Note: This uses a Level 8+ API.
 * @return the absolute path to the application specific cache directory
 */
public String getInternalCacheDirectory() {
    String cacheDirPath = null;
    File cacheDir = getCacheDir();
    if (cacheDir != null) {
        cacheDirPath = cacheDir.getPath();
    }
    return cacheDirPath;
}
```

Listing 30 shows a helper method that returns the path to the cache in external memory.

Listing 30. Retrieving the path to the external memory cache

```java
/**
 * Helper method to retrieve the absolute path to the application specific external cache directory on the file system. These files will be ones that get deleted when the application is uninstalled or when the device runs low on storage. There is no guarantee when these files will be deleted.
 * Note: This uses a Level 8+ API.
 */
public String getExternalCacheDirectory() {
    String cacheDirPath = null;
    File cacheDir = getCacheDir();
    if (cacheDir != null) {
        cacheDirPath = cacheDir.getPath();
    }
    return cacheDirPath;
}
```
public String getExternalCacheDirectory() {
    String extCacheDirPath = null;
    File cacheDir = getExternalCacheDir();
    if (cacheDir != null) {
        extCacheDirPath = cacheDir.getPath();
    }
    return extCacheDirPath;
}

Through the use of the example application, you should now have a good understanding of how to use the device's external storage for public data.

Conclusion

This article covered the Android storage APIs, from preferences to using SQLite and internal and external memory. With the preferences API, you can have your application collect and store simple preference information. Using the SQLite API, you can store more complex data, and with the internal and external storage, you can store files that are private to the application or publicly available to other applications. Stored data that persists across sessions allows your application to work even when disconnected from the network. You should now have the expertise to take advantage of all of these types of storage when you develop Android applications.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>android.storage.source.zip</td>
<td>38KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

Information about download methods
Resources

Learn

- **Facebook mobile web page**: Learn how to incorporate Facebook into your own mobile application using the same APIs as those provided for all websites, formatted to fit on a mobile phone.

- **Create an Application - Facebook Developers**: Register your Facebook application.

- **Extended permissions - Facebook Developers**: Request extended permissions if your application needs to access other parts of the user’s profile that might be private, or if your application needs to publish content to Facebook on a user's behalf.

- The official **Facebook developer documentation**: Explore the powerful APIs that enable you to create social experiences to drive growth and engagement on your website.

- **Facebook Authentication**: Learn about authentication and authorization when using the Facebook platform to develop your applications.

- The **Facebook Developer Roadmap**: Use this roadmap to plan for changes that might require code modifications.

- **Develop Android applications with Eclipse** (Frank Ableson, developerWorks, February 2008): The easiest way to develop Android applications is to use Eclipse. Learn all about this topic in this developerWorks tutorial.

- **Introduction to Android development** (Frank Ableson, developerWorks, May 2009): Get an introduction to the Android platform and learn how to code a basic Android application.

- **Networking with Android** (Frank Ableson, developerWorks, June 2009): Explore the networking capabilities of Android.

- **Working with XML on Android** (Michael Galpin, developerWorks, June 2009): Learn about the different options for working with XML on Android and how to use them to build your own Android applications.

- **Under the Hood of Native Web Apps for Android**: Learn about hybrid applications in Android.


• Read the **Data Storage Dev Guide**: Choose the right solution to save persistent application data as noted on the official Android Developer website.

• **Intents**: Learn about this abstract description of an operation to be performed from the Android Developer site.

• **Android SDK documentation**: Get the latest information in the Android API reference.

• **The Open Handset Alliance**: Visit Android's sponsor.

• **More articles by this author** (C. Enrique Ortiz, developerWorks, July 2004-current): Read articles about Android, mobile applications, web services, and other technologies.

• **XML area on developerWorks**: Get the resources you need to advance your skills in the XML arena.

• **My developerWorks**: Personalize your developerWorks experience.

• **IBM XML certification**: Find out how you can become an IBM-Certified Developer in XML and related technologies.

• **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks. Also, read more XML tips.

• **developerWorks technical events and webcasts**: Stay current with technology in these sessions.

• **developerWorks on Twitter**: Join today to follow developerWorks tweets.

• **developerWorks podcasts**: Listen to interesting interviews and discussions for software developers.

• **developerWorks on-demand demos**: Watch demos ranging from product installation and setup for beginners to advanced functionality for experienced developers.

**Get products and technologies**

• **Facebook Android SDK** (Currently an Alpha release): Work with this library to integrate Facebook into your Android mobile application.

• The **Facebook Platform**: Expand your ability to build social applications on Facebook and the web.

• The **Facebook Old Rest API** (The previous version of the Graph API): Interact with the Facebook website programmatically through simple HTTP requests.

• The **Facebook Graph API** (The current version): Dig into this core Facebook Platform API.
• **OAuth 2.0 Protocol specification** (July 2010): Work with OAuth authentication, supported by the Facebook Platform.

• **Android SDK**: Download the SDK, access the API reference, and get the latest news on Android from the official Android developers' site. Version 1.5 and later will work.

• The **Android Open Source Project**: Find the open source information and source code you need to build an Android-compatible device.

• **JDK 6 Update 21**: Get the Java Platform, Standard Edition.

• **Eclipse**: Obtain the latest Eclipse IDE.

• **IBM product evaluation versions**: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

**Discuss**

• **XML zone discussion forums**: Participate in any of several XML-related discussions.

• **developerWorks blogs**: Check out these blogs and get involved.

**About the author**

C. Enrique Ortiz

C. Enrique Ortiz is a long-time mobile technologist, developer and author. He blogs at the About Mobility weblog and is the founder of the Austin chapter of MobileMonday.

**Trademarks**

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of IBM trademarks.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered