PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
MECOLAB

Andrés Matte Vallejos
matte.andres@gmail.com
1er semestre del 2015

Tutorial Sync Adapter

Índice

1. Introducción

Mantener una aplicación móvil no sincronizada con la web la vuelve prácticamente inservible. Hoy en día es difícil pensar en una aplicación importante que no descargue o suba datos a algún servidor. Claramente este feature hace que una aplicación luzca más profesional y da una mejor experiencia de usuario. Por ejemplo a un usuario de Facebook le gustaría que un cambio de foto de perfil no solo se realice en la web, si no que también ocurra en todos sus dispositivos móviles.

Hay muchas formas de realizar una sincronización de datos con un servidor. Si bien cada desarrollador puede implementar su propio sistema, no hay necesidad de reinventar la rueda. O en este caso, para qué vamos a reinventar algo mucho más complejo que una rueda: un Sync Adapter. Este framework ayuda a manejar y automatizar las transferencias de datos.

1.1. Beneficios

1. Ejecución automatizada: la sincronización estará definida de acuerdo a tu configuración y se ejecutará automáticamente de acuerdo a eso. Con esto podemos eliminar el botón de refresh.
2. Revisión de conectividad: por ejemplo, si en algún momento no hay conexión a internet y corresponde una sincronización el framework se encargará de realizar la sincronización nuevamente cuando la haya.
3. Encolación de transferencias fallidas: si se está descargando un archivo y esto falla, se vuelve a intentar.
4. Gasta menos batería: muchas veces el framework realizará la sincronización de más de una aplicación al mismo tiempo. De esta forma la radio se enciende con menor frecuencia.
5. Centralización de la transferencia: la sincronización de los datos se realiza toda al mismo tiempo y en el mismo lugar.
6. Manejo de cuenta y autenticación: si el usuario de tu aplicación requiere credenciales especiales se puede integrar el manejo de cuentas y autenticación en la transferencia.

1.2. Qué haremos

Durante el siguiente tutorial se utilizará una aplicación sencilla que liste a los integrantes de un curso. Se podrá añadir estudiantes, tanto en el servidor web como en la aplicación android, y se deberán mantener sincronizados los datos a través de un sync adapter. No se entrará en temas que escapan del objetivo del tutorial como el servidor, las vistas o sobre como se realizan los requests. De todas formas en `https://github.com/aamatte/ListaAlumnosAndroid` se puede revisar el código de la aplicación utilizada para los ejemplos[1].

1.3. Qué se necesita

Para que el Sync Adapter esté funcionando necesitamos implementar seis componentes, donde cada una juega un papel importante:

1. Base de datos
2. Content Provider
3. Authenticator
4. Clase Sync Adapter
5. Bound Service

[1] Algunos detalles en el código pueden variar en el tiempo pero los conceptos serán los mismos

2. Base de datos

En android se pueden guardar información de diferentes maneras, pero en este caso utilizaremos bases de datos. Usaremos SQLite, el sistema por defecto para android. En esta parte del tutorial no se ahondará de sobremanera debido a que es un tema transversal y hay mucha documentación al respecto. Se necesitará lo siguiente:

1. Definir un esquema y sus constantes
2. Crear base de datos y tablas
3. Crear métodos CRUD

2.1. Definir un esquema y sus constantes

Una vez que el modelo relacional esté definido este se debe plasmar en una clase que llamaremos Database-Contract. Esta tendrá definidas como String los nombres de las tablas, de las columnas y algunas operaciones importantes. Gracias a esto podemos cambiar el nombre de, por ejemplo, una columna sin la necesidad de cambiar el resto de nuestro código.

```
// DatabaseContract.java
public final class DatabaseContract {

    public DatabaseContract() {
    }

    public static abstract class Students implements BaseColumns {
      // BaseColumns nos entrega las constantes _ID y _COUNT

        public static final String TABLE_NAME = "STUDENTS";
        public static final String COLUMN_NAME_STUDENT_NAMES = "names";
        public static final String COLUMN_NAME_FIRST_LASTNAME = "firstlastname";
        public static final String COLUMN_NAME_SECOND_LASTNAME = "secondlastname";
        public static final String COLUMN_ID_CLOUD = "idcloud";

        public static final String TEXT_TYPE = " TEXT";
        public static final String INTEGER_TYPE = " INTEGER";
        public static final String COMMA_SEP = ",";

        public static final String SQL_CREATE_STUDENTS_TABLE =
                "CREATE TABLE " + Students.TABLE_NAME + " (" +
                        Students._ID + " INTEGER PRIMARY KEY," +
                        Students.COLUMN_NAME_STUDENT_NAMES + TEXT_TYPE + COMMA_SEP +
                        Students.COLUMN_NAME_FIRST_LASTNAME + TEXT_TYPE + COMMA_SEP +
                        Students.COLUMN_NAME_SECOND_LASTNAME + TEXT_TYPE + COMMA_SEP+
                        Students.COLUMN_ID_CLOUD + INTEGER_TYPE +
                        " )";

        public static final String SQL_DELETE_STUDENTS =
                "DROP TABLE IF EXISTS " + Students.TABLE_NAME;
    }
}
```

2.2. Crear base de datos y tablas

Ahora utlizaremos la clase SQLiteOpenHelper, una API para poder interactuar con nuestra base de datos. Cuando utilizas esta clase el sistema realiza las operaciones de larga duración cuando tu lo decidas y no en el inicio de la aplicación. Lo único que se debe hacer es llamar a $getWritableDatabase()$ o a $getReadableDatabase()$.

Las instancias de SQLiteDatabase retornadas por estos métodos están especialmente configuradas para realizar las operaciones especificadas. Esto quiere decir que $getWritableDatabase()$ retornará una instancia en el que la escritura de datos será rápida en desmedro de la lectura. Con $getReadableDatabase()$ no se podrá escribir y tendrá una lectura más rápida. Además, al llamar a estos métodos se crea la base de datos en caso de no existir aún.

Otro método escencial es $onCreate(SQLiteDatabase)$. Ahí es donde se ejecuta el código SQL para crear las tablas de la base de datos. También están $onUpgrade(SQLiteDatabase)$ y $onDowngrade(SQLiteDatabase)$.

```
// StudentsDbHelper.java
// Documentacion:
// http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class StudentsDbHelper extends SQLiteOpenHelper {

    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "Students.db";

    private static StudentsDbHelper sInstance;

    private StudentsDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

   // Nos aseguramos de que solo haya una instancia para evitar errores.
   // Mas detalles:
   // http://www.androiddesignpatterns.com/2012/05/correctly-managing-your-sqlite-database.html
    public static synchronized StudentsDbHelper getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new StudentsDbHelper(context.getApplicationContext());
        }
        return sInstance;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DatabaseContract.Students.SQL_CREATE_STUDENTS_TABLE);
    }

   // Cambia la version del esquema en caso de haber modificaciones.
   // Por simplicidad asumimos que esto no va a pasar y tan solo se resetea la db.
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL(DatabaseContract.Students.SQL_DELETE_STUDENTS);
        onCreate(db);
```

```
    }
}
```

Para tener acceso a la base de datos se debe hacer lo siguiente:

```
StudentsDbHelper mDbHelper = StudentsDbHelper.newInstance();
SQLiteDatabase db = mDbHelper.getWritableDatabase();
```

O en el otro caso:

```
StudentsDbHelper mDbHelper = StudentsDbHelper.newInstance();
SQLiteDatabase db = mDbHelper.getReadableDatabase();
```

2.3. Crear métodos CRUD

2.3.1. Agregar datos

Para insertar datos se deben ingresar los datos del elemento a añadir en ContentValues y usar el método insert. El segundo argumento especifica la columna en que la base de datos puede puede insertar null si values (los datos a ingresar) está vacio. Si es que no quieres que se ingrese un record con valores vacios debes setearlo como null. Por ejemplo, si se quiere añadir un estudiante:

```
// Codigo para insertar nuevo estudiante

if (mDbHelper == null) {
    mDbHelper = StudentsDbHelper.getInstance(getActivity());
}
SQLiteDatabase db = mDbHelper.getWritableDatabase();

ContentValues values = new ContentValues();
values.put(DatabaseContract.Students.COLUMN_NAME_STUDENT_NAMES, student.getNames());
values.put(DatabaseContract.Students.COLUMN_NAME_FIRST_LASTNAME, student.getFirstLastname());
values.put(DatabaseContract.Students.COLUMN_NAME_SECOND_LASTNAME, student.getSecondLastname());
values.put(DatabaseContract.Students.COLUMN_ID_CLOUD, student.getIdCloud());

// Retorna la columna en la que fue insertado
long success = db.insert(
       DatabaseContract.Students.TABLE_NAME,
       null,
       values);

if (success >= 0) return true;
return false;
```

2.3.2. Consultar datos

Para esto se usa el método query(). Si no se quiere filtrar y solo obtener toda la información de la tabla se pueden dejar todos los valores como null excepto el nombre de la tabla. Para seleccionar todos los estudiantes:

```
// Codigo para seleccionar todos los estudiantes

if (mDbHelper == null) {
    mDbHelper = StudentsDbHelper.getInstance(getActivity());
```

```
}

// Selecciono las columnas que debe retornar de cada fila. Podria dejarse como null y me retorna
    todas.
String[] projection = {DatabaseContract.Students.COLUMN_NAME_STUDENT_NAMES,
                    DatabaseContract.Students.COLUMN_NAME_FIRST_LASTNAME,
                    DatabaseContract.Students.COLUMN_NAME_SECOND_LASTNAME,
                    DatabaseContract.Students.COLUMN_ID_CLOUD
                    };

// Se deja como null porque no se requiere filtrar. Por ejemplo, si se necesitara filtrar por
    primer apellido:
// String selection = DatabaseContract.Students.COLUMN_NAME_FIRST_LASTNAME + "= ?"
String selection = null;

// Que el primer apellido sea Perez:
// String[] selectionArgs = new String[]{ "Perez" };
// En este caso dejamos como null
String[] selectionArgs = null;

SQLiteDatabase db = mDbHelper.getReadableDatabase();

Cursor c = db.query(DatabaseContract.Students.TABLE_NAME, // Tabla
                 projection,                          // Columnas a retornar
                 selection,                           // Columnas de WHERE
                 selectionArgs,                       // Valores de WHERE
                 null,                                // Group by
                 null,                                // Filtro por columnas de grupos
                 DatabaseContract.Students.COLUMN_NAME_FIRST_LASTNAME +" ASC"); // Ordenados

ArrayList<Student> studentsInDb = new ArrayList<Student>();

c.moveToFirst();

if (c.getCount()<1){
    return false;
}

while (c.moveToNext()){
    String names =
        c.getString(c.getColumnIndexOrThrow(DatabaseContract.Students.COLUMN_NAME_STUDENT_NAMES))
            .toUpperCase();
    String firstLast =
        c.getString(c.getColumnIndexOrThrow(DatabaseContract.Students.COLUMN_NAME_FIRST_LASTNAME))
            .toUpperCase();
    String secondLast =
        c.getString(c.getColumnIndexOrThrow(DatabaseContract.Students.COLUMN_NAME_SECOND_LASTNAME))
           .toUpperCase();
    int idCloud = c.getInt(c.getColumnIndexOrThrow(DatabaseContract.Students.COLUMN_ID_CLOUD));

    studentsInDb.add(new Student(names, firstLast, secondLast, idCloud));
}
```

2.3.3. Borrar datos

Se utiliza el método delete.

2.3.4. Actualizar datos

Se utiliza el método update.

3. Content Provider

Un content provider es exactamente lo que su nombre dice, un proveedor de contenido o de datos. Ofrece un esquema estructurado para acceder, crear, borrar o actualizar los datos que se pongan a disposición a través de él. En definitiva es la forma en que tu aplicación ofrece sus datos para que puedan ser consumidos o editados por procesos externos a ella o por otras aplicaciones sin la necesidad de que esta esté abierta.

Se pueden usar incluso como una forma de abstraer la implementación de un sistema de datos complejo frente al resto de los desarrolladores de una misma aplicación. De esta forma el resto de los integrantes del equipo solo deben estar al tanto de los estándares de comunicación con el content provider y no necesitan preocuparse de detalles de la implementación. De todas formas su principal objetivo es compartir los datos con otros procesos.

En esta sección cubriremos los siguientes temas:

1. Funcionamiento general
2. Content URIs
3. Clase Contrato
4. Clase Content Provider
5. Cómo utilizar el Content Provider
6. Permisos

3.1. Funcionamiento general

Como se aprecia en la Figura 1 el flujo parte desde tu aplicación obteniendo una instancia de ContentResolver[2]. Luego, a través de esta se hace una consulta pasando como primer parámetro una URI que hace referencia a un conjunto de datos que ofrece un content provider específico. Es importante saber que pueden haber muchos content providers disponibles, cada uno con su conjunto de URIs (corresponde un URI para cada conjunto de datos ofrecido en el provider). Por ejemplo, android provee Contacts Provider[3] y Calendar Provider[4]. En la próxima sección se entrará más en detalles en el diseño de las URIs.

Nuestra instancia de ContentResolver decide a qué provider hacer la consulta basado en la URI. Luego el content provider determina a qué conjunto de datos debe afectar (también basado en la URI) e interactúa con la base de datos para realizar la operación que fue invocada.

[2]ContentResolver: http://developer.android.com/reference/android/content/ContentResolver.html
[3]Contacts provider: http://developer.android.com/guide/topics/providers/contacts-provider.html
[4]Calendar provider: http://developer.android.com/guide/topics/providers/calendar-provider.html

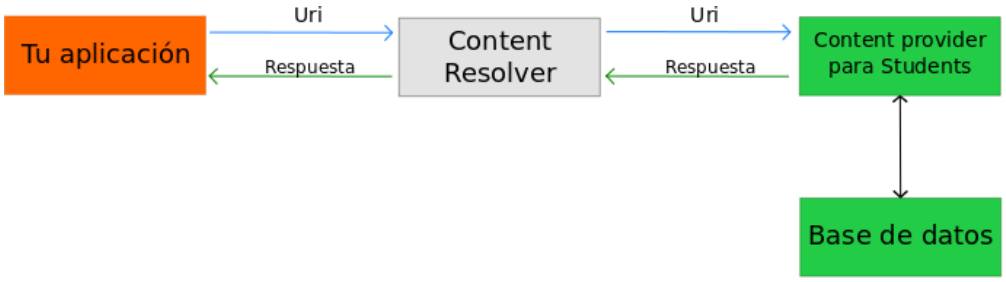

Figura 1: Flujo general de content provider

3.2. Content URIs

Un content URI es un identificador de un conjunto de datos en un provider. Está formado por el identificador del provider (authority) y por un nombre que indica a la tabla o al contenido que hace referencia. Cuando se llama a un método para acceder a algún conjunto de datos se debe incluir en los parámetros una URI determinada. Esto para que el ContentResolver pueda determinar a qué provider le corresponde manejar el llamado, y para que el provider pueda determinar sobre que datos realizar la operación.

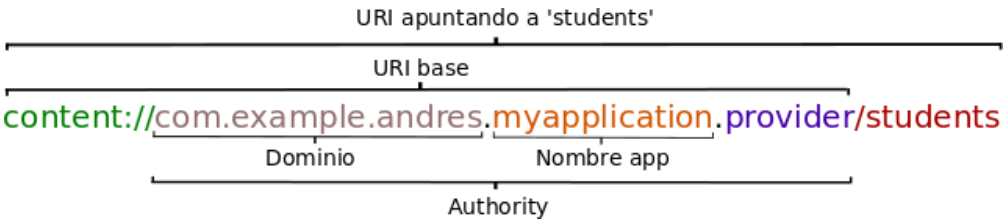

Figura 2: Estructura content URI

3.2.1. Authority

La authority es el identificador de tu aplicación dentro de android. Por lo general el formato de este viene dado por el package de la aplicación más 'provider'. Por ejemplo, en el caso de nuestra aplicación es *com.example.andres.myapplication.provider*. En el package (y en este caso, en la authority) se suele utilizar como primeras palabras un dominio de internet, al revés, al cual pertenece la aplicación. En este caso asumimos que hay un dominio con la ruta *andres.example.com*.

3.2.2. Estructura de rutas URI

Para el content provider se puede diseñar una estructura nueva de los datos de acuerdo a lo que se quiera ofrecer. En la clase contrato definiremos nuevas tablas (que pueden ser iguales a las de tu base de datos o no) que conformarán la estructura de rutas. En nuestro caso solo definiremos una tabla Students igual a la de nuestra base de datos. Claramente se pueden hacer estructuras más complejas con tablas anidadas y más esoterismos, pero no se cubrirán en este tutorial[5].

Para nuestra aplicación definiremos dos URIs, una para obtener la lista completa de estudiantes y otra para obtener a un estudiante en particular basandonos en su ID. La primera es exactamente como la que está en la Figura 2. La segunda debe poder soportar llamados del tipo

content://com.example.andres.myapplication.provider/students/3,

[5]Para más detalles mirar: `http://developer.android.com/guide/topics/providers/content-provider-creating.html#ContentURI`

donde 3 es el id del estudiante a seleccionar. Para poder soportar esto debemos definir la URI de la siguiente manera:

content://com.example.andres.myapplication.provider/students/#

3.3. Clase Contrato

La clase contrato es la que contiene todos los URIs, MIME Types y constantes necesarias para poder tener un protocolo de comunicación fijo entre los distintos procesos que utilizan el provider. Esta clase es la que debes compartir con otros desarrolladores si quieres que usen tu provider, ya que con ella les proveerás todos las valores necesarios para que se puedan comunicar con él de manera correcta. Básicamente, en la aplicación que utilice el provider se debe copiar y pegar la clase contrato.

A continuación se presenta el código de la clase contrato de nuestra aplicación de ejemplo. Los conceptos involucrados se explicarán luego.

```
import android.content.ContentResolver;
import android.net.Uri;
import android.provider.BaseColumns;

/**
 * Esta clase provee las constantes y URIs necesarias para trabajar con el StudentsProvider
 */
public final class StudentsContract {

    public static final String AUTHORITY = "com.example.andres.myapplication.provider";
    public static final Uri BASE_URI = Uri.parse("content://" + AUTHORITY);
    public static final Uri STUDENTS_URI = Uri.withAppendedPath(StudentsContract.BASE_URI,
         "/students");

    /*
        MIME Types
        Para listas se necesita 'vnd.android.cursor.dir/vnd.com.example.andres.provider.students
        Para items se necesita 'vnd.android.cursor.item/vnd.com.example.andres.provider.students'
        La primera parte viene esta definida en constantes de ContentResolver
     */
    public static final String URI_TYPE_STUDENT_DIR = ContentResolver.CURSOR_DIR_BASE_TYPE +
            "/vnd.com.example.andres.provider.students";

    public static final String URI_TYPE_STUDENT_ITEM = ContentResolver.CURSOR_ITEM_BASE_TYPE +
            "/vnd.com.example.andres.provider.students";

    /*
        Tabla definida en provider. Aca podria ser una distinta a la de la base de datos,
        pero consideramos la misma.
     */
    public static final class StudentsColumns implements BaseColumns{

        private StudentsColumns(){}

        public static final String NAMES = "names";
        public static final String FIRST_LASTNAME = "firstlastname";
        public static final String SECOND_LASTNAME = "secondlastname";
        public static final String ID_CLOUD = "idcloud";
```

```
        public static final String DEFAULT_SORT_ORDER = FIRST_LASTNAME + " ASC";

    }

}
```

3.3.1. MIME Types

De lo que está presente en la clase contrato y no se ha hablado es de los MIME Types. Estos conforman una manera estandar de clasificar los tipos de archivos o datos. Estos tipos son los mismos siempre, independiente del sistema operativo o de cualquier otra variante que se presente. Un MIME Type tiene dos partes: un tipo y un sub-tipo que están separados por un slash (/). Por ejemplo, las imagenes de formato JPEG tienen el MIME Type *image/jpeg*.

En el caso del provider se especifican dos MIME Types principales. El primero es para items individuales, donde el tipo viene dado por ***vnd.android.cursor.item*** y el subtipo (para nuestro provider) por */vnd.com.example.andres.provider.students*. El otro es para listas, donde el tipo viene dado por ***vnd.android.cursor.dir*** y el subtipo (para nuestro provider) por */vnd.com.example.andres.provider.students* (igual que para el de items individuales). De esta forma los MIME Types completos quedan como se especifica en el código.

Los MIME Types son importantes debido a que con ellas el desarrollador puede determinar el tipo de dato que se le retornará si utiliza una URI determinada en una consulta. No hay que olvidar que esta clase es principalmente un apoyo para que otros puedan utilizar los datos que les provees, por lo tanto se debe ser consistente y claro en la implementación.

3.4. Clase Content Provider

A continuación se creará una clase StudentProvider que herede de la clase abstracta ContentProvider. Esta obliga a implementar una serie de métodos especificados en el código que se mostrará a continuación. Por ejemplo, se pide implementar el método *query*. Este método tiene el mismo nombre que el que se usa desde ContentResolver. Es decir, si llamamos a ContentResolver.query(), el content resolver determinará a qué provider llamar basado en la URI y llamará al método query() de ese provider.

```
import android.content.ContentProvider;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.net.Uri;
import android.text.TextUtils;

/*
   Clase que extiende ContentProvider y que interactua con la base de datos
 */
public class StudentsProvider extends ContentProvider {
    public static final int STUDENT_LIST = 1;
    public static final int STUDENT_ID = 2;
    private static final UriMatcher sUriMatcher;
    static{
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        /*
            URI para todos los estudiantes.
```

```
        Se setea que cuando se pregunta a UriMatcher por la URI:
        content://com.example.andres.myapplication.provider/students
        se devuelva un entero con el valor de 1.
     */
    sUriMatcher.addURI(StudentsContract.AUTHORITY, "students", STUDENT_LIST);
    /*
        URI para un estudiante.
        Se setea que cuando se pregunta a UriMatcher por la URI:
        content://com.example.andres.myapplication.provider/students/#
        se devuelva un entero con el valor de 2.
     */
    sUriMatcher.addURI(StudentsContract.AUTHORITY, "students/#", STUDENT_ID);
}
/*
    Instancia de StudentsDbHelper para interactuar con la base de datos
 */
private StudentsDbHelper mDbHelper;

public StudentsProvider() { }

@Override
public boolean onCreate() {
    mDbHelper = StudentsDbHelper.getInstance(getContext());
    return true;
}

/*
    Llamado para borrar una o mas filas de una tabla
 */
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    SQLiteDatabase db = mDbHelper.getWritableDatabase();
    int rows = 0;
    switch (sUriMatcher.match(uri)) {
        case STUDENT_LIST:
            // Se borran todas las filas
            rows = db.delete(DatabaseContract.Students.TABLE_NAME, null, null);
            break;
        case STUDENT_ID:
            // Se borra la fila del ID seleccionado
            rows = db.delete(DatabaseContract.Students.TABLE_NAME, selection, selectionArgs);
    }
    // Se retorna el numero de filas eliminadas
    return rows;
}

/*
    Se determina el MIME Type del dato o conjunto de datos al que apunta la URI
 */
@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)){
        case STUDENT_LIST:
            return StudentsContract.URI_TYPE_STUDENT_DIR;
        case STUDENT_ID:
            return StudentsContract.URI_TYPE_STUDENT_ITEM;
```

```
        default:
            return null;
    }
}

/*
    Inserta nuevo estudiante
 */
@Override
public Uri insert(Uri uri, ContentValues values) {
    SQLiteDatabase db = mDbHelper.getWritableDatabase();
    db.insert(DatabaseContract.Students.TABLE_NAME, null, values);
    return null;
}

/*
    Retorna el o los datos que se le pida de acuerdo a la URI
 */
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {

    SQLiteDatabase db = mDbHelper.getReadableDatabase();
    switch (sUriMatcher.match(uri)){

        // Se pide la lista completa de estudiantes
        case STUDENT_LIST:
            // Si no hay un orden especificado,
            // lo ordenamos de manera ascendente de acuerdo a lo que diga el contrato
            if (sortOrder == null || TextUtils.isEmpty(sortOrder))
                sortOrder = StudentsContract.StudentsColumns.DEFAULT_SORT_ORDER;
            break;

        // Se pide un estudiante en particular
        case STUDENT_ID:
            // Se adjunta la ID del estudiante selecciondo en el filtro de la seleccion
            if (selection == null)
                selection = "";
            selection = selection + "_ID = " + uri.getLastPathSegment();
            break;

        // La URI que se recibe no esta definida
        default:
            throw new IllegalArgumentException(
                   "Unsupported URI: " + uri);
    }
    Cursor cursor = db.query(DatabaseContract.Students.TABLE_NAME,
                             projection,
                             selection,
                             selectionArgs,
                             null,
                             null,
                             sortOrder);
    // Se retorna un cursor sobre el cual se debe iterar para obtener los datos
    return cursor;
}
```

```
    @Override
    public int update(Uri uri, ContentValues values, String selection,
                      String[] selectionArgs) {
        // No se implemento un update
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

3.4.1. URI Matcher

Una clase importante que se utiliza en StudentProvider es UriMatcher. Esta clase se encarga de ayudarte a determinar que acción seguir para cada URI definida. Esto lo logra asociando cada URI a un entero que idealmente debes definir como constante. Además permite definir URIs genéricas, como se ve en el código para la elección de estudiantes por su ID.

3.5. Cómo utilizar el Content Provider

Una vez que se tiene todo lo anterior definido correctamente podemos hacer uso de nuestro ContentProvider. Para probar si funciona correctamente hay dos opciones: probarlo simplemente en tu aplicación o crear otra aplicación de prueba que lo utilice. Ahora el uso será muy sencillo, por ejemplo para seleccionar todos los estudiantes se debe hacer lo siguiente:

```
Cursor c = mContentResolver.query(StudentsContract.STUDENTS_URI, null, null, null, null);
c.moveToFirst();
while (c.moveToNext()){
   String names = c.getString(c.getColumnIndexOrThrow(StudentsContract.StudentsColumns.NAMES));
    String firstLast =
        c.getString(c.getColumnIndexOrThrow(StudentsContract.StudentsColumns.FIRST_LASTNAME));
    String secondLast =
        c.getString(c.getColumnIndexOrThrow(StudentsContract.StudentsColumns.SECOND_LASTNAME));
   // Hacer lo que se necesite con los datos
}
c.close();
```

Para seleccionar un estudiante cuyo ID = 1 se podría hacer lo siguiente:

```
Cursor c = mContentResolver.query(Uri.withAppendedPath(StudentsContract.STUDENTS_URI, "1"), null,
    null, null, null);
c.moveToFirst();
String names = c.getString(c.getColumnIndexOrThrow(StudentsContract.StudentsColumns.NAMES));
String firstLast =
    c.getString(c.getColumnIndexOrThrow(StudentsContract.StudentsColumns.FIRST_LASTNAME));
String secondLast =
    c.getString(c.getColumnIndexOrThrow(StudentsContract.StudentsColumns.SECOND_LASTNAME));
// Hacer lo que se necesite con los datos
c.close();
```

3.6. Permisos

Se deben setear ciertos permisos en el archivo AndroidManifest.xml presente en toda aplicación android. Para nuestra aplicación agregaremos solo lo fundamental, que es lo siguiente:

```
...
<provider
   android:name=".Provider.StudentsProvider"
   android:authorities="com.example.andres.myapplication.provider"
   android:enabled="true"  // Permite al sistema iniciar el provider
   android:exported="true"  // Permite que otras apps usen el provider
   android:syncable="true">   // Indica que los datos del provider son sincronizados con la nube
</provider>
...
```

Para más información sobre tipos de permisos más complejos se recomienda visitar `http://developer.android.com/guide/topics/providers/content-provider-creating.html#Permissions`.

4. Authenticator

Un authenticator es básicamente una componente que nos ofrece android para manejar las cuentas de nuestra aplicación de manera profesional, elegante y con una gran cobertura de los posibles casos que se puedan presentar. Su implementación, aunque sea *stub*, es obligatoria si se quiere implementar un sync adapter. Para esta parte supondremos que tenemos una API desarrollada que permite enviar un usuario y contraseña y retorna el token necesario para que podamos interactuar con ella.

4.1. Beneficios

En este caso estamos obligados a implementar un authenticator si es que queremos hacer un sync adapter. Pero si no estuvieramos obligados aún sería extremadamente útil implementar esta componente para manejar las cuentas. Se podría pensar que basta con hacer un log-in que utilice la API para obtener el token y guardar este en la base de datos. De esta forma nos ahorramos estudiar e implementar esta componente. La verdad es que eso puede funcionar pero deja una gran gama de casos posibles sin cubrir.

Imaginemos que el usuario cambia su contraseña en otro cliente y quiere que esto se vea reflejado en la aplicación. Con la implementación anteriormente indicada no podemos enterarnos de esto. O qué pasa si el usuario tiene su token expirado. Para darnos cuenta tendriamos que implementar nuestro propio sistema que maneje este caso. O si el usuario quiere que al loguearse en una aplicación se loguee automáticamente en todas las otras aplicaciones relacionadas (como las de Google). El authenticator maneja todos estos casos simplificando la tarea del desarrollador, por lo que su uso es absolutamente recomendable.

En resumen, podemos obtener los siguientes beneficios:

1. Forma estandar de autenticar a los usuarios.
2. Simplifica autenticación para el desarrollador.
3. Maneja casos de acceso denegados.
4. Puede manejar distintos tipos de tokens (que pueden otorgar distintos permisos).
5. Compartir cuentas entre aplicaciones.
6. Nos permite implementar un SyncAdapter.
7. Además, tu aplicación se mete en terreno de gigantes ;).

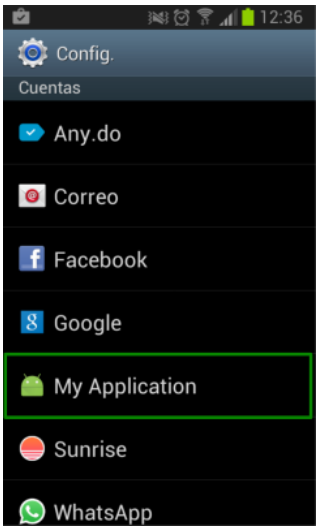

Figura 3: La cuenta relacionada con nuestra aplicación en la configuración del celular

4.2. Qué haremos

Para implementar este componente abordaremos los siguientes temas:

1. Definiciones básicas
2. Flujo general
3. Implementar el authenticator
4. Crear las actividades
5. Crear el authenticator service

4.3. Definiciones básicas

4.3.1. Token

Es una clave de acceso temporal que el servidor le entrega a un cliente. El usuario se identifica, por lo general con usuario y contraseña, y el servidor le retorna esta clave que debe adjuntar en todos los requests que haga. Puede ser limitado y expirar pasada cierta cantidad de tiempo.

4.3.2. AccountManager

Esta clase es como el maestro de la orquesta. Básicamente se encarga de la gestión de todas las cuentas en el dispositivo y sabe a quien llamar en cada caso que se presente. Esta clase la provee android por lo que no necesitamos implementarla.

4.3.3. AccountAuthenticator

Cada empresa o conjunto de aplicaciones tiene distintas maneras de autenticar a los usuarios, por lo tanto android ofrece AccountAuthenticator para personalizar este proceso. Cada grupo de aplicaciones, que también puede ser solo una, (por ejemplo Facebook, Whatsapp, o Google) tiene su propio AccountAuthenticator. Esta clase sabe que actividad mostrar para que el usuario ingrese sus credenciales y donde encontrar algún token retornado por el servidor previamente.

4.3.4. AccountAuthenticatorActivity

Actividad llamada por AccountAuthenticator para que el usuario ingrese a su cuenta o se registre. Esta debe interactuar con el servidor para obtener el token y retornarlo al AccountAuthenticator.

4.4. Flujo general

El flujo no es muy complejo. Primero se le dice a AccountManager que me dé el token de cierta cuenta. Luego, este le pregunta al AccountAuthenticator relevante si tiene algún token. En caso de no existir hace que se abra la actividad de registro/logueo, obtiene el token retornado por el servidor y se retorna al AccountManager. El token se guarda en AccountManager para uso futuro y este lo retorna al que lo pidió por primera vez a través de un callback.

A continuación se presenta un gráfico que estuvo alguna vez en la documentación de google y que puede ayudar a clarificar el flujo. Se irán explicando los conceptos principales a medida que vayamos avanzando en la implementación.

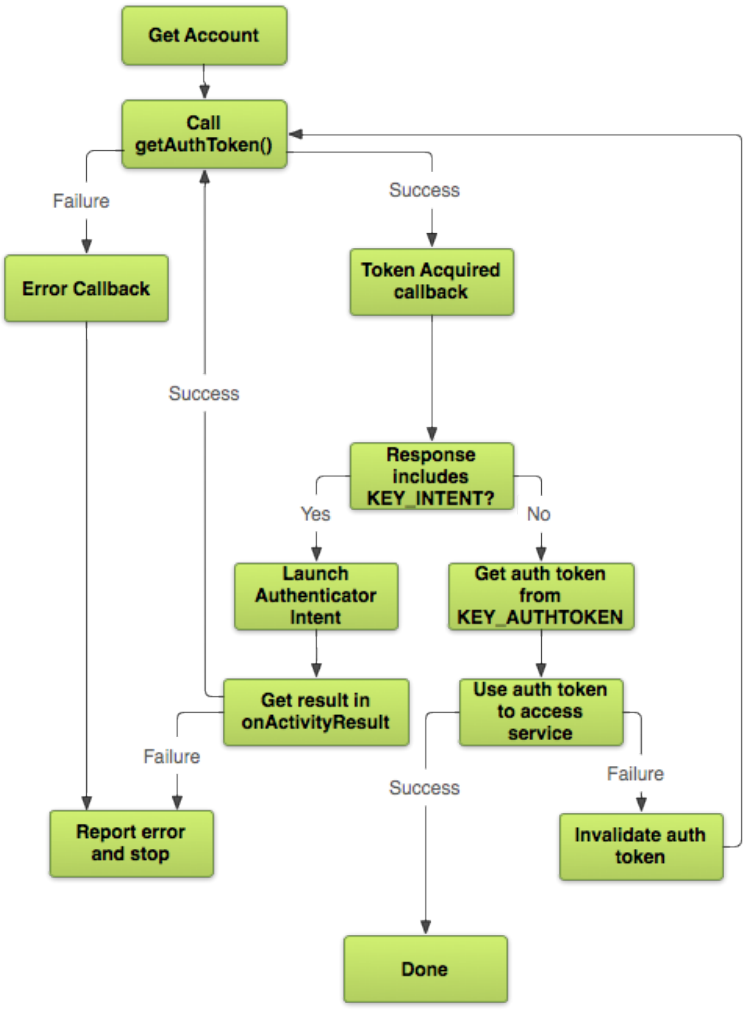

Figura 4: Flujo de log-in Authenticator

5. Clase Sync Adapter

6. Bound Service