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

Andrés Matte Vallejos
aamatte@uc.cl
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 SyncAdapter 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 dapter
5. Correr el SyncAdapter

[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.

2.1. Qué haremos

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

2.2. Definir un esquema y sus constantes

Una vez que el modelo relacional esté definido este se debe plasmar en una clase que llamaremos DatabaseContract. 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;
    }
}
```

2.3. 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. En cambio, con $getReadableDatabase()$ se 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();
```

2.4. Crear métodos CRUD

2.4.1. 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;
```

2.4.2. 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));
}
```

2.4.3. Borrar datos

Se utiliza el método delete.

2.4.4. 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 compartir los datos desde bases de datos o archivos, pero en este caso utilizaremos la primera.

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.

3.1. Qué haremos

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

3.2. 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.

[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

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.

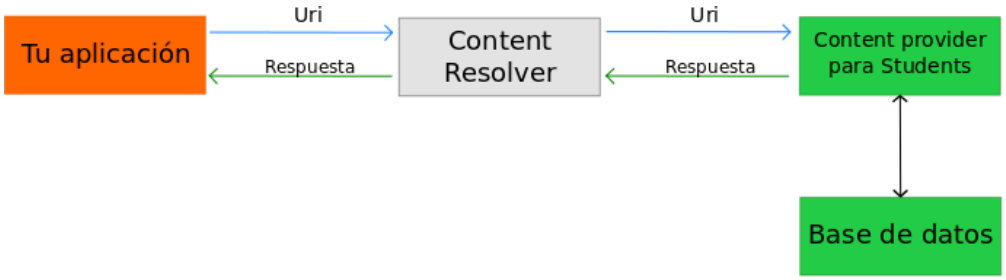

Figura 1: Flujo general de content provider

3.3. 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.

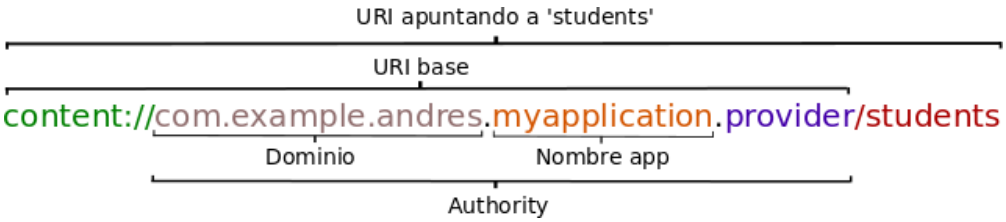

Figura 2: Estructura content URI

3.3.1. Authority

La authority es el nombre de tu provider. 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*.

3.3.2. 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].

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

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,

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/#

3.4. 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";

    }

}
```

3.4.1. 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.

3.5. 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);

    // Le avisa a los observadores
    getContext().getContentResolver().notifyChange(uri, null);

    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");
    }
}
```

3.5.1. 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.

3.6. 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();
```

3.7. 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 ;).

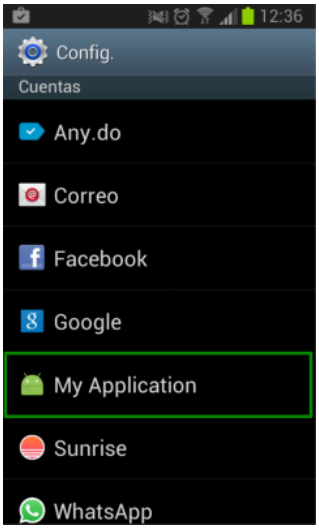

Figura 3: La cuenta relacionada con nuestra aplicación en la configuración del celular

4.2. Qué haremos

Para implementar este componente abordaremos los siguientes temas:

1. Definiciones básicas
2. Flujo general
3. Crear authenticator
4. Crear la actividad de log-in
5. Crear el authenticator service
6. Cómo usarlo
7. Permisos

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 para uso futuro y se 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.

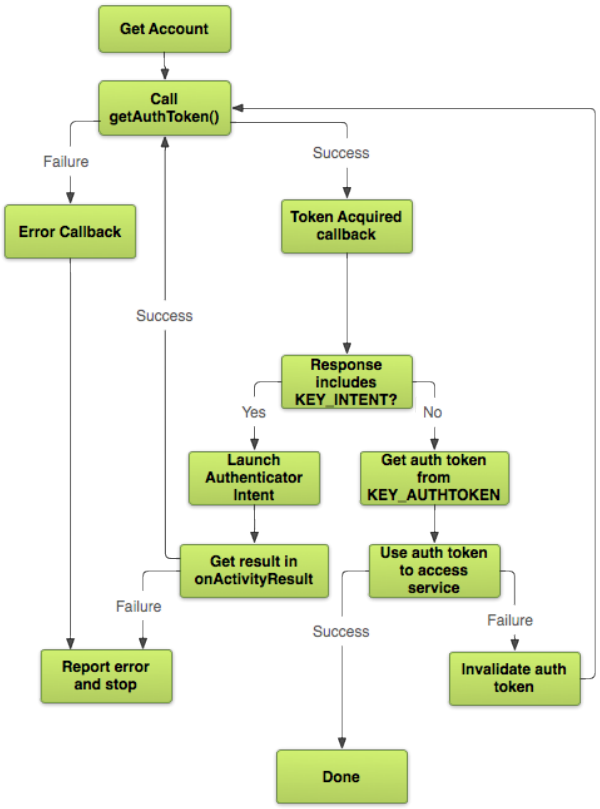

Figura 4: Flujo de log-in Authenticator

4.5. Crear Authenticator

El AccountAuthenticator es el encargado de realizar todas las operaciones importantes relacionadas con la cuenta: obtener el token, mostrar pantalla de logueo y comunicarse con el servidor. Para crear un nuestro propio AccountAuthenticator tenemos que extender la clase abstracta AbstractAccountAuthenticator e implementar algunos métodos, donde los más importantes son addAccount y getAuthToken.

Para implementar los métodos del authenticator se debe seguir un patrón relativamente estandar. Para cada uno de los métodos se debe devolver una de las siguientes opciones:

- Si los argumentos que se entregan al método son suficientes para llevar a cabo la operación entonces se debe realizar la acción y devolver un Bundle con los resultados.
- Si el authenticator necesita información del usuario y este no la tiene, entonces se creará un Intent para iniciar una actividad que le pedirá los datos al usuario. Este Intent tiene que ser retornado en un Bundle con el key KEY_INTENT. Como la actividad tiene que devolver al authenticator el resultado de la petición de datos, entonces se debe incluir el AccountAuthenticatorResponse en el intent con el key KEY_ACCOUNT_MANAGER_RESPONSE. Luego, la actividad tiene que llamar onResult(Bundle) si tuvo éxito, o onError(int, String) si hubo algún error. Esto se evita al hacer que la actividad extienda de AccountAuthenticatorActivity.
- En caso de haber algún error se debe retornar un Bundle que contenga un par de valores con un código de error como key (int) y con un String que explique el error como valor.

En el caso nuestro, haremos una implementación relativamente sencilla de estos métodos. El esqueleto es como sigue:

```
import android.accounts.AbstractAccountAuthenticator;
import android.accounts.Account;
import android.accounts.AccountAuthenticatorResponse;
import android.accounts.AccountManager;
import android.accounts.NetworkErrorException;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;

import com.example.andres.myapplication.Activities.AuthenticatorActivity;

/*
    Clase que maneja la autenticacion y realiza la gran mayoria de las operaciones importantes de
        una cuenta.
 */
public class AccountAuthenticator extends AbstractAccountAuthenticator {

    private Context mContext;

    public AccountAuthenticator(Context context) {
        super(context);
        mContext = context;
    }

    @Override
    public Bundle editProperties(AccountAuthenticatorResponse response, String accountType) {
        return null;
    }

    /*
      Llamado cuando el usuario quiere loguearse y anadir un nuevo usuario.
      @return bundle con intent para iniciar AuthenticatorActivity.
     */
    @Override
    public Bundle addAccount(AccountAuthenticatorResponse response, String accountType, String
        authTokenType, String[] requiredFeatures, Bundle options) throws NetworkErrorException {

      ...

    }

    @Override
    public Bundle confirmCredentials(AccountAuthenticatorResponse response, Account account,
        Bundle options) throws NetworkErrorException {
        return null;
    }

    /*
        Obtiene el token de una cuenta. Si falla, se avisa que se debe llamar a
            AuthenticatorActivity.
        @return Si resulta, bundle con informacion de cuenta y token.
                Si falla, bundle con informacion de cuenta y activity.
```

```
     */
    @Override
    public Bundle getAuthToken(AccountAuthenticatorResponse response, Account account, String
         authTokenType, Bundle options) throws NetworkErrorException {

      ...

    }

    @Override
    public String getAuthTokenLabel(String authTokenType) {
        return null;
    }

    @Override
    public Bundle updateCredentials(AccountAuthenticatorResponse response, Account account, String
         authTokenType, Bundle options) throws NetworkErrorException {
        return null;
    }

    @Override
    public Bundle hasFeatures(AccountAuthenticatorResponse response, Account account, String[]
         features) throws NetworkErrorException {
        return null;
    }
}
```

4.5.1. addAcount

Este método se llama cuando un usuario se loguea y queremos añadir una nueva cuenta al dispositivo. Se puede llamar a través de la aplicación, para lo cual necesitamos algunos permisos que se detallarán después. También es el método que se llama cuando se ingresa a configuración, apretamos Agregar Cuenta y seleccionamos nuestro Authenticator.

Figura 5: Agregar cuenta en configuraciones

```
    /*
      Llamado cuando el usuario quiere loguearse y anadir un nuevo usuario.
```

```
     @return bundle con intent para iniciar AuthenticatorActivity.
     */
    @Override
    public Bundle addAccount(AccountAuthenticatorResponse response, String accountType, String
        authTokenType, String[] requiredFeatures, Bundle options) throws NetworkErrorException {

        final Intent intent = new Intent(mContext, AuthenticatorActivity.class);
        intent.putExtra(AuthenticatorActivity.ARG_ACCOUNT_TYPE, accountType);
        intent.putExtra(AuthenticatorActivity.ARG_AUTH_TYPE, authTokenType);
        intent.putExtra(AuthenticatorActivity.ARG_IS_ADDING_NEW_ACCOUNT, true);
        intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE, response);

        final Bundle bundle = new Bundle();
        bundle.putParcelable(AccountManager.KEY_INTENT, intent);

        return bundle;
    }
```

4.5.2. getAuthToken

Es el método descrito en la Figura 4. Obtiene un token guardado de algún log-in anterior. Si no existe aún, el usuario deberá loguearse. Para lograr eso tenemos que llamar al método peekAuthToken() de AccountManager. Si no hay un token retornamos lo mismo que para addAcount. De esta forma se lanzará la actividad para que el usuario se loguee.

```
    /*
        Obtiene el token de una cuenta. Si falla, se avisa que se debe llamar a
            AuthenticatorActivity.
        @return Si resulta, bundle con informacion de cuenta y token.
                Si falla, bundle con informacion de cuenta y activity.
     */
    @Override
    public Bundle getAuthToken(AccountAuthenticatorResponse response, Account account, String
        authTokenType, Bundle options) throws NetworkErrorException {

        // Extrae username y pass del account manager
        final AccountManager am = AccountManager.get(mContext);

        // Pide el authtoken
        String authToken = am.peekAuthToken(account, authTokenType);

        // Si authToken esta vacio (no hay token guardado), se intenta autenticar en servidor
        if (TextUtils.isEmpty(authToken)){
            final String password = am.getPassword(account);
            if (password != null) {
                // Se autentica en el servidor
                authToken = authenticateInServer(account);
            }
        }
        // Si obtenemos un authToken, lo retornamos
        if (!TextUtils.isEmpty(authToken)) {

            final Bundle result = new Bundle();
            result.putString(AccountManager.KEY_ACCOUNT_NAME, account.name);
            result.putString(AccountManager.KEY_ACCOUNT_TYPE, account.type);
```

```
            result.putString(AccountManager.KEY_AUTHTOKEN, authToken);

            return result;
        }
        // Si llegamos aca aun no podemos obtener el token.
        // Necesitamos pedirle de nuevo que se loguee.
        final Intent intent = new Intent(mContext, AuthenticatorActivity.class);
        intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE, response);
        intent.putExtra(AuthenticatorActivity.ARG_ACCOUNT_TYPE, account.type);
        intent.putExtra(AuthenticatorActivity.ARG_AUTH_TYPE, authTokenType);

        final Bundle bundle = new Bundle();
        bundle.putParcelable(AccountManager.KEY_INTENT, intent);
        return bundle;

    }
```

Si por alguna razón el token que se tiene ya no es válido, este se debe invalidar con el método invalidateAuthToken() de AccountManager. Luego, se debe pedir al usuario que se loguee nuevamente para que obtenga un token válido.

4.6. Crear la actividad de log-in

En esta actividad el usuario ingresará sus credenciales y obtendrá el token que debemos guardar para enviarlo al authenticator asociado. Un punto importante es que esta actividad va a extender de AccountAuthenticatorActivity. Al extender de esta clase se obtiene el método setAccountAuthenticatorResult(). Además, al extender de esta clase se sobreescribe el método finish() de la actividad, que es llamado siempre cuando la actividad termina. Ahora este método llamará automáticamente a onResult() si es que se invocó a setAuthenticatorResult() o a onError() si es que no se invocó (ver últimas lineas del código de finishLogin() más abajo).

Se creó un método submit() que se llama al apretar el botón de log-in.

```
public void submit() {

    // Se obtiene el usuario y contrasena ingresados
    final String userName = ((TextView) findViewById(R.id.account_name)).getText().toString();
    final String userPass = ((TextView) findViewById(R.id.account_password)).getText().toString();

    // Se loguea de forma asincronica para no entorpecer el UI thread
    new AsyncTask<Void, Void, Intent>() {
        @Override
        protected Intent doInBackground(Void... params) {

            // Se loguea en el servidor y retorna token
            String authtoken = logIn(userName, userPass);

            // Informacion necesaria para enviar al authenticator
            final Intent res = new Intent();
            res.putExtra(AccountManager.KEY_ACCOUNT_NAME, userName);
            res.putExtra(AccountManager.KEY_ACCOUNT_TYPE, "com.example.andres.myapplication");
            res.putExtra(AccountManager.KEY_AUTHTOKEN, authtoken);
```

```
            res.putExtra(PARAM_USER_PASS, userPass);

            return res;
        }
        @Override
        protected void onPostExecute(Intent intent) {

            finishLogin(intent);
        }
    }.execute();
}
```

También se crea el método finishLogin(), llamado al finalizar el método submit(). Este se encarga de crear la cuenta nueva si es que no existe y de enviar la información al authenticator.

```
private void finishLogin(Intent intent) {

    String accountName = intent.getStringExtra(AccountManager.KEY_ACCOUNT_NAME);
    String accountPassword = intent.getStringExtra(PARAM_USER_PASS);
    final Account account = new Account(accountName,
        intent.getStringExtra(AccountManager.KEY_ACCOUNT_TYPE));

    // Si es que se esta anadiendo una nueva cuenta
    if (getIntent().getBooleanExtra(ARG_IS_ADDING_NEW_ACCOUNT, false)) {

        String authtoken = intent.getStringExtra(AccountManager.KEY_AUTHTOKEN);
        // Pueden haber muchos tipos de cuenta. En este caso solo tenemos una que llame 'normal'
        String authtokenType = "normal";
        // Creando cuenta en el dispositivo y seteando el token que obtuvimos.
        mAccountManager.addAccountExplicitly(account, accountPassword, null);

        // Ojo: hay que setear el token explicitamente si la cuenta no existe, no basta con
            mandarlo al authenticator
        mAccountManager.setAuthToken(account, authtokenType, authtoken);
    }

    // Si no se esta anadiendo cuenta, el token antiguo estaba invalidado.
    // Seteamos contrasena nueva por si la cambio.
    else {
        // Solo seteamos contrasena
        // Aca no es necesario setear el token explicitamente, basta con enviarlo al Authenticator
        mAccountManager.setPassword(account, accountPassword);
    }
    // Setea el resultado para que lo reciba el Authenticator
    setAccountAuthenticatorResult(intent.getExtras());
    setResult(RESULT_OK, intent);

    // Cerramos la actividad
    finish();
}
```

4.7. Crear el authenticator service

Ahora tenemos que hacer que nuestro Authenticator esté disponible para todas las apps que quieran utilizarlo, como por ejemplo las configuraciones del teléfono. También necesitamos que obviamente esté corriendo sin que tengamos la aplicación abierta, por lo tanto lo más sensato es usar Servicios.

4.7.1. Servicios

Un servicio es un componente de una aplicación hecho para correr operaciones de larga duración en el background. No posee interfaz gráfica y necesita que otra componente de la aplicación la inicie. Luego, y pese a que la aplicación que la inició se cierre, el servicio seguirá corriendo hasta que termine con su tarea. Puede usarse para tocar música, para interactuar con un content provider o para realizar operaciones con un servidor.

En este caso usaremos un servicio Bound. Este tipo de servicios dan la opción de enlazarse con una componente de la aplicación a través del método bindService(). Este tipo de servicios ofrece una interfaz cliente-servidor que permite a los componentes interactuar con el servicio, enviar requests, obtener resultados, etc. Un bound service corre solo mientras la componente de la aplicación esté enlazado a el.

Para crear un bound service tenemos que implementar onBind(), un callback que retorna un IBinder. Este objeto define la interfaz de comunicación con el servicio. La implementación que haremos será muy sencilla, por lo que si quieres averiguar más sobre servicios puedes mirar la documentación[6] .

```
public class AuthenticatorService extends Service {
    private AccountAuthenticator mAuthenticator;

    @Override
    public void onCreate() {
        // Create a new authenticator object
        mAuthenticator = new AccountAuthenticator(this);
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mAuthenticator.getIBinder();
    }
}
```

4.8. XML y manifest

En el manifest debemos agregar nuestro servicio:

```
<service android:name=".Services.AuthenticatorService">
   <intent-filter>
        <action android:name="android.accounts.AccountAuthenticator" />
   </intent-filter>
   <meta-data android:name="android.accounts.AccountAuthenticator"
              android:resource="@xml/authenticator" />
</service>
```

[6] http://developer.android.com/guide/components/services.html#CreatingBoundService

Ahora, si es que aún no tienes una carpeta con nombre *xml* dentro de *res* creala. Dentro de ella crearemos un archivo xml con el nombre authenticator.xml. Este archivo nos permitirá definir algunos atributos.

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="com.example.andres.myapplication"
    android:icon="@drawable/ic_launcher"
    android:smallIcon="@drawable/ic_launcher"
    android:label="@string/app_name"
/>
```

Algunos de los atributos importantes que se pueden poner en el archivo son:

- accountType: nombre para identificar nuestro tipo de cuenta. Cuando alguna aplicación quiera autenticarse con nuestra cuenta debe usar ese accountType.
- icon y smallIcon: iconos a ser mostrados en la configuración del celular, en la zona de cuentas.
- label: nombre de nuestro tipo de cuenta en la configuración del celular.
- accountPreferences: también se puede definir un archivo de preferencias que se desplegará al apretar nuestra cuenta en configuraciones del celular. En este caso no lo definimos.

4.9. Cómo usarlo

El uso más básico es preguntarle a AccountManager por las cuentas del tipo que nos interesa y elegir la primera, si es que sabemos que solo habrá una. En caso de poder haber más de una (como las cuentas de Google) sería conveniente poner esas cuentas en una lista y que el usuario decidiera cual utilizar. En la actividad principal de la aplicación de ejemplo se utiliza el siguiente código:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   ...
    accountManager = (AccountManager) getSystemService(ACCOUNT_SERVICE);

    Account[] accounts = accountManager.getAccountsByType(ACCOUNT_TYPE);
    if (accounts.length == 0){
        // Tambien se puede llamar a metodo accountManager.addAcount(...)
        Intent intent = new Intent(this, AuthenticatorActivity.class);
        intent.putExtra(AuthenticatorActivity.ARG_IS_ADDING_NEW_ACCOUNT, true);
        startActivity(intent);
    }
    else{
        mAccount = accounts[0];
        accountManager.getAuthToken(mAccount, "normal", null, this, mGetAuthTokenCallback, null);
       ...
    }
    ....
}
```

El código de *mGetAuthTokenCallback* es:

```
private AccountManagerCallback<Bundle> mGetAuthTokenCallback =
      new AccountManagerCallback<Bundle>() {
          @Override
          public void run(final AccountManagerFuture<Bundle> arg0) {
                try {
                   token = (String) arg0.getResult().get(AccountManager.KEY_AUTHTOKEN);
                } catch (Exception e) {
                   // handle error
                }
          }
        };
```

5. Clase SyncAdapter

El SyncAdapter es la componente de la aplicación que encapsulará el código para transferir datos entre el dispositivo y un servidor. Basado en la configuración que le des el SyncAdapter ejecutará el código de sincronización. Para añadir dejar el SyncAdapter funcionando (al fin) necesitamos los siguientes componentes:

1. Clase SyncAdapter: contiene tu código de sincronización en una interface compatible con el framework sync adapter.
2. Bound service: permite al framework correr el código en el tu clase sync adapter.
3. Archivo de metadata xml: contiene información de tu sync adapter. El framework lee esta información para saber como calendarizar y cargar tu transferencia de datos.
4. Declaraciones en manifest: declara el bound service y apunta a datos específicos de la metadata del sync adapter xml.

5.1. Clase SyncAdapter

Debemos crear una nueva clase que extienda de AbstractThreadedSyncAdapter, la clase base del sync adapter. También debemos definir constructores para la clase e implementar el método donde se definen las tareas de sincronización.

5.1.1. Extender AbstractThreadedSyncAdapter

En los constructores de la clase que extienda a AbstractThreadedSyncAdapter (en este caso la llamaremos SyncAdapter) debemos hacer un proceso muy similar a la configuración que se hace cuando se está creando una actividad. Se debe hacer el setup del sync adapter, al igual que en el Activity.onCreate(). Por ejemplo, nosotros utilizaremos el ContentProvider para hacer la sincronización de datos, por lo que es conveniente obtener una referencia a ContentResolver en el constructor. Por ejemplo en nuestro caso quedaría como sigue:

```
public class SyncAdapter extends AbstractThreadedSyncAdapter {
    private ContentResolver mContentResolver;
    private String students;
    private String token;
    private AccountManager mAccountManager;
   ...
```

```
    public SyncAdapter (Context context, boolean autoInitialize){
        super(context, autoInitialize);
        this.mContentResolver = context.getContentResolver();
        mAccountManager = AccountManager.get(context);

    }

   ...
}
```

5.1.2. Implementar onPerformSync

El método onPerformSync es el método que el framework llama para hacer la sincronización de datos. Por lo tanto acá debes implementar toda la lógica de tu sincronización. Los parámetros de onPerformSync son:

- Account: cuenta asociada a la sincronización.
- Extras: Bundle que contiene flags con información mandada el evento que desencadenó la sincronización.
- Authority: la authority del content provider que implementamos.
- ContentProviderClient: es una clase con algunas funcionalidades de ContentResolver y que nos permite interactuar con el ContentProvider identificado por el authority que viene en los argumentos. Si no se quiere utilizar se puede ignorar.
- SyncResult: objeto para mandar información al sync adapter framework.

El código para sincronizar datos que utilizamos en la app de ejemplo no tiene mucha importancia, pero es el siguiente:

```
@Override
public void onPerformSync(Account account, Bundle extras, String authority, ContentProviderClient
    provider, SyncResult syncResult) {
       try {
           // No importa que el thread se bloquee ya que es asincronico.
           token = mAccountManager.blockingGetAuthToken(account, "normal" , true);
        // Obtiene datos del servidor
           ArrayList<String> results = performRequest(url, "GET");

           // Sincroniza los datos
           updateData(results, "GET");

       // Manejo de errores
       } catch (IOException e) {
           e.printStackTrace();
       } catch ...

       ...
}
```

Dentro de este método debes hacer las siguientes cosas:

- Conectarse al servidor
- Descargar y subir datos

- Manejar conflictos y determinar como sincronizar los datos
- Cerrar conexiones y limpiar archivos temporales y caché

5.2. Bound Service

Ahora debemos enlazar nuestro SyncAdapter al framework sync adapter para que este tenga acceso a nuestro código. Para eso usaremos un bound service, al igual que en el authenticator. Este servicio pasará un Binder especial al framework. Con este Binder el framework puede invocar a nuestro onPerformSync(). El código sería como sigue:

```
public class SyncService extends Service {
    private static final Object sSyncAdapterLock = new Object();
    private static SyncAdapter sSyncAdapter = null;

    @Override
    public void onCreate() {
        super.onCreate();

        synchronized (sSyncAdapterLock) {
            if (sSyncAdapter == null) {
                sSyncAdapter = new SyncAdapter(getApplicationContext(), true);
            }
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return sSyncAdapter.getSyncAdapterBinder();
    }

    ...

}
```

5.3. Agregar SyncAdapter metadata

Ahora debemos poner a disposición del framework la metadata necesaria que describe el componente y que provee algunos flags extras. Se especifica el tipo de cuenta que creaste para el tu SyncAdapter, declara un authority del content provider asociado con la app, controla una parte de la interfaz de usuario relacionada con el sync adapter y declara otros flags.

Debemos crear nuestro archivo syncadapter.xml en /res/xml. El contenido es como sigue:

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="com.example.andres.myapplication.provider"
    android:accountType="com.example.andres.myapplication"
    android:userVisible="true"        // dice que es visible en configuraciones del telefono
    android:allowParallelSyncs="false"  // multiples instancias de sync adapter no pueden correr
    android:isAlwaysSyncable="true"   // puede sincronizar en cualquier momento que hayas dicho
    android:supportsUploading="true"/>
```

5.4. Declararlo en el manifest

Tenemos que declarar el servicio que creamos y agregar los permisos necesarios para que el sync adapter pueda correr adecuadamente. Los permisos necesarios son los siguientes:

- android.permission.INTERNET
- android.permission.READ_SYNC_SETTINGS: permite leer el estado de la configuración del sync adapter.
- android.permission.WRITE_SYNC_SETTINGS: permite a tu app controlar la configuración del sync adapter.
- android.permission.AUTHENTICATE_ACCOUNTS: ya deberias tener esta desde que creamos el Authenticator.

Además, el servicio queda declarado como sigue:

```
<service
   android:name=".Services.SyncService"
   android:exported="true"  // permite a procesos externos a tu app acceder al servicio
   android:process=":sync"> // le dice al sistema que corra el servicio en un proceso global
       llamado sync
   <intent-filter>
      <action android:name="android.content.SyncAdapter"/>
   </intent-filter>
   <meta-data android:name="android.content.SyncAdapter"
           android:resource="@xml/syncadapter" />
</service>
```

6. Corriendo el SyncAdapter

Debes tratar de correr el SyncAdapter basado en un calendario o como resultado indirecto de algún evento. Por ejemplo, puedes realizar la sincronización en un momento particular del día, o cuando hayan cambios en los datos almacenados en el dispositivo. Debes evitar correr la sincronización por una acción directa del usuario porque haciendo esto no aprovechas todos los beneficios de una sincronización calendarizada.

Para realizar la sincronización tienes varias alternativas basadas en distintos eventos.

6.1. Datos del servidor cambian

Cambiar los datos en respuesta a un mensaje desde el servidor indicando que los datos han cambiado. Esta opción te permite actualizar los datos sin utilizar mucha batería y mejora bastante el desempeño. No cubriremos en detalle ya que requiere utilizar Google Cloud Messaging (GCM), tema que da para un tutorial por si solo.

6.2. Datos del content provider cambian

Para realizar esto debes registrar un observer para el content provider. Cuando los datos en el content provider cambia, el content provider framework llama al observer. En el observer se debe llamar a requestSync() para decirle al framework que corra el sync adapter. También es muy importante que cuando se quiera hacer la sincronización se debe llamar a ContentResolver.notifyChange(uri, null). Esto le avisa a los observadores que se ha hecho un cambio y ellos toman las acciones especificadas. En este caso se usa en el método insert() de la clase Content Provider.

Para crear el observer debes extender la clase ContentObserver e implementar el método onChange(). Dentro de este método debes llamar a requestSync() para iniciar el SyncAdapter. Para registrar al observer hay que pasar esta clase como argumento de registerContentObserver(). También debes pasar un content URI que indica los datos que quieres observar. La clase observer quedaría como sigue:

```
    public class TableObserver extends ContentObserver {
        /**
         * Crea un content observer
         *
         */
        public TableObserver(Handler handler) {
            super(handler);
        }

        /*
         * Define el metodo que es llamado cuando los datos en el content provider cambian.
         * Este metodo es solo para que haya compatibilidad con plataformas mas viejas.
        */
        @Override
        public void onChange(boolean selfChange) {
            onChange(selfChange, null);
        }
        /*
         * Define el metodo que es llamado cuando los datos en el content provider cambian.
         */
        @Override
        public void onChange(boolean selfChange, Uri changeUri) {

            if (mAccount != null) {
                // Corre la sincronizacion
                ContentResolver.requestSync(mAccount, StudentsContract.AUTHORITY, null);
            }
        }
    }
```

Luego para registrar el observer tenemos que hacer lo siguiente:

```
// En activity.onCreate()
...
TableObserver observer = new TableObserver(null);
/*
 * Registra el obsever para students
 */
mResolver.registerContentObserver(StudentsContract.STUDENTS_URI, true, observer);
...
```

6.3. Hay un Network Message

Cuando una conexión de red está disponible el sistema manda unos mensajes para mantener la conexión TCP/IP abierta. Este mensaje también va al ContentResolver. Utilizando el método setSyncAutomatically() la sincronización se realiza siempre que ContentResolver recibe el mensaje. Utilizando este forma de sincronzar los datos te aseguras de que la sincronización se realice siempre que haya una conexión.

Usa esta opción si necesitas que los datos estén regularmente actualizados, pero no es completamente necesario que se actualicen apenas los datos del content provider cambian. También es buena opción si no necesitas correr la sincronización con un calendario definido. Se puede usar setSyncAutomatically() y addPeriodicSync() (la forma de sincronizar de la siguiente sección) en conjunto.

Es importante destacar que esto solo funcionará si se cumple con que:

1. En la declaración del provider en AndroidManifest.xml debe estar *android:syncable="true"*
2. En la declaración del archivo syncadapter.xml debe estar *android:isAlwaysSyncable="true"* o se setea programáticamente a través de ContentResolver.setIsSyncable()
3. Se activa el sync adapter llamando a ContentResolver.setSyncAutomatically() o el usuario puede activarlo manualmente en configuraciones de cuenta si es que el flag userVisible en syncadapter.xml es true

Configurar esta forma de sincronizar es bastante sencillo, solo se debe agregar:

```
// En activity.onCreate()
...
mResolver.setSyncAutomatically(mAccount, StudentsContract.AUTHORITY , true);
...
```

6.4. Correr SyncAdapter periódicamente

Puedes configurar que el sync adapter se ejecute cada ciertos intervalos de tiempo, o a una hora específica, o ambos. Por ejemplo, puedes subir datos en una hora específica, cuando sabes que tu servidor está menos ocupado. Si ocupas esta estrategia debes procurar que cada dispositivo ejecute la sincronización a una hora ligeramente diferente, para no sobrecargar el servidor.

Una sincronización periódica tiene sentido si el usuario no necesita actualización instantanea pero si necesita que los datos se actualicen regularmente. Un ejemplo podría ser una aplicación de noticias. Sería bueno que la ejecución se realice cuando se han publicado las noticias de la mañana, por lo que una sincronización un poco después de eso podría ser de utilidad.

Para utilizar esta opción debes usar el método addPeriodicSync(). Esto hace que se ejecute la sincronización luego de que un tiempo determinado a pasado. Este tiempo puede variar en algunos segundos para que el framework pueda optimizar las veces que se utiliza el internet al coordinarlo con otros sync adpaters.

Para correr el sync adapter a ciertas horas específicas diariamente debes usar AlarmManager, tema que no se revisará en este tutorial. El método addPeriodicSync() y el setSyncAutomatically() pueden ir juntos perfectamente.

6.5. Correr SyncAdapter por acción del usuario

Es una opción muy poco recomendada, debido a que hace un uso ineficiente de la calenderización del framework. Si se quiere utilizar, se debe usar el método requestSync() frente a la acción específica del usuario. Es importante acá entregar como parámetros unos flags que indiquen que se quiere realizar una sincronización manual. Con estos se le dice al framework que olvide su filosofía y haga la sincronización de inmediato. Se debe usar de la siguiente manera:

```
Bundle settings = new Bundle();
// Fuerza sincronizacion manual
settings.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, true);
// Que se realice de inmediato. Si no se pone este se puede esperar varios segundos hasta que el
    framework decida ejecutar la sincronizacion.
settings.putBooleanContentResolver.SYNC_EXTRAS_EXPEDITED, true);
ContentResolver.requestSync(mAccount, AUTHORITY, settingsBundle);
```

7. Comentarios

Espero que el tutorial haya sido de utilidad para quienes estén tratando de implementar un Sync Adapter, o para quienes por simple curiosidad querían aprender más sobre este importante tema. Agradecería mucho que me avises (aamatte@uc.cl) ante cualquier error en los conceptos o en el código para poder modificarlo.