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

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. Además entrega una serie de beneficios que es importante considerar.

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 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. Authenticator service
5. Clase Sync Adapter
6. Bound Service

Durante el siguiente tutorial se creará 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 o sobre como se realizan los requests. De todas formas en la última página se encuentra un link donde se puede descargar el código fuente de la aplicación.

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. Completar los 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;
    }
}
```

Los datos que pongas en tu base de datos estarán seguros. Esto debido a que android las guarda en un espacio de disco privado al que solo tiene acceso tu aplicación. Si se quiere compartir los datos recolectados hay que implementar un Content Provider, tema de la siguiente sección.

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 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:

```
// 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:

```
// 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 sin la necesidad de que esta esté 'abierta'.

Se pueden usar incluso como una forma de abstraer 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.

3.1. Flujo general

Como se aprecia en la Figura 1 el flujo parte desde tu aplicación obteniendo una instancia de ContentResolver[1]. 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 (un Uri para cada conjunto de datos ofrecido). Por ejemplo, android provee Contacts Provider[2] y Calendar Provider[3]. En la próxima sección se entrará más en detalles en el diseño de las Uris.

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

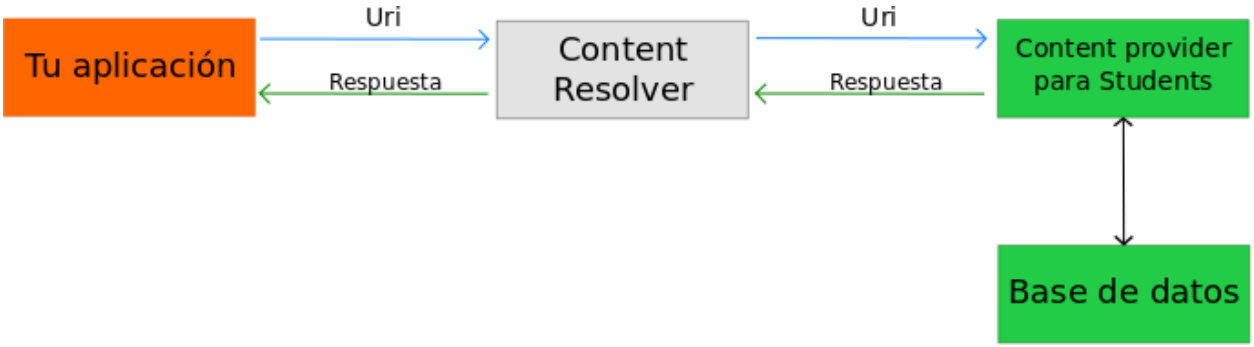

Figura 1: Flujo general de content provider