Persistencia en Android: proveedores de contenidos y SharedPreferences

Índice

1. Shared Preferences

Comenzamos esta sesión hablando de una de las técnicas más simples de persistencia en Android junto al uso de ficheros: *Shared Preferences*. Se trata de un mecanismo ligero para almacenar datos basado en pares clave-valor, utilizado normalmente para guardar preferencias u opciones de la aplicación. También puede ser utilizado para almacenar el estado de la interfaz gráfica para contemplar el caso de que nuestra actividad deba finalizar abruptamente su ejecución. Otro posible uso es el de compartir información entre componentes de una misma aplicación.

Nota:
Los *Shared Preferences* nunca son compartidos entre diferentes aplicaciones.

Este método se basa en el uso de la clase `SharedPreferences`. Es posible utilizarla para guardar datos en muy diversos formatos: booleanos, cadenas, flotantes y enteros.

1.1. Guardar Shared Preferences

Para crear o modificar un *Shared Preference* llamamos al método `getSharedPreferences` del contexto de la aplicación, pasando como parámetro el identificador del conjunto de valores de preferencias con el que queremos trabajar. Para realizar la modificación del valor de un *Shared Preference* hacemos uso de la clase `SharedPreferences.Editor`. Para obtener el objeto editor invocamos el método `edit` del objeto *Shared Preferences* que queremos modificar. Los cambios se guardan mediante el método `commit`. Veamos un ejemplo:

```
int modo = Activity.MODE_PRIVATE;
SharedPreferences misSharedPreferences = getSharedPreferences("Mis
preferencias", modo);

// Obtenemos un editor para modificar las preferencias
SharedPreferences.Editor editor = misSharedPreferences.edit();

// Guardamos nuevos valores en el objeto SharedPreferences
editor.putBoolean("isTrue",true);
editor.putFloat("unFloat",1.0);
editor.putInt("numeroEntero",12);
editor.putLong("otroNumero",2);
editor.putString("unaCadena","valor");

// Guardamos los cambios
editor.commit();
```

1.2. Leer Shared Preferences

Para leer *Shared Preferences* utilizamos el método `getSharedPreferences`, tal como se ha comentado anteriormente. Pasamos como parámetro el identificador del conjunto de

Shared Preferences al que deseamos acceder. Una vez hecho esto ya podemos hacer uso de métodos de tipo `get` para acceder a preferencias individuales. Hay diferentes métodos `get` para diferentes tipos de datos. Cada uno de ellos pasamos como parámetro la clave que identifica la preferencia cuyo valor deseamos obtener, y un valor por defecto, el cual se usará en el caso concreto en el que no existiera une preferencia con la clave pasada como primer parámetro. Podemos ver un ejemplo a continuación:

```
public static String MIS_PREFS = "MIS_PREFS";

public void cargarPreferences() {
        // Obtener las preferencias almacenadas
        int modo = Activity.MODE_PRIVATE;
        SharedPreferences misSharedPreferences =
getSharedPreferences(MIS_PREFS, modo);

        boolean isTrue = misSharedPreferences.getBoolean("isTrue",false);
        float unFloat = misSharedPreferences.getFloat("unFloat",0);
        int numeroEntero = misSharedPreferences.getInt("numeroEntero",0);
        long otroNumero = misSharedPreferences.getLong("otroNumero",0);
        String unaCadena = misSharedPreferences.getString("unaCadena","");
}
```

1.3. Interfaces para Shared Preferences

Android proporciona una plataforma basada en XML para la creación de interfaces gráficas para el manejo de preferencias. Estas interfaces tendrán un aspecto similar al de las del resto de aplicaciones del sistema. Al utilizarla, nos estaremos asegurando de que nuestras actividades para el manejo de preferencias serán consistentes con las del resto de aplicaciones del sistema. De esta forma los usuarios estarán familiarizados con el manejo de esta parte de nuestra aplicación.

La plataforma consiste en tres elementos:

- **Layout de la actividad de preferencias**: se trata de un archivo XML que definirá la interfaz gráfica de la actividad que muestre los controles para modificar los valores de las preferencias. Permite especificar las vistas a mostrar, los posibles valores que se pueden introducir, y a qué clave de *Shared Preferences* se corresponde cada vista.
- **Actividad de preferencias**: una subclase de `PreferenceActivity` que se corresponderá con la pantalla de preferencias de nuestra aplicación.
- **Shared Preference Change Listener**: una implementación de la clase `onSharedPreferenceChangeListener` que actuará en el caso en el que cambie el valor de alguna *Shared Preference*.

1.4. Definiendo una pantalla de preferencias con un layout en XML

Esta es sin duda la parte más importante de una actividad de preferencias. Se trata de un archivo XML utilizado para definir varios aspectos de dicha actividad. Al contrario que en el caso de los archivos de layout para interfaces gráficas que vimos en la sesión correspondiente del módulo de Android, estos archivos de layout se guardan en la carpeta

/res/xml/ de los recursos de la aplicación.

Aunque conceptualmente estos layouts son similares a los utilizados para definir interfaces gráficas, los layouts de actividades de preferencias utilizan un conjunto especializado de componentes específicos para este tipo de actividades. Están pensados para proporcionar interfaces con un aspecto similar al del resto de actividades de preferencias del sistema. Estos controles se describen en más detalle en la siguiente sección.

Los layout de preferencias contendrán un elemento `PreferenceScreen`:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
        xmlns:android="http://schemas.android.com/apk/res/android">
</PreferenceScreen>
```

Se pueden añadir más elementos de este tipo si se desea. Al hacerlo, éstos se representarán como un elemento seleccionable en un listado que mostrará una nueva ventana de preferencias en caso de que sea seleccionado.

Dentro de cada elemento `PreferenceScreen` podemos incluir tantos elementos de tipo `PreferenceCategory` y elementos específicos para añadir controles como se desee. Los elementos `PreferenceCategory` son utilizados para dividir cada pantalla de preferencias en subcategorías mediante una barra de título. Su sintaxis es la siguiente:

```
<PreferenceCategory
        android:title="My Preference Category"/>
</PreferenceCategory>
```

Una vez dividida la pantalla en subcategorías tan sólo quedaría añadir los elementos correspondientes a los controles específicos, que veremos en la siguiente sección. Los atributos de los elementos XML para estos controles pueden variar, aunque existe un conjunto de atributos comunes:

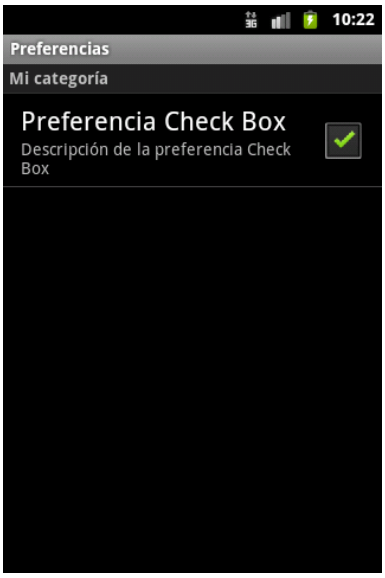
- `android:key`: la clave de Shared Preference que se usará para guardar el valor del control.
- `android:title`: texto a mostrar para describir la preferencia.
- `android:summary`: una descripción más larga a mostrar debajo del texto definido con el campo anterior.
- `android:defaultValue`: el valor por defecto que se mostrará inicialmente y también el que finalmente se guardará si ningún otro valor fue asignado a este campo.

El siguiente listado muestra una pantalla de preferencias muy simple con una única categoría y un único control (un check box):

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
        xmlns:android="http://schemas.android.com/apk/res/android">
        <PreferenceCategory
                android:title="Mi categoría">
                <CheckBoxPreference
                        android:key="MI_CHECK_BOX"
                        android:title="Preferencia Check Box"
```

```
                    android:summary="Descripción de la preferencia
Check Box"
                    android:defaultValue="true"
                />
        </PreferenceCategory>
</PreferenceScreen>
```

Esta pantalla de preferencias se mostraría de la siguiente forma:



Ejemplo sencillo de ventana de preferencias

1.5. Controles nativos para preferencias

Android incluye diferentes controles que podemos añadir a nuestras ventanas de preferencias por medio de los elementos XML que se indican a continuación:

- `CheckBoxPreference`: un check box estándar que puede ser utilizado para preferencias de tipo booleano.
- `EditTextPreference`: permite al usuario introducir una cadena como valor para una preferencia. Al seleccionar el texto se mostrará un diálogo con el que introducir el nuevo valor.
- `ListPreference`: el equivalente a un `Spinner`. Seleccionar este elemento de la interfaz mostrará un diálogo con las diferentes opciones que es posible seleccionar. Se utilizan arrays para asociar valores y textos a las opciones de la lista.
- `RingtonePreference`: un tipo específico de preferencia de tipo listado que permite seleccionar entre diferentes tonos de teléfono. Esta opción es particularmente útil en el caso en el que se esté creando una pantalla de preferencias para configurar las opciones de notificación de una aplicación o componente de la misma.

También es posible crear nuestros propios controles personalizados definiendo una subclase de `Preference` (o cualquiera de sus subclases). Se puede encontrar más

información en la documentación de Android: http://developer.android.com/reference/android/preference/Preference.html.

1.6. Actividades de preferencias

Para mostrar una pantalla de preferencias debemos definir una subclase de `PreferenceActivity`:

```
public class MisPreferencias extends PreferenceActivity {
}
```

La interfaz gráfica de la interfaz se puede crear a partir del layout en el método `onCreate`. mediante una llamada a `addPreferencesFromResource`:

```
@Override
public void onCreate(Bundle savedInstaceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.milayout);
}
```

Como en el caso de cualquier otra actividad, la actividad de preferencias que hayamos definido debe ser incluida en el *Manifest* de la aplicación:

```
<activity android:name=".MisPreferencias"
        android:label="Mis preferencias">
<activity>
```

Eso es todo lo que necesitamos para crear este tipo de actividades y mostrar una ventana con las preferencias definidas en el layout. Esta actividad puede ser iniciada como cualquier otra mediante un `Intent`, ya sea a través de una llamada a `startActivity` o bien a `startActivityForResult`:

```
Intent i = new Intent(this, MisPreferencias.class);
startActivityForResult(i, MOSTRAR_PREFERENCIAS);
```

Los valores para las diferentes preferencias de la actividad son almacenadas en el contexto de la aplicación. Esto permite a cualquier componente de dicha aplicación, incluyendo a cualquier actividad y servicio de la misma, acceder a los valores tal como se muestra en el siguiente código de ejemplo:

```
Context contexto = getApplicationContext();
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(contexto);
// PENDIENTE: hacer uso de métodos get para obtener los valores
```

1.7. Shared Preference Change Listeners

El último elemento que nos queda por examinar en esta plataforma de *Shared Preferences* es la interfaz `onSharedPreferenceChangeListener`, que es utilizada para invocar un evento cada vez que una preferencia es añadida, eliminada o modificada. Para registrar *Listeners* usamos un código como el que se muestra a continuación, en el que la parte más importante es sin duda el método `onSharedPreferenceChanged`, dentro del cual

deberemos determinar qué preferencia ha cambiado de estado a partir de sus parámetros para llevar a cabo las acciones oportunas:

```
public class MiActividad extends Activity implements
OnSharedPreferenceChangeListener {

        @Override
        public void onCreate(Bundle SavedInstanceState) {
                // Registramos este objeto
OnSharedPreferenceChangeListener
                Context contexto = getApplicationContext();
                SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(contexto);
                prefs.registerOnSharedPreferenceChangeListener(this);
        }

        public void onSharedPreferenceChanged(SharedPreferences prefs,
String clave) {
                // PENDIENTE: comprobar qué clave concreta ha cambiado y
su nuevo valor
                // para modificar la interfaz de una actividad o el
comportamiento de
                // algún componente
        }
}
```

2. Proveedores de contenidos

Los proveedores de contenidos o `ContentProvider` proporcionan una interfaz para publicar y consumir datos, identificando la fuente de datos con una dirección URI que empieza por `content://`. Son una forma más estándar que los adaptadores (como el que vimos en la sesión anterior en la sección dedicada a SQLite) de desacoplar la capa de aplicación de la capa de datos.

Podemos encontrar dos tipos principales de proveedores de contenidos en Android. Los proveedores nativos son proveedores que el propio sistema nos proporciona para poder acceder a datos del dispostivo, incluyendo audio, vídeo, imágenes, información personal, etc. Por otra parte, también es posible que creemos nuestros propios proveedores con tal de permitir a nuestra aplicación compartir datos con el resto del sistema. En esta sesión hablaremos de ambos tipos.

2.1. Proveedores nativos

Android incluye una serie de proveedores de contenidos que nos pueden proporcionar información de diferentes tipos: información sobre nuestros contactos, información del calendario e incluso ficheros multimedia. Ejemplos de estos proveedores de contenidos nativos son el `Browser`, `CallLog`, `ContactsContract`, `MediaStore`, `Settings` y el `UserDictionary`. Para poder hacer uso de estos proveedores de contenidos y acceder a los datos que nos proporcionan debemos añadir los permisos adecuados al fichero `AndroidManifest.xml`. Por ejemplo, para acceder al listín telefónico podemos añadir el permiso correspondiente de la siguiente forma:

```
    ...
        <uses-sdk android:minSdkVersion="8" />
        <uses-permission android:name="android.permission.READ_CONTACTS"/>
</manifest>
```

Para obtener información de un `ContentProvider` debemos hacer uso del método `query` de la clase `ContentResolver`. El primero de los parámetros de dicho método será la URI del proveedor de contenidos al que deseamos accder. Este método devuelve un cursor que nos permitirá iterar entre los resultados, tal como se vio en la sesión anterior en el apartado de cursores:

```
ContentResolver.query(
                Uri uri,
                String[] projection,
                String selection,
                String[] selectionArgs,
                String sortOrder)
```

Por ejemplo, la siguiente llamada nos proporcionará un cursor que nos permitirá acceder a todos los contactos de nuestro teléfono móvil. Para ello hemos hecho uso de la constante `ContactsContract.Contacts.CONTENT_URI`, que almacena la URI del proveedor de contenidos correspondiente. Al resto de parámetros se le ha asignado el valor null:

```
        ContentResolver cr = getContentResolver();
        Cursor cursor = cr.query(ContactsContract.Contacts.CONTENT_URI,
                null, null, null, null);
```

Nota:

En versiones anteriores a Android 2.0 las estructuras de datos de los contactos son diferentes y no se accede a esta URI.

Una vez obtenido el cursor es posible mapear sus campos con algún componente de la interfaz gráfica de la actividad. De esta forma cualquier cambio que se produzca en el cursor se reflejará automáticamente en el componente gráfico sin que sea necesario que programemos el código que lo refresque. Esto se consigue con el siguiente método:

```
        cursor.setNotificationUri(cr,
ContactsContract.Contacts.CONTENT_URI);
```

Para asignar un adaptador a una lista de la interfaz gráfica de la actividad, más concretamente una `ListView`, utilizamos el método `setAdapter(Adapter)`:

```
                ListView lv = (ListView)findViewById(R.id.ListView01);
                SimpleCursorAdapter adapter = new SimpleCursorAdapter(
                        getApplicationContext(),
                        R.layout.textviewlayout,
                        cursor,
```

```
                        new String[]{
                                ContactsContract.Contacts._ID,
                                ContactsContract.Contacts.DISPLAY_NAME},
                        new int[]{
                                R.id.TextView1,
                                R.id.TextView2});
                lv.setAdapter(adapter);
```

En este ejemplo los identificadores `R.id.TextView1` y `R.id.TextView2` se corresponden con views del layout que define cada fila de la `ListView`, como se puede ver en el archivo `textviewlayout.xml` que tendríamos que crear:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content">
        <TextView android:id="@+id/TextView1"
          android:textStyle="bold"
          android:ems="2"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content">
        </TextView>
        <TextView android:id="@+id/TextView2"
          android:textStyle="bold"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content">
        </TextView>
</LinearLayout>
```

La lista en sí (identificada por `R.id.ListView01` en el presente ejemplo) estaría en otro XML layout, por ejemplo en `main.xml`.

Se debe tener en cuenta que en el caso de este ejemplo, si quisiéramos obtener los números de teléfono de cada uno de nuestros contactos, tendríamos que recorrer el cursor obtenido a partir del proveedor de contenidos, y por cada persona en nuestra agenda, crear un nuevo cursor que recorriera los teléfonos, ya que una persona puede tener asignados varios teléfonos.

Puedes acceder a un listado completo de los proveedores de contenidos nativos existentes en Android consultando su manual de desarrollo. Concretamente, leyendo la sección dedicada al paquete `android.provider` en http://developer.android.com/reference/android/provider/package-summary.html.

2.2. Proveedores propios: crear un nuevo proveedor de contenidos

Para acceder a nuestras fuentes de datos propias de manera estándar nos interesa implementar nuestros propios `ContentProvider`. Gracias a que proporcionamos nuestros datos a partir de una URI, quien use nuestro proveedor de contenidos no deberá preocuparse de dónde provienen los datos indicados por dicha URI; podrían provenir de ficheros locales en la tarjeta de memoria, de un servidor en Internet, o de una base de

datos SQLite. Dada una URI nuestro proveedor de contenidos deberá proporcionar en su interfaz los métodos necesarios para realizar las operaciones básicas en una base de datos: inserción, lectura, actualización y borrado.

Para crear un nuevo proveedor de contenidos definimos una subclase de `ContentProvider`. Utilizaremos una sobrecarga del método `onCreate` para crear e inicializar la fuente de datos que queramos hacer pública a través de dicho proveedor. Un esqueleto de subclase de `ContentProvide` podría ser el siguiente:

```
public class MiProveedor extends ContentProvider {

        @Override
        public boolean onCreate() {
                // Inicializar la base de datos
                return true;
        }
}
```

Dentro de la clase deberemos crear un atributo público y estático de nombre `CONTENT_URI`, en el cual almacenaremos el URI completo del proveedor que estamos creando. Este URI debe ser único, por lo que una buena idea puede ser tomar como base el nombre de paquete de nuestra aplicación. La forma general de un URI es la siguiente:

```
content://[NOMBRE_PAQUETE].provider.[NOMBRE_APLICACION]/[RUTA_DATOS]
```

Por ejemplo:

```
content://es.ua.jtech.android.provider.miaplicacion/elementos
```

Estos URIs se pueden presentar en dos formas. La URI anterior representa una consulta dirigida a obtener todos los valores de ese tipo (en este caso todos los elementos). Si a la URI anterior añadimos un sufijo `/[NUMERO_FILA]`, estamos representando una consulta destinada a obtener un único elemento (por ejemplo, el quinto elemento en el siguiente ejemplo):

```
content://es.ua.jtech.android.provider.miaplicacion/elementos/5
```

Nota:

Se considera una buena práctica de programación el proporcionar acceso a nuestro proveedor de contenidos utilizando cualquiera de estas dos formas.

La forma más simple de determinar cuál de las dos formas anteriores de URI se utilizó para hacer una petición a nuestro proveedor de contenidos es mediante un *Uri Matcher*. Crearemos y configuramos un elemento *Uri Matcher* para analizar una URI y determinar a cuál de las dos formas se corresponde. En el siguiente código se muestra el código básico que deberemos emplear para implementar este patrón:

```
public class MiProveedor extends ContentProvider {

        private static final String miURI =
"content://es.ua.jtech.android.provider.miaplicacion/elementos";
        public static final Uri CONTENT_URI = Uri.parse(miUri);
```

```
        @Override
        public boolean onCreate() {
                // PENDIENTE: inicializar la base de datos
                return true;
        }

        // Creamos las constantes que nos van a permitir diferenciar
        // entre los dos tipos distintos de peticiones a partir
        // de la URI
        private static final int TODAS_FILAS = 1;
        private static final int UNA_FILA = 2;

        private static final UriMatcher uriMatcher;

        // Inicializamos el objeto UriMatcher. Una URI que termine en
        // 'elementos' se corresponderá con una consulta para todas las
        // filas, mientras que una URI con el sufijo 'elementos/[FILA]'
        // representará una única fila
        static {
                uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
uriMatcher.addURI("es.ua.jtech.android.provider.miaplicacion",
                        "elementos",TODAS_FILAS);
uriMatcher.addURI("es.ua.jtech.android.provider.miaplicacion",
                        "elementos/#",UNA_FILA);
        }
}
```

Esta misma técnica se puede emplear para proporcionar diferentes alternativas de URI para diferentes subconjuntos de datos, como por ejemplo diferentes tablas en una base de datos, a través del mismo proveedor de contenidos.

2.3. Proveedores propios: crear la interfaz de consultas

Para permitir realizar operaciones con los datos publicados a través de nuestro proveedor de contenidos deberemos implementar los métodos `delete`, `insert`, `update` y `query` para el borrado, inserción, actualización y realización de consultas, respectivamente. Estos métodos son la interfaz estándar utilizada con proveedores de contenidos, de tal forma que para compartir datos entre aplicaciones no se tendrán interfaces distintos para diferentes fuentes de datos.

Lo más habitual suele ser implementar un proveedor de contenidos como un mecanismo para permitir el acceso a una base de datos SQLite privada a una aplicación, aunque a través de estos métodos indicados se podría en principio acceder a cualquier fuente de datos.

El siguiente código muestra el esqueleto de estos métodos dentro de una subclase de `ContentProvider`. Obsérvese que se hace uso del objeto `UriMatcher` para determinar que tipo de consulta se está realizando:

```
@Override
public Cursor query(Uri uri,
        String[] projection,
        String selection,
        String[] selectionArgs,
        String sort) {
```

```
        // Si se trata de una consulta para obtener una única fila,
limitamos
        // los resultados a obtener de la fuente de datos por medio de una
        // cláusula where
        switch(UriMatcher.match(uri)) {
                case UNA_FILA:
                        // PENDIENTE: modifica la sentencia SELECT
mediante una cláusula
                        // where, obteniendo el identificador de fila
como:
                        // numeroFila = uri.getPathSegments().get(1));
        }
        return null;
}

@Override
public Uri insert(Uri _uri, ContentValues _valores) {
        long idFila = [ ... Añadir un nuevo elemento ... ]

        // Devolvemos la URI del elemento recién añadido
        if (idFila > 0)
                return ContentUris.withAppendendId(CONTENT_URI, idFila);

        throw new SQLException("No se pudo añadir un nuevo elemento a " +
_uri);
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
        switch(uriMatcher.match(uri)) {
                case TODAS_FILAS:
                case UNA_FILA:
                default:
                        throw new IllegalArgumentException("URI no
soportada: " + uri);
        }
}

@Override
public int update(Uri uir, ContentValues valores, String where, String[]
whereArgs) {
        switch(uriMatcher.match(uri)) {
                case TODAS_FILAS:
                case UNA_FILA:
                default:
                        throw new IllegalArgumentException("URI no
soportada: " + uri);
        }
}
```

2.4. Proveedores propios: tipo MIME

El último paso para crear un proveedor de contenidos es definir el tipo MIME que identifica el tipo de datos que devuelve el proveedor. Debemos sobrecargar el método `getType` para que devuelva una cadena que identifique nuestro tipo de datos de manera única. Se deberían definir dos tipos posibles, uno para el caso de una única fila y otro para el caso de devolver todos los resultados. Se debe utilizar una sintaxis similar a la del siguiente ejemplo:

- Un único elemento:
  `vnd.es.ua.jtech.android.cursor.item/miproveedorcontent`
- Todos los elementos:
  `vnd.es.ua.jtech.android.cursor.dir/miproveedorcontent`

Como se puede observar, la cadena comienza siempre por `vnd`. A continuación tendríamos el nombre del paquete de nuestra aplicación. Lo siguiente es `cursor`, ya que nuestro proveedor de contenidos está devolviendo datos de tipo `Cursor`. Justo antes de la barra pondremos `item` en el caso del tipo MIME para un único elemento, y `dir` para el caso del tipo MIME para todos los elementos. Finalmente, tras la barra, pondremos el nombre de nuestra clase (en minúsculas) seguido de `content`. En el siguiente código de ejemplo se muestra cómo sobrecargar `getType` según la URI:

```
@Override
public String getType(Uri _uri) {
        switch (uriMatcher.match(_uri)) {
                case TODAS_FILAS:
                        return
"vnd.es.ua.jtech.android.cursor.dir/miproveedorcontent";
                case UNA_FILA:
                        return
"vnd.es.ua.jtech.android.cursor.item/miproveedorcontent";
                default:
                        throw new IllegalArgumentException("URI no
soportada: " + _uri);
        }
}
```

2.5. Proveedores propios: registrar el proveedor

No debemos olvidar incorporar nuestro proveedor de contenidos al *Manifest* de la aplicación una vez que lo hemos completado. Esto se hará por medio del elemento `provider`, cuyo atributo `authorities` podremos utilizar para especificar la URI base, tal como se puede ver en el siguiente ejemplo:

```
<provider android:name="MiProveedor"
        android:authorities="es.ua.jtech.android.miaplicacion"/>
```

2.6. Content Resolvers

Aunque en la sección de proveedores nativos ya se ha dado algún ejemplo de cómo hacer uso de un *Content Resolver* para realizar una consulta, ahora que hemos podido observar en detalle la estructura de un proveedor de contenidos propio es un buen momento para ver en detalle cómo realizar cada una de las operaciones que éstos permiten.

El contexto de una aplicación incluye siempre una instancia de la clase `ContentResolver`, a la cual se puede acceder mediante el método `getContentResolver`. Esta clase incorpora diversos métodos para realizar consultas a proveedores de contenidos. Cada uno de estos métodos acepta como parámetro una URI que indica con qué proveedor de contenidos se desea interactuar.

Las consultas realizadas sobre un proveedor de contenidos recuerdan a las consultas realizadas sobre bases de datos. Los resultados de las consultas se devuelven como objetos de la clase `Cursor`. Se pueden extraer valores del cursor de la misma forma en la que se explicó en el apartado de bases de datos de la sesión anterior. El método `query` de la clase `ContentResolver` acepta los siguientes parámetros:

- La URI del proveedor del contenidos al que se desea realizar la consulta.
- Una proyección que enumera las columnas que se quieren incluir en los resultados obtenidos.
- Una cláusula *where* que define las filas a obtener. Se pueden utilizar comodines *?* que se reemplazarán por valores incluidos en el siguiente parámetro.
- Una matriz de cadenas que se pueden utilizar para reemplazar los comodines *?* en la cláusula *where* anterior.
- Una cadena que describa la ordenación de los elementos devueltos.

El siguiente código muestra un ejemplo:

```
ContentResolver cr = getContentResolver();
// Devolver todas las filas
Cursor todasFilas = cr.query(MiProveedor.CONTENT_URI, null, null, null,
null);
// Devolver todas columnas de las filas para las que la columna 3
// tiene un valor determinado, ordenadas según el valor de la columna 5
String where = COL3 + "=" + valor;
String orden = COL5;
Cursor algunasFilas = cr.query(MiProveedor.CONTENT_URI, null, where, null,
orden);
```

2.7. Otras operaciones con Content Resolvers

Para realizar otro tipo de operaciones con los datos de un proveedor de contenidos haremos uso de los métodos `delete`, `update` e `insert` de un objeto `ContentResolver`.

Con respecto a la **inserción**, la clase `ContentResolver` proporciona en su interfaz dos métodos diferentes: `insert` y `bulkInsert`. Ambos aceptan como parámetro la URI del proveedor de contenidos, pero mientras que el primer método tan solo toma como parámetro un objeto de la clase `ContentValues`, el segundo toma como parámetro una matriz de elementos de este tipo. El método `insert` devolverá la URI del elemento recién añadido, mientras que `bulkInsert` devolverá el número de filas correctamente insertadas:

```
// Obtener el Content Resolver
ContentResolver cr = getContentResolver();

// Crear una nueva fila de valores a insertar
ContentValues nuevosValores = new ContentValues();

// Asignar valores a cada columna
nuevosValores.put(NOMBRE_COLUMNA, nuevoValor);
[ ... Repetir para cada columna ... ]

Uri miFilaUri = cr.insert(MiProveedor.CONTENT_URI, nuevosValores);
```

```
// Insertamos ahora una matriz de nuevas filas
ContentValues[] arrayValores = new ContentValues[5];
// PENDIENTE: rellenar el array con los valores correspondientes
int count = cr.bulkInsert(MiProveedor.CONTENT_URI, arrayValores);
```

Para el caso del **borrado** hacemos uso del método `delete` del *Content Resolver* pasando como parámetro la URI de la fila que deseamos eliminar de la fuente de datos. Otra posibilidad es incluir una cláusula *where* para eliminar múltiples filas. Ambas técnicas se muestran en el siguiente ejemplo:

```
ContentResolver cr = getContentResolver();

// Eliminar una fila específica
cr.delete(miFilaUri, null, null);
// Eliminar las primeras cinco filas
String where = "_id < 5";
cr.delete(MiProveedor.CONTENT_URI, where, null);
```

Una **actualización** se lleva a cabo mediante el método `update` del objeto *Content Resolver*. Este método toma como parámetro la URI del proveedor de contenidos objetivo, un objeto de la clase `ContentValues` que empareja nombres de columna con valores actualizados, y una cláusula *where* que indica qué filas deben ser actualizadas. El resultado de la actualización será que en cada fila en la que se cumpla la condición de la cláusula *where* se cambiarán los valores de las columnas especificados por el objeto de la clase `ContentValues`; también se devolverá el número de filas correctamente actualizadas. A continuación se muestra un ejemplo:

```
ContentValues nuevosValores = new ContentValues();

// Utilizamos el objeto ContentValues para especificar en qué columnas
// queremos que se produzca un cambio, y cuál debe ser el nuevo valor
// de dichas columnas
nuevosValues.put(NOMBRE_COLUMNA, nuevoValor);

// Aplicamos los cambios a las primeras cinco filas
String where = "_id < 5";

getContentResolver().update(MiProveedor.CONTENT_URI, nuevosValores, where,
null);
```