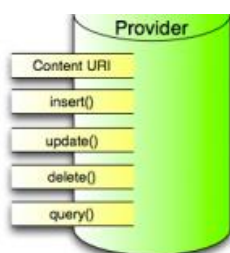


Content Provider





Contenidos

Componentes de una aplicación	3
Content Provider - Proveedores de contenido	3
Content Provider nativos	4
URI	4
URI's de Content Provider nativos	5
Content Provider propios	5
Caso práctico para Content Provider propio	5
Construcción de Content Provider propio	5
Modelo de datos	6
Clase Constantes	7
Clase MiProveedor	9
Método getType()	10
Método onCreate()	11
Método insert()	11
Método query()	11
Método update()	12
Método delete()	12
Optimizando código	13
Declarar Content provider	13
Crear el Content Provider	14
Acceder al Content Provider	14
Clase ContantesUsar	15
Insertar registros	16
Obtención de plato buscado por _ID (clave)	16
Obtención de plato buscado por nombre (no clave)	16
Obtener todos los registros	17
Borrar el plato de un determinado nombre (no clave)	17
Modificar un registro buscado por _ID (clave)	17
Listado con filtro de tipo de platos (Primeros y/o Segundos y/o Postres)	18
Usando Content Provider nativos	19
Caso práctico Content Nativo Llamadas	19
Mejoras	21
Prácticas propuestas	21

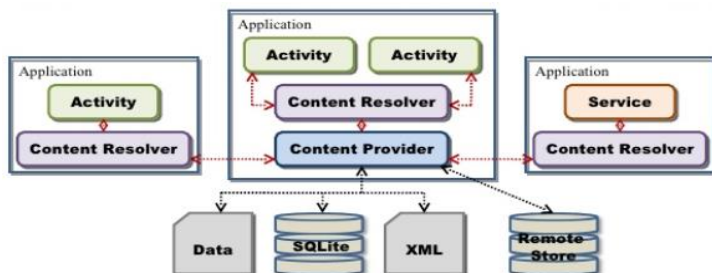
Componentes de una aplicación

- Activity ✓
- Intent ✓
- Content Provider
- Broadcast Receiver
- Service



Content Provider - Proveedores de contenido

Es el mecanismo proporcionado por la plataforma Android para permitir compartir información ya que gestiona el acceso a un repositorio central de datos.



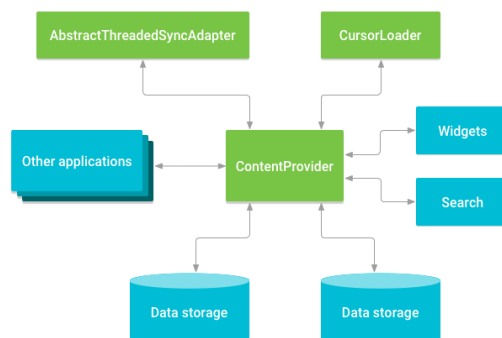
Los datos a compartir pueden estar guardados en archivos del sistema, en una base de datos en SQLite, en la web o en cualquier otro lugar de almacenamiento persistente a la que la aplicación pueda tener acceso.

Un proveedor de contenido presenta los datos a aplicaciones externas como una o más tablas que son similares a las tablas que se encuentran en una base de datos relacional. Una fila representa una instancia de un cierto tipo de datos que el proveedor recoge y cada columna de la fila representa una pieza individual de los datos recogidos para una instancia.

Una aplicación que desee que todo o parte de la información que almacena esté disponible de una forma controlada para el resto de aplicaciones del sistema se dice que es la **aplicación proveedora** y deberá proporcionar un Content Provider a través del cual se pueda realizar el acceso a dicha información y fijar los permisos de uso. A menudo, la propia aplicación proporciona su propia interfaz para que trabajar con los datos; pero no es obligatorio, la función principal de la aplicación es la de ser el repositorio de los datos. Por tanto, los proveedores de contenido están destinados principalmente a ser utilizados por otras aplicaciones. Las aplicaciones que hacen uso de esos datos se dicen que son **aplicaciones clientes** y deberán tener los permisos necesarios para acceder a ellos.

Un Content Provider es necesario:

- si se comparten datos entre múltiples aplicaciones:
- si se desea implementar sugerencias de búsqueda personalizada en solicitudes de datos usando SearchRecentSuggestionsProvider
- si hay que exponer datos de una aplicación en widgets
- si hay que sincronizar datos de la aplicación con el servidor utilizando una implementación de AbstractThreadedSyncAdapter
- si hay que cargar datos en la interfaz de usuario utilizando un CursorLoader





Content Provider nativos

Android viene con un número determinado de Content Provider para tipos de datos comunes (audio, video, imágenes, información personal de contactos, etc.). Se pueden ver algunos de ellos en el paquete android.provider.

Content Provider	Información contenida
Browser	Marcadores, historial de navegación, etc.
CallLog	Llamadas perdidas, detalles de llamada, etc.
ContactsContract	Detalles de contactos
MediaStore	Archivos multimedia como imágenes, audio y vídeo
Settings	Preferencias y ajustes del dispositivo

Esto quiere decir que podríamos acceder a los datos desde nuestras propias aplicaciones clientes haciendo uso de los Content Providers correspondientes (aunque para algunos de ellos, se deben conseguir permisos adecuados para leer los datos).

URI

El **acceso** a un *content provider* se realiza siempre mediante una URI (cadena de texto que permite identificar un recurso de manera única).

Las URI para realizar consultas sobre Content Provider tienen el siguiente formato:

content://<autoridad>/<ruta_de_datos>/<identificador>

- En primer lugar el prefijo "*content://*" que indica que dicho recurso deberá ser tratado por un *content provider*.
- En segundo lugar se indica el identificador en sí del *content provider*, también llamado *authority*. (Dado que este dato debe ser único es una buena práctica en las aplicaciones proveedoras que al crear el contenedor se utilice un authority de tipo "nombre de clase java invertido", ej:com.pdm.qqq, siendo qq q la aplicación proveedora de contenidos)
- A continuación, se indica la entidad concreta a la que queremos acceder dentro de los datos que proporciona el *content provider* (p.e.: nombre de la tabla a la que se quiere acceder). Un *content provider* que controla varios conjuntos de datos (múltiples tablas), tiene un URI por cada uno.

Ejemplo: **content://com.pdm.qqq/clientes**

- Por último, en una URI se puede hacer referencia directamente a un registro concreto de la entidad seleccionada. Esto se haría indicando al final de la URI el ID de dicho registro.

Ejemplo: **content://com.pdm.qqq/clientes/7**



URI's de Content Provider nativos

Por ejemplo para acceder al registro de llamadas sería necesario construir la siguiente URI:

`content://call_log/calls`

Android define las constantes `CONTENT_URI` para todos los *providers* que vienen con la plataforma:

Ejemplo de tres URIs definidos en Android:

- `MediaStore.Images.Media.INTERNAL_CONTENT_URI`
- `MediaStore.Images.Media.EXTERNAL_CONTENT_URI`
- `ContactsContract.Contacts.CONTENT_URI`

Las cadenas de texto equivalentes a estas URIs serían las siguientes:

- `content://media/internal/images`
- `content://media/external/images`
- `content://com.android.contacts/contacts/`

Content Provider propios

Si se quiere hacer públicos los datos almacenados por una aplicación, se tienen dos opciones:

- Se puede crear un *content provider* (una subclase de `ContentProvider`)
- Se pueden añadir los datos a un *provider* ya existente (si existe uno que controle el mismo tipo de datos y se tiene permiso para escribir).

Caso práctico para Content Provider propio

Supongamos que un restaurante desea tener aplicaciones distintas que puedan acceder a los datos sobre platos guardados en una BD SQLite (por ejemplo, los cocineros usan una aplicación y los camareros otra).

Debe haber una aplicación proveedora de contenidos que gestione esa persistencia de datos a través de la creación de un Content Provider (Proyecto P_72_Content_Crear). Las aplicaciones clientes harán uso de ese Content Provider (Proyecto P_73_Content_Usar).

Construcción de Content Provider propio

Crea con AS el proyecto P_72_Content_Crear (la aplicación proveedora de contenidos) para analizar el código que permite gestionar la persistencia de datos deseada.

Pasos a seguir para la construcción del Content Provider:

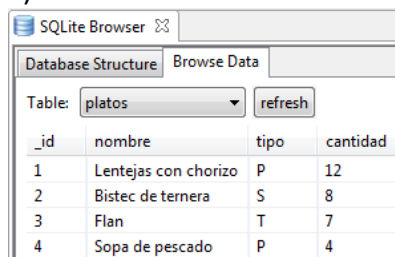
1. Definir el modelo de datos a compartir
2. Crear una clase llamada Constantes donde fijaremos las múltiples constantes que usaremos en los métodos de nuestros códigos (no es obligatorio, aunque sí aconsejable para poder reutilizar el proyecto como base para otros semejantes)
3. Crear una clase que extienda de la clase `ContentProvider` y sobrescribir los diversos métodos definidos dentro del mismo: funciones "CRUD" (crear, recuperar, actualizar y eliminar) de los datos persistentes.
4. Declarar y fijar los permisos necesarios en `AndroidManifest.xml` (IMPORTANTÍSIMO no olvidar en el caso de "copia y pega de otros proyectos ☺!)



Modelo de datos

El proveedor de contenido presenta los datos a aplicaciones externas como una tabla que se encuentran en una base de datos SQLite.

En el siguiente ejemplo, tenemos como se presentan los distintos platos de nuestro caso práctico (P_72_Content_Crear):



The screenshot shows the SQLite Browser interface. The 'Database Structure' tab is active, and the 'platos' table is selected. The table has four columns: _id, nombre, tipo, and cantidad. The data is as follows:

_id	nombre	tipo	cantidad
1	Lentejas con chorizo	P	12
2	Bistec de ternera	S	8
3	Flan	T	7
4	Sopa de pescado	P	4

Cada registro incluye un campo numérico `_id` que identifica unívocamente el registro en la tabla. No se está obligado a tener una clave principal y no es necesario utilizar `_id` como el nombre de la columna de una clave principal si existe otra clave. Sin embargo, si desea enlazar los datos de un proveedor a un `ListView`, uno de los nombres de columna debe ser `_id` ya que facilita la tarea.

En nuestro proyecto este modelo está presente por la clase `AdminSqlite`:

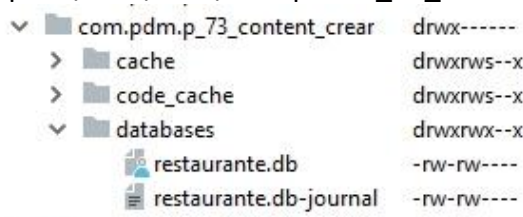
```
public class AdminSqlite extends SQLiteOpenHelper {
    private static AdminSqlite sInstance;
    public static synchronized AdminSqlite getInstance(Context context, String name, CursorFactory
factory, int version) {
        if (sInstance == null) {
            sInstance = new AdminSqlite(context.getApplicationContext(), name, factory, version);
        }
        return sInstance;
    }

    public AdminSqlite(Context context, String name, CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE platos (_id INTEGER primary key AUTOINCREMENT, nombre TEXT, tipo
TEXT, cantidad INTEGER)");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS platos");
        db.execSQL("CREATE TABLE platos (_id INTEGER primary key AUTOINCREMENT, nombre TEXT, tipo
TEXT, cantidad INTEGER)");
    }
}
```

La BD se creará en la carpeta `/data/data/com.pdm.P_72_Content_Crear/databases/`





Clase Constantes

Creamos una clase llamada Constantes que iremos rellenando poco a poco. Para otros casos de proveedores de contenido, generalmente nos bastará con cambiar los valores de estas constantes y el código de la clase de tipo *ContentProvider* nos servirá igual.

```
public class Constantes {  
}
```

Constantes para trabajar con la BD SQLite:

Podemos fijar nombre y versión de la BD

```
// Constantes para la Base de datos  
static final String BD_NOMBRE = "restaurante.db";  
static final int BD_VERSION = 1;
```

Constantes para fijar el proveedor de contenidos:

Es una buena idea definir una constante para el URI, para simplificar el código y para que las futuras actualizaciones sean más limpias. Además, para seguir la práctica habitual de todos los *content providers* de Android, encapsularemos además esta dirección en un objeto estático de tipo *Uri* llamado *CONTENT_URI*.

```
//Autoridad del Content Provider  
static final String AUTHORITY = "com.pdm.restaurante";  
// Representación de la tabla a consultar  
static final String TABLA = "platos";  
// URI de contenido principal  
static final String uri = "content://" + AUTHORITY + "/" + TABLA;  
static final Uri CONTENT_URI = Uri.parse(uri);
```

Constantes para interpretar los URI's:

La primera tarea que nuestro *content provider* deberá hacer cuando se acceda a él, será interpretar la URI utilizada. En nuestro ejemplo, hay dos posibles casos:

- "content://com.pdm.restaurante/platos"→ Acceso a todos los registros de la tabla platos
- "content://com.pdm.restaurante/platos/3"→ Acceso directo al registro de la tabla platos con *_id* = 3

Definiremos dos nuevas constantes que representen los dos tipos de URI que hemos indicado: acceso genérico a tabla (lo llamaré *TODOS*) o acceso a registro por clave (lo llamaré *UNO*):

```
// Para URIs de multiples registros  
static final int TODOS = 0;  
// Para URIS de un solo registro  
static final int UNO = 1;
```



La clase [UriMatcher](#) es capaz de interpretar determinados patrones en una URI. Por eso, definiremos también un objeto UriMatcher y lo inicializaremos (agregando la URI NO_MATCH) y le añadiremos los tipos de formato de URI susceptibles de ser manejadas por nuestro ContentProvider, de forma que pueda diferenciarlos y devolvernos el tipo que devuelven (una de las dos constantes definidas, TODOS o UNO).

```
// Comparador de URIs de contenido
static final UriMatcher uriMatcher;
// Asignación de URIs
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(AUTHORITY, TABLA, TODOS);
    uriMatcher.addURI(AUTHORITY, TABLA + "/#", UNO);
}
```

El método addUri() recibe los siguientes parámetros:

- authority: La autoridad del Content Provider a comparar.
- path: La ruta a comparar. Se pueden usar los caracteres comodín que sean necesarios; por eso, usamos # para representar a cadenas que tengan caracteres numéricos y cualquier tamaño.
- code: El código o identificador que distingue la URI de contenido.

El primer caso representa la URI de contenido principal, donde el cliente consulta múltiples platos. El segundo caso representa el conjunto de todas las URIs que consulten plato por su identificador.

Constantes para los tipos MIME

Un tipo MIME es un identificador de formato estándar, que representa la estructura de un flujo de datos que será transmitido. Su uso es indispensable en las comunicaciones web.

Para usar este tipo de referencias se usa un tipo y un subtipo: tipo/subtipo. Ejemplo: image/jpg.

Una vez más existirán dos tipos MIME distintos para cada entidad del *content provider*, uno de ellos destinado a cuando se devuelve una lista de registros como resultado y otro para cuando se devuelve un registro único concreto.

Para nuestros Content Providers que usan tablas usaremos el formato MIME "vendor-specific" que usa la cadena **vnd** antepuesta al tipo y al subtipo.

- Tipo: esta sección del formato MIME definido para el framework de Android depende del número de elementos a retornar:
 - Si son múltiples elementos, entonces se usa **android.cursor.dir**
 - Si es un solo ítem, se usa **android.cursor.item**
- Subtipo: Esta sección del formato MIME depende de la forma de la autoridad del Content Provider en concreto. Usaremos, por ejemplo, el nombre del paquete y una cadena que distinga la tabla: **<paquete>.<cadena>**

Seguiremos los siguientes patrones para definir uno y otro tipo de datos:

- Registro único: "vnd.android.cursor.item/vnd.xxxxxx"
- Lista de registros: "vnd.android.cursor.dir/vnd.xxxxxx"

En nuestro caso los tipos serían

- "vnd.android.cursor.dir/vnd.com.pdm.plato" (todos)
- "vnd.android.cursor.item/vnd.com.pdm.plato" (uno)

Con estas pautas claras, declararemos en Constantes un tipo MIME para la obtención de múltiples registros y otro para un solo registro:

```
//Tipo MIME que devuelve la consulta de una sola fila
static final String MIME_SIMPLE = "vnd.android.cursor.item/vnd.pdm." + TABLA;
//Tipo MIME que devuelve la consulta de todas las filas
static final String MIME_MULTIPLE = "vnd.android.cursor.dir/vnd.pdm." + TABLA;
```



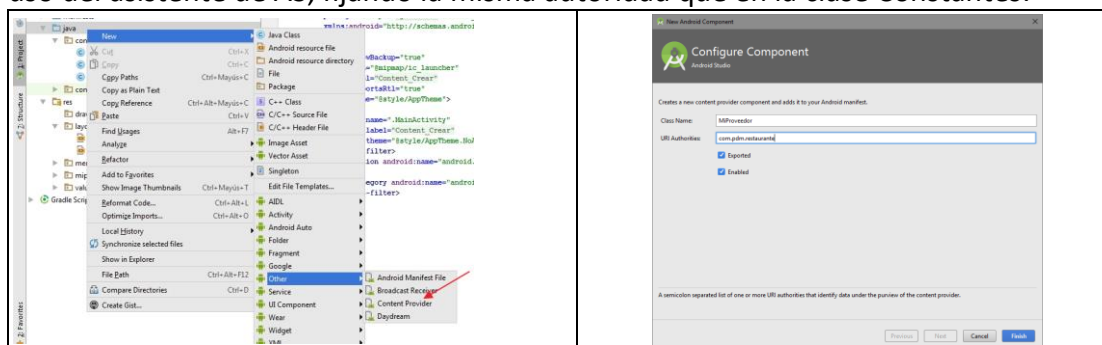

Clase MiProveedor

La aplicación proveedora de datos persistentes debe crear una clase que **extienda de la clase [ContentProvider](#)**.

Los métodos abstractos que tendremos que implementar son los siguientes:

- **onCreate():** para inicializar todos los recursos necesarios para el funcionamiento del nuevo *content provider*.
- **query():** para consultar de datos
- **insert():** para inserción
- **update():** para actualización
- **delete():** para borrado
- **getType():** tipo de datos devueltos por el *content provider*

Si hay que crear este tipo de clase que extiende de ContentProvider, lo mejor es hacer uso del asistente de AS, fijando la misma autoridad que en la clase Constantes.



El asistente nos proporciona la plantilla base de la clase y además añade la declaración en el AndroidManifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.pdm.p_72_content_crear">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="P_72 Content_Crear"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
        <provider
            android:name=".MiProveedor"
            android:authorities="com.pdm.restaurant"
            android:enabled="true"
            android:exported="true"></provider>

        <activity
            android:name=".MainActivity"
            android:label="P_72 Content_Crear"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```



La plantilla que suministra el asistente tiene los métodos por orden alfabético y vacíos:

```
public class MiProveedor extends ContentProvider {
    public MiProveedor() {
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        // Implement this to handle requests to delete one or more rows.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public String getType(Uri uri) {
        // TODO: Implement this to handle requests for the MIME type of the data
        // at the given URI.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        // TODO: Implement this to handle requests to insert a new row.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public boolean onCreate() {
        // TODO: Implement this to initialize your content provider on startup.
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {
        // TODO: Implement this to handle query requests from clients.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
        // TODO: Implement this to handle requests to update one or more rows.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Método `getType()`

El método permite obtener el tipo de datos que devuelve el *content provider* correspondiente a una Uri que envíe el cliente como parámetro.

En nuestro caso, la implementación es sencilla. Solo usamos una estructura switch junto a la clase UriMatcher para determinar los posibles casos. Utilizaremos las constantes definidas previamente:

```
@Override
public String getType(Uri uri) {
    int match = Constantes.uriMatcher.match(uri);
    switch (match) {
        case Constantes.TODOS:
            return Constantes.MIME_MULTIPLE;
        case Constantes.UNO:
            return Constantes.MIME_SIMPLE;
        default:
            return null;
    }
}
```

La dinámica consiste en comparar el parámetro uri que llega por `getType()` con los patrones que tenemos almacenados en `uriMatcher`. Según sea el caso así retornaremos el tipo MIME correspondiente.



Método onCreate()

En este método inicializamos nuestra base de datos, a través de su nombre y versión y utilizando para ello la clase MiAdminSqlite creada para implementar el modelo de datos:

```

@Override
public boolean onCreate() {
    MiAdminSqlite admin = MiAdminSqlite.getInstance(getContext(), Constantes.BD_NOMBRE, null,
Constantes.BD_VERSION);
    SQLiteDatabase db = admin.getWritableDatabase();
    if (db == null) {
        return false;
    }
    if (db.isReadOnly()) {
        db.close();
        return false;
    }
    db.close();
    return true;
}

```

Método insert()

Devuelve la URI que hace referencia al nuevo registro insertado.

Para ello, obtendremos el nuevo ID del elemento insertado como resultado del método insert() de SQLiteDatabase, y posteriormente construiremos la nueva URI mediante el método auxiliar ContentUris.withAppendedId() que recibe como parámetro la URI de nuestro content provider y el ID del nuevo elemento.

```

@Override
public Uri insert(Uri uri, ContentValues values) {
    MiAdminSqlite admin = MiAdminSqlite.getInstance(getContext(), Constantes.BD_NOMBRE, null,
Constantes.BD_VERSION);
    SQLiteDatabase db = admin.getWritableDatabase();
    long regId = db.insert(Constantes.TABLA, null, values);
    Uri newUri = ContentUris.withAppendedId(Constantes.CONTENT_URI, regId);
    return newUri;
}

```

Método query()

El siguiente paso es sobrescribir el método query() para retornar un cursor de datos hacia las aplicaciones cliente. Lo que quiere decir que usaremos SQLiteDatabase.query() dentro de ContentResolver.query().

El método tiene los parámetros necesarios para realizar la consulta:

public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)

- una URI (**Uri uri**, semejante a *from* de SQL),
- una lista de nombres de columna (**String[] projection**, semejante a *select* de SQL; si es null es como *** de SQL),
- un criterio de selección (**String selection**, semejante a *where* de SQL),
- una lista de valores para las variables utilizadas en el criterio anterior (**String[] selectionArgs**) y
- un criterio de ordenación (**String sortOrder**, semejante a *order by* de SQL).

```

@Override
public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
    MiAdminSqlite admin = MiAdminSqlite.getInstance(getContext(), Constantes.BD_NOMBRE, null,
Constantes.BD_VERSION);
    SQLiteDatabase db = admin.getWritableDatabase();
    String where = null;
    switch (Constantes.uriMatcher.match(uri)) {
        case Constantes.TODOS:
            where=selection;
            break;
        case Constantes.UNO:
            where=" _id = "+ uri.getLastPathSegment();
            break;
    }
    Cursor cursor = db.query(Constantes.TABLA, projection, where, selectionArgs, null, null,
sortOrder);
    return cursor;
}

```

Para distinguir entre los dos tipos de URI posibles utilizaremos el objeto uriMatcher, utilizando su método match().

Si se trata de un acceso a un plato concreto, para obtener su clave se usa el método getLastPathSegment() del objeto uri que extrae el último elemento de la URI.

El método devuelve un cursor resultado de realizar la consulta a BD mediante el método query() de la clase SQLiteDatabase. Los null son debidos a que no utilizamos clausulas Group ni Having en la consulta.



Método update()

Actualiza la BD y devuelve el número de registros afectados. La actualización es muy similar a la inserción, tanto que podemos reutilizar la mayor parte del código (y por tanto refactorizar a un método).

```
@Override
public int update(Uri uri, ContentValues values, String selection,
    String[] selectionArgs) {
    MiAdminSqlite admin = MiAdminSqlite.getInstance(getContext(), Constantes.BD_NOMBRE, null,
        Constantes.BD_VERSION);
    SQLiteDatabase db = admin.getWritableDatabase();
    String where = null;
    switch (Constantes.uriMatcher.match(uri)){
        case Constantes.TODOS:
            where=selection;
            break;
        case Constantes.UNO:
            where=" _id = "+ uri.getLastPathSegment();
            break;
    }
    int cont = db.update(Constantes.TABLA, values, where, selectionArgs);
    return cont;
}
```

Método delete()

Devuelve el número de registros afectados.

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    MiAdminSqlite admin = MiAdminSqlite.getInstance(getContext(), Constantes.BD_NOMBRE, null,
        Constantes.BD_VERSION);
    SQLiteDatabase db = admin.getWritableDatabase();
    String where = null;
    switch (Constantes.uriMatcher.match(uri)){
        case Constantes.TODOS:
            where=selection;
            break;
        case Constantes.UNO:
            where=" _id = "+ uri.getLastPathSegment();
            break;
    }
    int cont = db.delete(Constantes.TABLA, where, selectionArgs);
    return cont;
}
```

Nota: El trabajo de código de la clase parece laborioso, pero solo la primera vez; en cuanto tengamos un Provider hecho (el que acabamos de estudiar, por ejemplo), nos bastará "cortar y pegar" y revisar el código relativo a las constantes, los métodos nos servirán casi tal y como están.



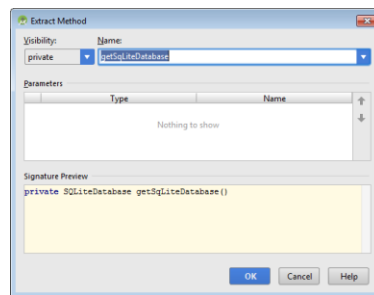
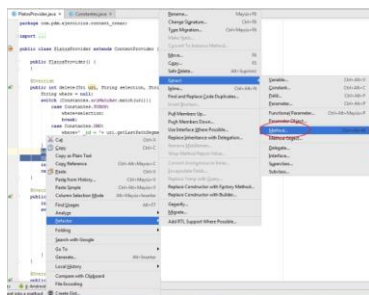
Optimizando código

Ya que estamos escribiendo el código de la clase por primera vez, vamos a optimizarlo. Observamos que hay mucho código que se repite en diferentes métodos, podemos crear métodos propios y utilizarlos:

- En varios métodos llamamos a la BD.

```
MiAdminSqlite admin = MiAdminSqlite.getInstance(getContext(), Constantes.BD NOMBRE, null,
Constantes.BD VERSION);
SQLiteDatabase db = admin.getWritableDatabase();
```

Podríamos tener un método propio con ese código y llamarlo las veces que haga falta. Si utilizamos la herramienta de refactorización de AS nos reemplazará todas las ocurrencias, el nombre que propone el asistente de refactorización es `getSqliteDatabase()` pero obviamente podríamos decidir que fuese otro:



- Haremos lo mismo con el código para obtener la String where, el método se llamará `getString`:

```
private String getString(Uri uri, String selection) {
    String where = null;
    switch (Constantes.uriMatcher.match(uri)) {
        case Constantes.TODOS:
            where=selection;
            break;
        case Constantes.UNO:
            where=" _id = "+ uri.getLastPathSegment();
            break;
    }
    return where;
}
```

- Y ya puestos a optimizar, puedes hacer caso a los warnings y corregir los errores.

Declarar Content provider

Debemos declarar el content provider en nuestro fichero `AndroidManifest.xml` de forma que una vez instalada la aplicación en el dispositivo Android conozca la existencia de dicho recurso.

El valor del atributo `name` es el nombre de la clase que vamos a utilizar como proveedor de contenido.

El valor del atributo `authorities` es la cadena string que se utiliza como URI de donde estará el contenido.

El valor del atributo `exported` indica si los datos son compartidos o no para otras aplicaciones.

También podemos indicar permisos que deben tener las aplicaciones que lo utilizarán

```
<provider
    android:name=".MiProveedor"
    android:authorities="com.pdm.restaurante"
    android:enabled="true"
    android:exported="true">
</provider>
```

AS ya lo ha hecho por nosotros si se ha utilizado el asistente de creación.

IMPORTANTE: Si copias el proyecto para que sea la base de otro proveedor de contenidos y en la clase Constantes cambias el valor de la authority, recuerda cambiarlo también aquí!



Crear el Content Provider

Basta con ejecutar la aplicación una vez y el proveedor de contenidos podrá usarse por otras aplicaciones:



Si se olvida esa primera ejecución y otras aplicaciones intentan acceder a la BD se romperán:

```
FATAL EXCEPTION: main
Process: com.pdm.ejercicios.content_usar, PID: 19932
java.lang.IllegalArgumentException: Unknown URL content://com.pdm.restaurant/platos
O
FATAL EXCEPTION: main
Process: com.pdm.ejercicios.content_usar, PID: 19846
java.lang.NullPointerException: Attempt to invoke interface method 'int android.database.Cursor.getCount()' on a null object reference
```

Después de esa primera ejecución, la aplicación proveedora puede detenerse ya que cada vez que otra aplicación necesite el Content Provider la pondrá en marcha, pero si se desinstala se borrará también el acceso al proveedor de contenidos y las aplicaciones clientes se romperán.

Esta aplicación del ejemplo solo crea el proveedor, pero podría también tener actividades para insertar, actualizar, listar o borrar registros. Para trabajar con la BD podría utilizar métodos vistos en la unidad anterior de BD o acceder ella misma al Content Provider como vamos a ver a continuación que se hace desde una aplicación cliente.

Acceder al Content Provider

En las aplicaciones clientes de contenido utilizar un *content provider* ya existente es muy sencillo, sobre todo comparado con el laborioso proceso de construcción de uno nuevo. Debemos obtener una referencia a un ContentResolver (objeto a través del que realizaremos todas las acciones necesarias sobre el *content provider*) utilizando el método getContentResolver() de la clase Context para obtener la referencia indicada:

```
ContentResolver cr = getContentResolver();           //desde actividades
```

```
ContentResolver cr = getContext().getContentResolver();           //desde fragmentos
```

Una vez obtenida la referencia al content resolver, podremos utilizar sus métodos query(), update(), insert() y delete() para realizar las acciones equivalentes sobre el content provider.

Abre con AS el proyecto P_73_Content_Usar (la aplicación cliente de contenidos) para analizar el código que permite hacer uso de la persistencia de datos creada anterior. La aplicación cuenta con un menú lateral (Navigation Drawer) con las típicas opciones para trabajar con la BD (listar todos los platos, buscar un plato por clave, añadir, etc.).



Clase ConstantesUsar

En nuestro proyecto vamos a tener que referirnos a la URI del Content Provider en numerosas ocasiones. Para evitar errores de tecleo, lo mejor es crearnos una clase con las constantes necesarias como hemos hecho en el proyecto anterior.

Además, debido a que usamos una base de datos SQLite como estructura de almacenamiento, es conveniente (aunque no obligatorio) fijar constantes para los nombres de las columnas que el Content Provider usará para la gestión de los datos.

En la clase BaseColumns existen columnas predefinidas que pueden tener todos los content providers, por ejemplo, la columna `_id` (`BaseColumns._ID="_id"`).

Por ello, para fijar las columnas de nuestro content provider definiremos una clase interna llamada `Column` tomando como base la clase `BaseColumns` y añadiremos nuestras columnas.

```
class ConstantesUsar {
    //Autoridad del Content Provider
    private final static String AUTHORITY = "com.pdm.restaurante";
    // Representación de la tabla a consultar
    private static final String TABLA = "platos";
    // URI de contenido principal
    static final String stringUri = "content://" + AUTHORITY + "/" + TABLA;
    static final Uri CONTENT_URI = Uri.parse(stringUri);
    // Estructura de la tabla
    static class Column implements BaseColumns {
        private Column() {
            // Sin instancias
        }
        //Nombres de columnas
        static final String COL_NOMB = "nombre";
        static final String COL_TIPO = "tipo";
        static final String COL_CANT = "cantidad";
    }
}
```

Por ello además, en cada clase en la que necesitemos usar las constantes definidas para nuestras columnas será conveniente importar

```
import com.pdm.P_73_Content_Usar.ConstantesUsar.Column;
```



Insertar registros

Parte del código InsertarFragment que trabaja con el proveedor de contenidos:

```

ContentValues registro = new ContentValues();
// URI para un registro
registro.put(ConstantsUsar.Column.COL_NOMB, nombre.getText().toString());
registro.put(ConstantsUsar.Column.COL_CANT, Integer.parseInt(cantidad.getText().toString()));
if (primero.isChecked())
    registro.put(ConstantsUsar.Column.COL_TIPO, "P");
if (segundo.isChecked())
    registro.put(ConstantsUsar.Column.COL_TIPO, "S");
if (tercero.isChecked())
    registro.put(ConstantsUsar.Column.COL_TIPO, "T");
Uri nuestroUri = ConstantsUsar.CONTENT_URI;
ContentResolver cr = getContext().getContentResolver();
cr.insert(nuestroUri, registro);
Toast.makeText(getContext(), "Registro insertado", Toast.LENGTH_SHORT).show();

```

Hacemos uso de las constantes para fijar el registro y el nombre del proveedor

Accedemos al Content Provider e insertamos el registro

Obtención de plato buscado por _ID (clave)

Código BuscarPorClaveFragment:

```

ContentResolver cr = getContext().getContentResolver();
// URI para un registro
Uri nuestroUri = Uri.parse(ConstantsUsar.stringUri + "/" + clave.getText().toString());
// Column de la tabla a recuperar que nos interesan ver
String[] projection = { Column.COL_NOMB, Column.COL_TIPO, Column.COL_CANT };
// Hacemos la consulta
Cursor cur = cr.query(nuestroUri,
    projection, // Columnas a devolver
    null, // Condición de la query
    null, // Argumentos variables de la query
    null); // Orden de los resultados

if (cur != null) {
    if (cur.moveToFirst()) {
        tipo.setText(cur.getString(cur.getColumnIndex(Column.COL_TIPO)));
        nombre.setText(cur.getString(cur.getColumnIndex(Column.COL_NOMB)));
        cantidad.setText(cur.getString(cur.getColumnIndex(Column.COL_CANT)));
    } else
        tipo.setText(R.string.noexiste);
    cur.close();
} else
    tipo.setText(R.string.problemas);

```

Obtención de plato buscado por nombre (no clave)

Código BuscarPorNombreFragment:

```

ContentResolver cr = getContext().getContentResolver();
Uri nuestroUri = ConstantsUsar.CONTENT_URI;
// Column de la tabla a recuperar que nos interesan ver
String[] projection = { Column.COL_TIPO, Column.COL_CANT };
// Filtro de los registros
String where = Column.COL_NOMB + " = '" + nombre.getText().toString() + "'";
// Hacemos la consulta
Cursor cur = cr.query(nuestroUri,
    projection, // Column a devolver
    where, // Condición de la query
    null, // Argumentos variables de la query
    null); // Orden de los resultados

if (cur != null) {
    if (cur.moveToFirst()) {
        tipo.setText(cur.getString(cur.getColumnIndex(Column.COL_TIPO)));
        cantidad.setText(cur.getString(cur.getColumnIndex(Column.COL_CANT)));
    } else
        tipo.setText(R.string.noexiste);
    cur.close();
} else
    tipo.setText(R.string.problemas);

```




Obtener todos los registros.

El código del fragmento VerTodosFragment nos visualiza todos los registros de la tabla Platos en un RecyclerView.

Este es el método que lee los datos:

```
private ArrayList<Item> leerDatos() {
    ArrayList<Item> datos = null;
    ContentResolver cr = Objects.requireNonNull(getContext()).getContentResolver();
    Uri nuestroUri = ConstantesUsar.CONTENT_URI;
    // Columnas de la tabla a recuperar;
    // Aunque no necesitamos id, nos facilita si necesitamos añadir listener de click en el item para
    poder buscar el registro
    String[] projection = {ConstantesUsar.Column._ID, ConstantesUsar.Column.COL_NOMB,
        ConstantesUsar.Column.COL_TIPO, ConstantesUsar.Column.COL_CANT};
    // Hacemos la consulta
    Cursor cursor = cr.query(nuestroUri,
        projection, // Columnas a devolver
        null, // Condición de la query
        null, // Argumentos variables de la query
        ConstantesUsar.Column.COL_TIPO); // Orden de los registros
    if (cursor != null) {
        datos = new ArrayList<>(cursor.getCount());
        while (cursor.moveToNext()) {
            datos.add(new Item(Integer.valueOf(cursor.getString(0)), cursor.getString(1),
                cursor.getString(2), Integer.valueOf(cursor.getString(3))));
        }
        cursor.close();
    }
    return datos;
}
```

Borrar el plato de un determinado nombre (no clave)

Primero lo busca para visualizarlo y solicitar a continuación la confirmación de borrado:

```
final ContentResolver cr = getContext().getContentResolver();
final Uri nuestroUri = ConstantesUsar.CONTENT_URI;
// Column de la tabla a recuperar para visualizar en pantalla
String[] projection = {Column.COL_TIPO, Column.COL_CANT};
final String where = Column.COL_NOMB + " = '" + nombre.getText().toString() + "'";
// Hacemos la consulta
Cursor cursor = cr.query(nuestroUri,
    projection, // Column a devolver
    where, // Condición de la query
    null, // Argumentos variables de la query
    null); // Orden de los resultados
if (cursor != null) {
    if (cursor.moveToFirst()) {
        tipo.setText(cursor.getString(cursor.getColumnIndex(Column.COL_TIPO)));
        cantidad.setText(cursor.getString(cursor.getColumnIndex(Column.COL_CANT)));
        borrar.setVisibility(View.VISIBLE);
        borrar.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                cr.delete(nuestroUri, where, null);
                Toast.makeText(getContext(), "Plato borrado", Toast.LENGTH_SHORT).show();
            }
        });
    } else {
        tipo.setText(R.string.noexiste);
        cursor.close();
    }
} else {
    tipo.setText(R.string.problemas);
}
```

Modificar un registro buscado por _ID (clave)

Primero lo busca para visualizarlo y solicitar a continuación los nuevos valores:

```
ContentValues registro = new ContentValues();
registro.put(Column.COL_NOMB, nombre.getText().toString());
registro.put(Column.COL_CANT, Integer.parseInt(cantidad.getText().toString()));
if (primeros.isChecked())
    registro.put(Column.COL_TIPO, "P");
if (segundos.isChecked())
    registro.put(Column.COL_TIPO, "S");
if (terceros.isChecked())
    registro.put(Column.COL_TIPO, "T");
cr.update(nuestroUri, registro, null, null);
Toast.makeText(getContext(), "Registro modificado", Toast.LENGTH_SHORT).show();
```



Listado con filtro de tipo de platos (Primeros y/o Segundos y/o Postres)

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    final View view= inflater.inflate(R.layout.fragment_listado_filtrado, container, false);
    Button button = view.findViewById(R.id.button3);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            CheckBox cb1 = view.findViewById(R.id.checkBox);
            CheckBox cb2 = view.findViewById(R.id.checkBox2);
            CheckBox cb3 = view.findViewById(R.id.checkBox3);
            // Empezamos a construir el filtro de la consulta
            int chequeados = 0;
            List<String> opciones = new ArrayList<>();
            if (cb1.isChecked()) {
                opciones.add("P");
                chequeados++;
            }
            if (cb2.isChecked()) {
                opciones.add("S");
                chequeados++;
            }
            if (cb3.isChecked()) {
                opciones.add("T");
                chequeados++;
            }
            if (chequeados > 0) {
                ContentResolver cr = Objects.requireNonNull(getContext()).getContentResolver();
                Uri nuestroUri = ConstantesUsar.CONTENT_URI;
                // Empezamos a construir la query
                // Column de la tabla a recuperar: incluimos id para reutilizar las clases y layouts
                String[] projection = {ConstantesUsar.Column.ID, ConstantesUsar.Column.COL_NOMB,
                ConstantesUsar.Column.COL_TIPO, ConstantesUsar.Column.COL_CANT};
                // Filtro de la consulta
                String where = "";
                switch (chequeados) {
                    case 1:
                        where = ConstantesUsar.Column.COL_TIPO + " = ?";
                        break;
                    case 2:
                        where = ConstantesUsar.Column.COL_TIPO + " = ? or " +
                        ConstantesUsar.Column.COL_TIPO + " = ?";
                        break;
                    case 3:
                        where = ConstantesUsar.Column.COL_TIPO + " = ? or " +
                        ConstantesUsar.Column.COL_TIPO + " = ? or " + ConstantesUsar.Column.COL_TIPO + " = ?";
                        break;
                }
                // El filtro del método query debe ser String[]
                // por eso realizamos la conversión desde ArrayList opciones
                String[] filtro = new String[opciones.size()];
                opciones.toArray(filtro);
                Cursor cursor = cr.query(nuestroUri,
                    projection, // Column a devolver
                    where, // Condición de la query
                    filtro, // Argumentos variables de la query
                    null); // Orden de los resultados
                ArrayList<Item> datos = null;
                if (cursor!=null){
                    datos = new ArrayList<>(cursor.getCount());
                    while (cursor.moveToNext()){
                        datos.add(new Item(Integer.valueOf(cursor.getString(0)),cursor.getString(1),
                        cursor.getString(2),Integer.valueOf(cursor.getString(3))));
                    }
                    int total = cursor.getCount();
                    if (total == 0)
                        Toast.makeText(getContext(), R.string.noplatos, Toast.LENGTH_LONG).show();
                    cursor.close();
                }
                RecyclerView recyclerView = view.findViewById(R.id.recycler);
                recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
                recyclerView.setAdapter(new MyItemRecyclerViewAdapter(datos));
            } else
                Toast.makeText(getContext(), "Debe escoger al menos un tipo",
                Toast.LENGTH_LONG).show();
        }
    });
    return view;
}
```



Usando Content Provider nativos

Para programar los Content Provider nativos hay que consultar la [documentación oficial](#) para saber qué datos pueden extraerse y qué tipo de permisos hay que establecer en las aplicaciones.

Obviamente las aplicaciones harán uso de esos Content Provider sin necesidad de construirlos.

Caso práctico Content Nativo Llamadas

Supongamos que deseamos una aplicación que nos muestre las llamadas de nuestro teléfono.

Haremos uso del Content Provider nativo [android.provider.CallLog.Calls.CONTENT_URI](#)

Constants		
String	CACHED_NAME	The cached name associated with the phone number, if it exists.
String	CACHED_NUMBER_LABEL	The cached number label, for a custom number type, associated with the phone number, if it exists.
String	CACHED_NUMBER_TYPE	The cached number type (Home, Work, etc) associated with the phone number, if it exists.
String	CONTENT_ITEM_TYPE	The MIME type of a CONTENT_URI sub-directory of a single call.
String	CONTENT_TYPE	The MIME type of CONTENT_URI and CONTENT_FILTER_URI providing a directory of calls.
String	DATE	The date the call occurred, in milliseconds since the epoch Type: INTEGER (long)
String	DEFAULT_SORT_ORDER	The default sort order for this table
String	DURATION	The duration of the call in seconds Type: INTEGER (long)
int	INCOMING_TYPE	Call log type for incoming calls.
String	IS_READ	Whether this item has been read or otherwise consumed by the user.
String	LIMIT_PARAM_KEY	Query parameter used to limit the number of call logs returned.
int	MISSED_TYPE	Call log type for missed calls.
String	NEW	Whether or not the call has been acknowledged Type: INTEGER (boolean)
String	NUMBER	The phone number as the user entered it.
String	OFFSET_PARAM_KEY	Query parameter used to specify the starting record to return.
int	OUTGOING_TYPE	Call log type for outgoing calls.
String	TYPE	The type of the call (incoming, outgoing or missed).

Abre el proyecto P_74_Content_Llamadas con AS para analizar el código.

Para que nuestra aplicación pueda acceder al historial de llamadas del dispositivo tendremos que incluir en el fichero AndroidManifest.xml los permisos:

```
<uses-permission android:name="android.permission.READ_CALL_LOG"/>
```

Veamos la parte de uso del Content Provider en la MainActivity:

```
public class MainActivity extends AppCompatActivity {
    private View layoutMain;
    private Uri miCall;
    private Cursor cur;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        layoutMain = findViewById(R.id.pantallaPrincipal);
        if (ContextCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.READ_CALL_LOG) == PackageManager.PERMISSION_GRANTED) {
            seguir();
        } else {
            pedirPermiso();
        }
    }
}
```

miCall se declara variable global para que poder utilizarla en el método seguir() sin problemas

Se comprueba que se han establecido los permisos en AndroidManifest

```

private void seguir() {
    miCall = android.provider.CallLog.Calls.CONTENT_URI;
    ContentResolver cr = getContentResolver();
    String[] projection = {CallLog.Calls.TYPE, CallLog.Calls.NUMBER, CallLog.Calls.DATE};
    cur = cr.query(miCall,
        projection, // Columnas a devolver
        null, // Condición de la query
        null, // Argumentos variables de la query
        CallLog.Calls.DATE + " desc");
    if (cur != null) {
        ArrayList<Item> datos = new ArrayList<>(cur.getCount());
        if (cur.moveToFirst()) {
            int tipo;
            int posicion;
            long fecha;
            String telefono;
            Drawable imagen = null;
            int colTipo = cur.getColumnIndex(CallLog.Calls.TYPE);
            int colNumero = cur.getColumnIndex(CallLog.Calls.NUMBER);
            int colDate = cur.getColumnIndex(CallLog.Calls.DATE);
            do {
                posicion = cur.getPosition();
                telefono = cur.getString(colNumero);
                tipo = cur.getInt(colTipo);
                if (tipo == CallLog.Calls.INCOMING_TYPE)
                    imagen = ContextCompat.getDrawable(this, android.R.drawable.sym_call_incoming);
                else if (tipo == CallLog.Calls.OUTGOING_TYPE)
                    imagen = ContextCompat.getDrawable(this, android.R.drawable.sym_call_outgoing);
                else if (tipo == CallLog.Calls.MISSED_TYPE)
                    imagen = ContextCompat.getDrawable(this, android.R.drawable.sym_call_missed);
                fecha = cur.getLong(colDate);
                Date fecha_ver = new Date(fecha);
                datos.add(new Item(posicion, telefono, imagen, fecha_ver));
            } while (cur.moveToNext());
            RecyclerView recyclerView = findViewById(R.id.list);
            recyclerView.setLayoutManager(new LinearLayoutManager(getApplicationContext()));
            recyclerView.setAdapter(new MyItemRecyclerViewAdapter(datos));
            cur.close();
        }
        else{
            Toast.makeText(getApplicationContext(),getResources().getString(R.string.sindatos),Toast.LENGTH_LONG).show();
        }
    }
    else{
        Toast.makeText(this,"No hay registro de llamadas. Tablet?",Toast.LENGTH_SHORT).show();
    }
}

private void pedirPermiso() {
    if (ActivityCompat.shouldShowRequestPermissionRationale(this, Manifest.permission.READ_CALL_LOG))
    {
        final Activity activity = this;
        Snackbar.make(layoutMain, "Sin el permiso no puedo seguir.", Snackbar.LENGTH_INDEFINITE)
            .setAction("OK", new View.OnClickListener() {
                @Override
                public void onClick(View view) {
                    ActivityCompat.requestPermissions(activity, new
String[]{Manifest.permission.READ_CALL_LOG}, 123);
                }
            })
            .show();
    }
    else {
        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.READ_CALL_LOG}, 123);
    }
}
}

```



Mejoras

- Cuando se accede al contenido de un Content Provider (o de una BD SQLite) si se quiere tener la certeza de tener los datos actualizados, en vez de hacer las llamadas a los métodos CRUD en el método onCreate (si es desde Activity) / onCreateView (si es Fragment) será mejor hacerlo en el método onResume()
- No debemos olvidar cerrar los cursores en método onStop()
- Hay que optimizar los métodos CRUD de nuestros Content Provider propios para implementar las respuestas sobre URI inválidos (qué pasaría si ejecutamos la aplicación Content_Usar antes de que se haya ejecutado Content_Crear?). En el arranque de cualquier aplicación cliente deberíamos hacer una consulta que nos devolviese un cursor, si el cursor es null es que no existe el Provider (no es lo mismo que que esté vacío `cursor.getCount()==0`)

Prácticas propuestas

Realiza los ejercicios propuestos en el fichero Ejercicios_17_Proveedores_de_contenido.