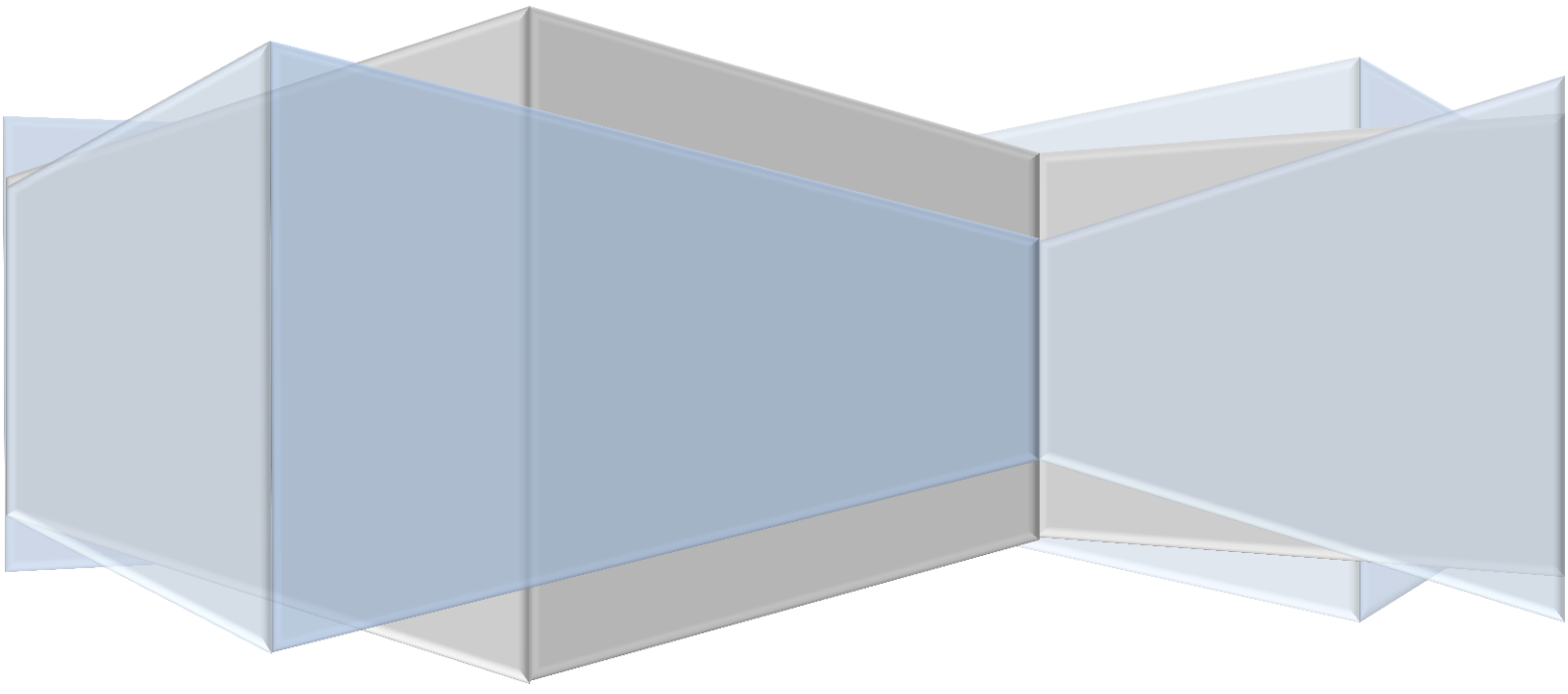
CONTENT PROVIDERS

Programación de Dispositivos Móviles

Ángel Collado García



CONTENT PROVIDERS

1. DEFINICIÓN:

Los ***Content Providers***, se encargan de gestionar el acceso a un conjunto de datos estructurado. Encapsulan los datos y mantienen la seguridad de los datos. Se trata de una interface estándar que conecta los datos de un proceso en otro.

Es un mecanismo proporcionado por la plataforma Android para permitir compartir información entre aplicaciones. Para el buen uso de los Content Providers, debemos saber construir nuevos Content Providers personalizados para nuestras aplicaciones, así como saber utilizar los ya existentes para acceder a los datos publicados por otras aplicaciones.

Cuando se trata de acceder a un dato en un Content Provider utilizaremos el objeto ***ContentResolver*** de la clase ***Context*** para comunicar al Provider con el cliente. El Provider recibe las solicitudes de datos de los clientes, realiza las operaciones y devuelve los resultados pedidos al cliente.

Si no se quiere compartir los datos con otras aplicaciones no es necesario desarrollar un Provider propio. En cambio, si queremos copiar o pegar datos desde aplicaciones externas a la nuestra sí que necesitaremos crear nuestro propio Provider.

Android incluye Providers manejadores de datos como audio, video, imágenes e información personal de contactos. Todos ellos están el paquete ***android.provider***. Cualquier aplicación de Android puede acceder a dicho paquete.

2. FUNCIONES BÁSICAS:

Como se ha comentado en el punto anterior la función principal de los Content Providers es la gestión de acceso a datos en un repositorio central. Son parte de Android y tienen su propia interface de usuario para trabajar con los datos con los que se mueven. Su uso más común es por distintas aplicaciones que acceden a ellos mediante un objeto Provider cliente. Cuando interaccionan juntos los Providers clientes y Content Providers forman la interface estándar de los datos, encargada de la comunicación entre los procesos y del acceso seguro a los datos.

El contenido que presentan los Content Providers a las aplicaciones externas que requieren su servicio, se muestra en tablas, tablas similares a las de bases de datos relacionales. Cada fila representa una instancia de datos de un tipo, recopilada por el Provider y a su vez cada una de las columnas representa un dato individual de los recogidos en la instancia.

Para poder imaginarnos como funciona, uno de los Content Providers disponible en Android es un diccionario. Este, almacena la ortografía de palabras no estándar que el usuario desee almacenar para usar en otra ocasión.

word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

Tabla 1. Diccionario personal

En este ejemplo, cada columna recoge datos específicos de cada palabra recogida en cada una de las filas de la tabla. El ID lo utiliza como clave, pero los Content Providers no están obligados a tener tal, a no ser que los datos que facilite dicho Content Provider sean enlazados a un ListView, eso lo veremos más adelante en las vistas de resultados de los Content Provider.

a. ACCESO A UN CONTENT PROVIDER:

Para que una aplicación tenga acceso a los datos de un Content Provider ha de hacerlo mediante un objeto ***ContentResolver***. Este objeto tiene unos métodos para esas funciones, métodos que requieren de otros en la parte Content Provider. Estos métodos de ContentResolver son los que proporcionan las funciones de almacenamiento persistente, conocido como "CRUD" (Crear, Recuperar, Actualizar y Eliminar).

Luego, aparecen los objetos ContentResolver, para la aplicación cliente, y los objetos ContentProvider para abastecer de datos a la aplicación cliente, manejar las comunicaciones automáticamente y como abstracción entre el repositorio de datos y la apariencia externa, generando para ello unas tablas. Para poder acceder a los Content Provider necesitaremos unos permisos.

Por ejemplo, para poder acceder en la tabla1 a la lista de palabras necesitaremos llamar desde la aplicación con el objeto ContentResolver a su método query, ***ContentResolver.query().***

```
// Consulta de resultados
mCursor =
getContentResolver().query(
        UserDictionary.Words.CONTENT_URI, mProjection, SelectionClause,
        SelectionArgs, mSortOrder);
// Contenido de la tabla palabra
// Columnas a devolver por fila
// Criterios de selección
// Criterios de selección
// El orden de los resultados
```

query() argument	SELECT keyword/parameter	Notes
`Uri`	`FROM table_name`	`Uri` maps to the table in the provider named *table_name*.
`projection`	`col,col,col,...`	`projection` is an array of columns that should be included for each row retrieved.
`selection`	`WHERE col = value`	`selection` specifies the criteria for selecting rows.
`selectionArgs`	(No exact equivalent. Selection arguments replace ? placeholders in the selection clause.)	
`sortOrder`	`ORDER BY col,col,...`	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

Tabla 2. Comparación del método query() y las consultas de sql.

b. URI's DE UN CONTENT PROVIDER

Se trata de la URI que identifica los datos contenidos en un Content Provider. Dentro de esta URI se incluye el nombre del Content y el nombre de la tabla a la que apunta.

Como podemos observar en el ejemplo anterior, la URI es un argumento que pasamos al método query() cuando llamamos desde el cliente al Content. El ContentResolver compara el nombre del Content con el de la URI. Cuando hay coincidencia, entonces, envía los argumentos del query() al Content requerido. Además, gracias a esta URI, el ContentProvider apuntara a la tabla donde se le requiere acceder. Cada Content tiene un camino distinto para cada tabla que recoge.

En el ejemplo anterior vemos el código de la URI (***UserDictionary.Words.CONTENT_URI***), donde ***UserDictionary*** es el nombre del Content Provider y ***Words*** es la ruta a la tabla requerida del Content. Por último ***CONTENT_URI*** es el esquema que identifica toda la sentencia como un contenido URI.

Hay Content que facilitan el acceso a una única tabla, para ello hay que añadir un valor ID unido al código de la sentencia anterior. Por ejemplo para acceder a la tabla de ID4:

```
Uri singleUri =
ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
```

Se suelen usar cuando recuperamos un conjunto de filas, y alguna de estas filas queremos actualizar o eliminar.

La clase Uri y la Uri.Builder, tienen métodos para construir objetos Uri a partir de Strings. Además tenemos la clase ContentUris, con su método withAppendedId(), añade un id a un URI.

3. COMO RECUPERAR DATOS DE UN CONTENT PROVIDER:

Necesitamos hacer consultas asíncronas en subprocesos independientes, para ello se usa la clase CursorLoader, que es un cargador líneas de código. Siempre han de seguirse dos pasos estrictamente, solicitar permiso de lectura y definir el envió de consulta.

a. SOLICITUD DE PERMISO:

Lo primero que necesitamos para conseguir los datos de un Content Provider es pedir permiso, este permiso no se puede solicitar en tiempo de ejecución, hay que solicitarlo en el manifest (<uses-permission>), el elemento y nombre de permiso del Content Provider. Datos que encontraremos en la documentación del Content Provider.

En el ejmplo del diccionario definimos el permiso android.permission.READ_USER_DICTIONARY en el manifest, luego, si queremos acceder al Content Provider, necesitaremos solicitar ese permiso.

b. CONSTRUCCIÓN DE LA CONSULTA:

Este código nos servirá de ejemplo para ver cómo acceder al Content Provider del diccionario del usuario.

```
// Una "projection" define las columnas que devolverá por fila
String[] mProjection =
{
    UserDictionary.Words._ID,    // Para el ID
    UserDictionary.Words.WORD,   // Para la palabra
    UserDictionary.Words.LOCALE  // Para la localidad
};
// Define String para la sentencia
String mSelectionClause = null;
// Inicializa un Array que contendrá los argumentos de selección
String[] mSelectionArgs = {""};
```

Como hemos comentado, son similares los Content Provider a AQL, luego, para entenderlo mejor el siguiente código servirá de ayuda para entender que una consulta de un Content Provider contiene un conjunto de columnas a devolver (***mProjection***), unas sentencias de selección (***mSelectionClause***) y un orden.

Usaremos ***mSelectionClause*** para expresiones lógicas y booleanas. Si especificáramos parámetros reemplazables (?), usaríamos ***mSelectionArgs*** .

Podemos controlar si el usuario no introduce datos, y poner la selección a null, o si introduce algo y entonces establecemos UserDictionary.Words.WORD + " = ?" junto con el primer elemento de los argumentos de selección.

```
// Definimos un Array de un elemento
String[] mSelectionArgs = {""};
// Cojo una palabra UI
mSearchString = mSearchWord.getText().toString();
// Remember to insert code here to check for invalid or malicious
input.
// Si la cadena es vacia, cojo todo
if (TextUtils.isEmpty(mSearchString)) {
    // Ajusto la selección a null y devuelvo todas las palabras
    mSelectionClause = null;
    mSelectionArgs[0] = "";
} else {
    // Contruimos la selec con la palabra introducida
    mSelectionClause = UserDictionary.Words.WORD + " = ?";
    // Movemos la palabra introducida a la lista de argumentos
    mSelectionArgs[0] = mSearchString;
}
// Hacemos la consulta a la table y devolvemos las filas
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,  // La URI de tabla de palabras
    mProjection,                       // Columnas devueltas por fila
    mSelectionClause                   // Entrada null
    mSelectionArgs,                    // Entrada vacia
    mSortOrder);                       // Filas devueltas
// A veces, si si devuelve null si error, otras vece una excepción
if (null == mCursor) {
    /*
     * Insert code here to handle the error. Be sure not to use the
cursor! You may want to
     * call android.util.Log.e() to log this error.
     *
     */
// If the Cursor is empty, the provider found no matches
} else if (mCursor.getCount() < 1) {

    /*
     * Insert code here to notify the user that the search was
unsuccessful. This isn't necessarily
     * an error. You may want to offer the user the option to insert a
new row, or re-type the
     * search term.
     */

} else {
    // Insert code here to do something with the results

}
```

El código anterior sería similar a la consulta de SQL:

```
SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY
word ASC;
```

c. *SEGURIDAD CONTRA ENTRADAS MALICIOSAS:*

```
// Construye una sentencia de elección mediante la concatenación de
entrada del usuario para el nombre de la columna
String mSelectionClause =  "var = " + mUserInput;
```

Hay que evitar ese tipo de selecciones de parámetro reemplazable, para evitar errores por sentencias donde la entrada del usuario no sea tratado como SQL. Hay que usar siempre sentencias de este tipo:

```
// Parámetro reemplazable
String mSelectionClause =  "var = ?";
 // Array de selección de argumentos
String[] selectionArgs = {""};
// Pasamos el elemento introducido al Array
selectionArgs[0] = mUserInput;
```

Esta sentencia, utiliza el parámetro reemplazable ? y el Array de selección de argumentos, es válida para cualquier sentencia, incluso no SQL.

d. *VER RESULTADOS:*

Hemos visto que necesitamos el ContentResolver.query() para consultar, esta sentencia devuelve un cursor con las columnas de la consulta, de las filas consultadas. Este cursor deja leer al azar tanto sus columnas como filas contenidas. Si la consulta no da resultados, si hacemos sobre el cursor Cursor.getCount() y da 0, cursor vacio. Si devuelve null o una excepción, entonces es que ha habido un error interno.

El cursor es una lista de filas, por eso la mejor opción es vincular el cursor con un ListView utilizando ***SimpleCursorAdapter***.

```
// Lista de columnas que se recuperan del cursor y las carga en una
fila de salida
String[] mWordListColumns =
{
    UserDictionary.Words.WORD,   // Constante del nombre de la columna
    UserDictionary.Words.LOCALE  // Constante de la localidad
};
// Lista de vistas IDs recibira las columnas por cada fila
int[] mWordListItems = { R.id.dictWord, R.id.locale};
// Creo SimpleCursorAdapter
mCursorAdapter = new SimpleCursorAdapter(
    getApplicationContext(),               // Context
    R.layout.wordlistrow,                  // Layaout unafila ListView
    mCursor,                               // Resultado
    mWordListColumns,                      // El Array String[]
    mWordListItems,                        // Array<Integer>view IDs
                                           //in the row layout
    0);                                    // Banderas, no usadas
// Pasamos el Adapter al ListView
mWordList.setAdapter(mCursorAdapter);
```

Es necesario, si queremos copiar a un ListView el resultado del cursor, que el cursor tenga una columna ID. Restricción que existe en la mayoría de los ContentProviders.

e. OBTENER RESULTADOS:

Hemos visto como mostrar resultados, pero si los queremos para trabajar con ellos:

```
// Indice de la columna "word"
int index = mCursor.getColumnIndex(UserDictionary.Words.WORD);

// Se ejecuta si el cursor es válido. Si devuelve null error interno.

if (mCursor != null) {
   //Se desplaza a la siguiente fila del cursor.
   //Antes del primer movimiento en el cursor,"Puntero de fila" es -1,
   //Si intenta recuperar datos en esa posición: Excepción.
    while (mCursor.moveToNext()) {
        // Coje el valor de la columna
        newWord = mCursor.getString(index);
        // Insert code here to process the retrieved word.
    }
} else {
    // Aqui el código de ERROR
}
```

Para el cursor usaremos los métodos get, como por ejemplo el getString(), getType()…

4. PERMISOS DE UN CONTENT PROVIDER:

El Content Provider tiene especificados unos permisos que cualquier aplicación externa debe tener para poder acceder a la información que ellos contienen. Hay aplicaciones que no los poseen, pero pueden solicitarlos, estos permisos se ven al instalar la aplicación. Si el Conten Provider no especifica permisos, puede que tampoco otras aplicaciones puedan acceder a el. En Cambio, los Content Provider siempre tiene acceso a lectura y escritura, independientemente de los permisos.

Hemos visto que para acceder a un Content Provider necesitamos android.permission.READ_USER_DICTIONARY o no podrá recuperar datos del Content Provider. En cambio el Content Provider tiene android.permission.WRITE_USER_DICTIONARY que le da permiso de insertar, actualizar y eliminar datos.

Para que esl manifest de la aplicación tenga acceso, cuando se instala por el Package Manager, el usuario debe aprobar todos los permisos de solicitud de aplicación. En caso de no aceptar para la instalación el PackageManager.

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

5. INSERTAR, ACTUALIZAR Y BORRAR DATOS:

a. INSERTAR:

Utilizaremos la siguiente sentencia ***ContentResolver.insert()***, que inserta una fila en el Content Provider y devuelve el contenido URI de esa fila.

```
// Definimos una nueva Uri para el resultado
Uri mNewUri;
...
// Valores a insertar
ContentValues mNewValues = new ContentValues();

// Damos valores the values a cada columna
mNewValues.put(UserDictionary.Words.APP_ID, "example.user");
mNewValues.put(UserDictionary.Words.LOCALE, "en_US");
mNewValues.put(UserDictionary.Words.WORD, "insert");
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");
mNewUri = getContentResolver().insert(
    UserDictionary.Word.CONTENT_URI,   // content URI
    mNewValues                         // valor a insertar
);
```

Los datos los pasamos en un ***ContentValues*** similar al cursor. No es necesario que las columnas sean del mismo tipo de dato, si no queremos valor, ponemos null. ***ContentValues.putNull()***

No es necesario pasarle el ID porque lo genera automáticamente, y siempre es único, por eso lo usan como clave primaria normalmente. El ***newUri*** identifica la fila recién agregada.

```
content://user_dictionary/words/<id_value>
```

El contenido de ID se lo pasamos <id_value> , para obtener la URI de retorno llamamos a ***ContentUris.parseId()***

b. ACTUALIZAR:

Usaremos también el ***ContentValues*** como en insertar, el cliente llamará a ***ContentResolver.update()***, si se quiere eliminar se pasa el valor null

```
// Objeto contain actualize valores
ContentValues mUpdateValues = new ContentValues();
String mSelectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";
String[] mSelectionArgs = {"en_%"};
// variable para el número de filas actualizadas
int mRowsUpdated = 0;
//pasamos los nuevos valores
mUpdateValues.putNull(UserDictionary.Words.LOCALE);
mRowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,   // content URI
    mUpdateValues                       // columnas actualizadas
    mSelectionClause                    // columnas seleccionadas
    mSelectionArgs                      // valor a comparar
);
```

No olvidar tener en cuenta la protección de entradas.

c. *BORRADO:*

El borrado nos permite especificar los criterios de selección para las filas a borrar, devolviendo el número de filas eliminadas.

```
// Criterios de selección par alas filas a borrar
String mSelectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] mSelectionArgs = {"user"};
// Variable para el número de filas borradas
int mRowsDeleted = 0;
// Borrado de filas según criterios
mRowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,   // content URI
    mSelectionClause                    // columna a comparar
    mSelectionArgs                      // valor a comparar
);
```

Al llamar a ***ContentResolver.delete()*** hay que tener en cuenta las restricciones con las entradas

6. TIPOS DE DATOS PROVIDERS:

Los Content Providers tienen multitud de datos diferentes, como enteros, long, float, doublé, el binario grande (BLOB), como una matriz de 64kB. Puede haber tantos tipos de datos como tipos de datos soporta un cursor. Para ver el tipo de dato ya hemos visto que usaremos ***Cursor.getType().***

7. FORMAS ALTERNATIVAS DE ACCESO AL PROVIDER:

Existen tres formas alternativas de acceder a un Content Provider:

- Por lotes, con la clase ContentProviderOperation y con ContentResolver.applyBatch()
- Consultas asíncronas con hilos, usando objetos CursorLoader (Loaders)
- Por intents, no pueden ser enviados de manera directa, pero si solicitudes.

a. *POR LOTES O BATCH:*

Siempre que se vayan a insertar un gran número de filas. Para llevarlo a cabo crea un Array de objetos ContentProviderOperation, enviándolos mediante la llamada ContentResolver.applyBatch(). Se le pasa la autorización y el URI del Content Provider. Cada llamada al método devuelve un Array de resutados.

b. *CON INTENTS:*

Los Intents permiten acceder indirectamente a los datos del Content Provider, incluso sin permisos cuando se trata de una aplicación con muchos permisos o activando una aplicación que permita al usuario trabajar desde ella.

Por ejemplo, una aplicación con permisos de acceso, puede utilizar un intent para mostrar los datos de otra aplicación. Como puede la App Calendario con el intent ACTION_VIEW muestra una fecha de un evento dado, conteniendo la URI, por ejemplo, para la imagen del contacto de un visor de imágenes.

i. PERMISOS TEMPORALES:

Se puede enviar un intent de una App que tiene permisos a otra aplicación, recibiendo otro intent con los resultados URI, que duraran hasta que termine la actividad que recibe datos.
Permisos:

Lectura: FLAG_GRANT_READ_URI_PERMISSION

Escritura: FLAG_GRANT_WRITE_URI_PERMISSION

Un Content Provider se define por su URI en el Manifest, además de android:grantUriPermission, atributo del elemento <proveeedor>, de esta manera con el permiso READ_CONTACTS puedo leer todos los contactos de usuario e información referente.
Estructuración de acceso:

- La App envía un intent, con la acción ACTION_PICK y el content CONTENT_ITEM_TYPE, para esto necesita el método startActivityForResult(), lanzando la activity a primer plano.
- El usuario seleccionara el contacto a actualizar, llamando a setResult(reultCode, intent), para devolver el control a la App llamaremos a finish().
- Para recibir los datos del Content Provider llamamos al método onActivityResult()
- Con la URI resultado, podremos leer los datos sin permisos permanentes

ii. USO DE OTRA APLICACIÓN:

Es la forma más sencilla para poder modificar datos de los que no tengo acceso, es activar una App que si tiene acceso y trabajar desde ella.

Por ejemplo en Calendar con INTENT ACTION_INSERT, permitiendo activar insert.

8. CLASES CONTRACT

Las clases Contract definen constantes, constantes que sirven de ayuda para que las aplicaciones trabajen con las URIs, nombres e intents de los Content Providers. Estas clases no se incluyen automáticamente, será el desarrollador del Provider quien las defina y ponga a disposición de otros desarrolladores. Las clases Contract incluidas en Android están en el paquete android.provider.

En el ejemplo, la clase Contract es UserDictionary, con el nombre (UserDictionary.Words) y la URI (UserDictionary.Words.CONTENT_URI). Por ejemplo:

```
String[] mProjection =
{
    UserDictionary.Words._ID,
    UserDictionary.Words.WORD,
    UserDictionary.Words.LOCALE
};
```

9. TIPO DE REFERENCIA MIME:

Los Content Provider pueden devolver MIME media type o cadenas MIME type o ambas a la vez, se trata de cadenas personalizadas, específicas de cada Provider, la clase MIME hereda de la clase Object que tienen formato tipo y subtipo, por ejemplo:

MIME type text/html. Text sería el tipo y html el subtipo. De tal manera que el Content Provider hace la consultade la URI con el text y devuelve el texto en etiquetas HTML.

El valor dependerá de si es para múltiples filas o simples filas:

```
vnd.android.cursor.dir
vnd.android.cursor.item
```

Lo normal es que los Providers contenidos en Android sean simples, por ejemplo, para añadir un teléfono a la lista:

```
vnd.android.cursor.item/phone_v2
```

Si creáramos un Content Provider propio, lo haríamos de la siguiente forma:

```
content://com.example/Line1
```

El Provider devolvería:

```
vnd.android.cursor.item/vnd.example.line1
```

Muchos de los Content Provider, definen sus propias clases Contract para tipos MIME. Se ayudan de la clase ContactsContarct.RawContacts, que facilita la definición de constantes CONTENT_ITEM_TYPE para los tipos MIME.

3. COMO CREAR UN CONTENT PROVIDER

Como hemos comentado, el Contente Provider administra el acceso a un repositorio central de datos y ofrece disponibilidad de datos proporcionados por otras aplicaciones, por lo que a la hora de desarrollar nuestra aplicación, sería aconsejable ofrecer operaciones que permitan gestionar dichos datos.

Para implementar un Content Provider necesitaremos una o más clases además de una serie de elementos en el archivo Manifiest. Por lo que una de las clases implementa una subclase ContentProvider, la cuál será la interfaz entre el Content Provider y las aplicaciones a desarrollar.

No es sencillo y nos llevara tiempo implementar un Content Provider, por lo que antes de nada necesitaremos enfrentarnos al diseño de nuestras APIs y ver que necesitamos.

1. PLANTEAMIENTO INICIAL:

- ✓ Necesitamos saber si realmente es necesario implementar un Content Provider o no. Los utilizaremos en el caso en el que necesitemos copiar datos o archivos de otras aplicaciones, queremos permitir a los usuarios copiar datos de sus aplicaciones a otras u ofrecer sugerencias de búsqueda personalizadas.
  En el caso de que nuestra aplicación no se comunique con aplicaciones externas, no tendría sentido utilizar un Content Provider.
- ✓ Debemos tener en cuenta para que sirve y como funciona un Content Provider.
- ✓ Es importante saber cómo almacenar los datos, teniendo como opciones:
  - ➢ *Ficheros de datos:* Los datos se almacenarán en un fichero de imágenes, audio o video. Almacenando estos en un espacio privado de la aplicación
  - ➢ *Datos estructurados:* En este caso, los datos se almacenarán en una base de datos (por ejemplo SQLite o almacenamiento por persistencia de datos), Arrays o estructuras similares. De tal manera que los datos serán almacenados en filas y columnas, representando las columnas los datos para una entidad.
- ✓ Debemos definir una implementación de la clase ContentProvider y los métodos requeridos. Ya que esta clase será la interface entre los datos y el resto del sistema Android.
- ✓ Definiremos también la cadena de autorización del Content Provider , contenidos URIs y nombres de las columnas.
- ✓ Añadir implementación de la clase AbstractThreadedSynAdapter, con el fin de sincronizar datos entre el Content Provider y la nube o base de datos.

2. DISEÑO DEL ALMACENAMIENTO DE DATOS:

Un Content Provider es una interface de datos guardados en un formato estructurado. Por lo que antes de crear la base de datos necesitaremos saber cómo almacenar los datos para después poder acceder a ellos o almacenar más. Alguna de las formas de almacenamiento disponibles en Android son:

- ✓ La API de bases de datos de SQLite. La clase SQLiteOpenHelper nos ayuda a crear una base de datos y con la clase SQLiteDatabases podremos acceder a la base de datos.
- ✓ Para el almacenamiento de archivos de datos, Android tiene varias APIs (compartir preferencias, almacenamiento interno, externo, con conexión a red). Si se trata de diseñar un Content Provider que almacena datos relacionados con música o videos, lo mejor será combinar tablas y archivos.
- ✓ Si se trata de bases de datos en red, entonces usaremos las clases java.net y android.net. Pudiendo también sincronizar bases de datos en red con bases de datos locales.

CLAVES PARA EL BUEN DISEÑO:

- ✓ Las tablas siempre tendrán una "Clave Primaria". BaseColumns._ID, que nos sirve para vincularlos a un ListView que siempre requiere un nombre_ID.
- ✓ Si almacenamos imágenes, o grandes archivos, tendremos que pasar los nombres de los usuarios con acceso, para poder acceder ellos desde ContentResolver y sus métodos de acceso.
- ✓ Para datos de tamaño variable, o estructura variable, usaremos los BLOB que son Objetos Binarios Grandes. Un uso de estos sería para protocolos de buffer o estructuras JSON.
- ✓ Los BLOB también pueden sernos útiles para implementar tablas con esquemas independientes. Definiremos las columnas de la clave primaria y del tipo MIME, después las columnas BLOB. Gracias al tipo de MIME conoceremos los datos en las columnas BLOB, lo que permite poder almacenar distintas tipos en las distintas filas de una misma tabla. Un ejemplo de estas tablas independientes son los ContactsContract.Data.

3. DISEÑO DE LOS URIs DEL CONTENT PROVIDER:

El URI es el identificador de los datos en un **ContentProvider**. Contiene:

- El nombre simbólico del **ContentProvider**, el authority.
  Si el nombre del paquete Android es com.example. y el nombre de la App es Name, el authority será com.example.android.Name
- Un nombre que apunta a una tabla o archivo.
- Un _ID, por convenio, todo **ContentProvider** accede a una única fila, y lo hará por el _ID.

Esto le permite determinar la tabla, fila o archivos donde acceder. Con el método addURI () asignaremos un authority. El método match() devuelve el valor entero para un URI.

4. IMPLEMENTACIÓN DE LA CLASE CONTENT PROVIDER

La clase **ContentProvider** tiene seis métodos abstract:

- **query** (), que recupera los datos del **ContentProvider**. Necesitaremos argumentos para seleccionar la tabla a consultar, las filas y columnas a volver, y el orden de clasificación de los resultados. Devolverá los datos con un objeto Cursor, sino devolverá una exception:
  - IllegalArgumentException (si su proveedor recibe una URI no válida)
  - NullPointerException.
- **insert** (), Inserta una nueva fila en el **ContentProvider**. Necesita argumentos para seleccionar la tabla de destino y para obtener los valores de columna a utilizar. Devolverá el contenido URI para la fila insertada.
  - Si un nombre de columna no está en los argumentos de ContentValues.
  - Debe devolver el contenido URI para la nueva fila, para poder construir el nuevo _ID de la fila, para la obtención de esta utilizaremos withAppendedId ().
- **update** (), Actualizará las filas existentes del **ContentProvider**. Utiliza los argumentos para seleccionar la tabla y las filas a actualizar y para obtener los valores de las columnas actualizadas. Devuelve el número de filas actualizadas.
  Usa los valores del ContentValues de insert(), delete() y query().
- **delete** (), Elimina filas del **ContentProvider**. Usa los argumentos para seleccionar la tabla y las filas que desea eliminar. Devuelve el número de registros borrados.
  No tiene por qué borrar físicamente la fila.
- **getType** (), Devuelve el tipo MIME correspondiente a un contenido URI.
- **onCreate** (), Inicializa el **ContentProvider**. El sistema Android llama a este método inmediatamente después de que se crea el **ContentProvider**. Aunque no se crea del todo por qué aplaza la creación de bases de datos y carga de datos hasta que el ContentResolver intenta acceder a él.

Todos estos métodos están disponibles para los ContentResolver, y además, todos ellos, excepto onCreate () puede ser llamados por varios subprocesos a la vez. Se deben evitar tareas largas en el onCreate (). Aplazar tareas de inicialización hasta que sean realmente necesarios.

5. IMPLEMENTACIÓN DE LOS MIME types:

La clase **ContentProvider** devuelve datos del tipo MIME y tiene dos métodos para devolver estos:

- getType (), para cualquier **ContentProvider**.
  Devuelve una cadena en formato MIME que describe el tipo de datos devueltos por el URI.
  Si el URI devuelve una sola fila: android.cursor.item /
  Si el URI devuelve más de una fila: android.cursor.dir /
- getStreamTypes (), si se espera que el **ContentProvider** devuelve archivos.
  Devuelve Array de cadenas de tipos MIME para los archivos devueltos por el **ContentProvider**.

6. IMPLEMENTACIÓN DE LAS CLASES CONTACT:

Una clase de este tipo, es una clase public final, contiene definiciones de constantes para el URI, nombres de columnas, tipos MIME, y otros meta-datos referidos al **ContentProvider**. Es una clase contract entre el **ContentProvider** y otras aplicaciones, garantizando que el acceso.

Una clase que por lo general tiene nombres mnemotécnicos para sus constantes, por lo que los desarrolladores. Se puede compilar en una aplicación desde un archivo .jar que nos proporcione el creados del **ContentProvider.**

7. IMPLEMENTACIÓN DE PERMISOS:

Todas las aplicaciones pueden leer o escribir en un **ContentProvider**, incluso si es privado, porque por defecto no tiene permisos establecidos. Para cambiar esto, establecemos los permisos en el Manifest.xml:

<provider>. com.example.app.provider.permission.READ_PROVIDER.

- Permiso individual de lectura y escritura
  android: permission atribute <provider>
- Permiso que controla la lectura y escritura individualmente
  android:readPermission atribute <provider>
  android:writePermission atribute <provider>
  Estos prevalecen sobre los anteriores
- Permiso de ruta URI
  Leer, escribir, o permiso de lectura / escritura según la URI.
  <path-permission>
- Permiso temporal
  Otorga el acceso temporal a una aplicación, reduciendo el número de permisos de una aplicación en su Manifest.
  <grant-uri-permission>
  Para quitar estos permisos hay que llamar a el método Context.revokeUriPermission().
  El valor de los atributos determina qué parte de su proveedor se hace accesible.
  - Si el atributo se establece en true, el sistema otorgará un permiso temporal para todo, anulando cualquier otro permiso.
  - Si el indicador se establece a false, entonces se debe agregar <grant-uri-permission>
    Cada elemento hijo especificará el contenido URI con acceso temporal.

  Si el atributo android: grantUriPermissions no está presente, se asume que es falsa.

Para delegar el acceso temporal a una aplicación, la intención debe contener el FLAG_GRANT_READ_URI_PERMISSION o FLAG_GRANT_WRITE_URI_PERMISSION, o ambos. Estos se establecen con el método setFlags ().

8. EL ELEMENTO <provider>:

Es necesario tenerlo declarado en el Manifest.xml, sl sistema Android tiene la siguiente información del elemento:

- Authority (android:authorities)
- Los nombres simbólicos
- android:name, clase que implementa el **ContentProvider**
- Los permisos, permisos que otras aplicaciones tiene para acceder al **ContentProvider**
- Los atributos de inicio y control, estos atributos determinan cómo y cuándo el sistema Android comienza el **ContentProvider**, las características y ajustes de tiempo de ejecución:
  - android:enabled: permitiendo que el sistema inicie el **ContentProvider**.
  - android:exported: permitir que otras aplicaciones utilicen el **ContentProvider**
  - android:initOrder: orden en que se debe iniciar el **ContentProvider**
  - android:multiProcess: permitiendo que el sistema inicie el **ContentProvider** en el mismo proceso que el cliente que llama
  - android:process: El nombre del proceso en el que el **ContentProvider** debe ejecutarse
  - android:syncable: indica que los datos del **ContentProvider** son datos en un servidor.

- Atributos informativos: Un icono opcional y una etiqueta para el **ContentProvider**:
  - android: icon
  - android:label: etiqueta informativa

9. INTENTS Y ACCESO A DATOS:

Las aplicaciones pueden acceder a un **ContentProvider** de forma indirecta con intents. La aplicación no llama a cualquiera de los métodos de ContentResolver o ContentProvider. En su lugar, envía un intent que inicia una actividad. La actividad de destino es el encargado de recuperar y mostrar los datos en su interfaz de usuario. Dependiendo de la acción en el intent, la actividad de destino también puede solicitar al usuario hacer modificaciones a los datos del **ContentProvider**. Un intent también puede contener "extras" de datos que muestra el destino de la actividad en la interfaz de usuario; El usuario entonces tiene la opción de cambiar estos datos antes de utilizarlo para modificar los datos en el **ContentProvider**.

Los intents ayudan a garantizar la integridad de los datos. El **ContentProvider** puede depender de tener datos que se insertan, actualizan y eliminan de acuerdo con la lógica definida. El

El Manejo de un intent de entrada que desea modificar los datos de un **ContentProvider** no es diferente al de otros intents.

3. EJEMPLO DE CREACIÓN CONTENT PROVIDER PERSONALIZADO:

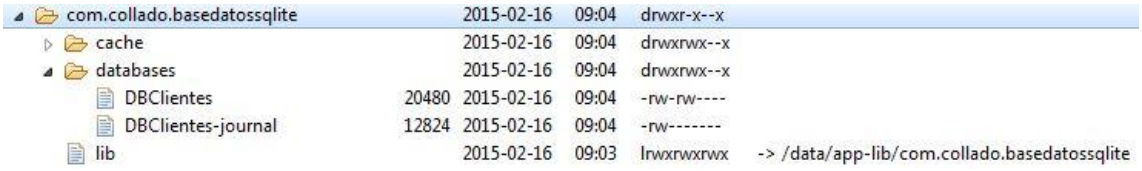
Vamos a poner en práctica lo aprendido y a añadir un **Content Provider** a nuestra aplicación tendremos que:

0) Crear una base de datos estructurada, utilizaré una basada en SQLite (apendice1).
1) Crear una nueva clase que extienda a la clase Android **ContentProvider**.
2) Declarar el nuevo **Content Provider** en nuestro fichero AndroidManifest.xml

Recordamos que el **Content Provider**, será nuestro mecanismo que permita acceder a los datos de nuestra base de datos a terceros de una forma homogénea y a través de una interfaz estandarizada. Abemos que es imprescindible para nuestro **Content Provider** un campo llamado _ID que identifique de forma unívoca un registro del resto.

0. Creación de la BBDD SQLite:

Almacenaré los datos de una serie de clientes (con su _ID, Nombre, Teléfono y email).
Creo una nueva clase (MiSQLiteHelper) que extienda a SQLiteOpenHelper, definiremos las sentencias SQL para crear nuestra tabla de clientes, e implementaremos finalmente los métodos onCreate() y onUpgrade().
Comprobamos que se he hecho correctamente la base de datos:




```
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\TROMPO>cd C:\pEclipse\sdk\platform-tools

C:\pEclipse\sdk\platform-tools>adb devices
List of devices attached
emulator-5554   device


C:\pEclipse\sdk\platform-tools>adb -s emulator-5554 shell
root@generic:/ # sqlite3 /data/data/com.collado.basedatossqlite/databases/DBClie
ntes
/data/data/com.collado.basedatossqlite/databases/DBClientes                    <
SQLite version 3.8.6 2014-08-15 11:46:33
Enter ".help" for usage hints.
sqlite> select * from Clientes;
select * from Clientes;
1¦Cliente1¦900-123-001¦email1@mail.com
2¦Cliente2¦900-123-002¦email2@mail.com
3¦Cliente3¦900-123-003¦email3@mail.com
4¦Cliente4¦900-123-004¦email4@mail.com
5¦Cliente5¦900-123-005¦email5@mail.com
6¦Cliente6¦900-123-006¦email6@mail.com
7¦Cliente7¦900-123-007¦email7@mail.com
8¦Cliente8¦900-123-008¦email8@mail.com
9¦Cliente9¦900-123-009¦email9@mail.com
10¦Cliente10¦900-123-0010¦email10@mail.com
11¦Cliente11¦900-123-0011¦email11@mail.com
12¦Cliente12¦900-123-0012¦email12@mail.com
13¦Cliente13¦900-123-0013¦email13@mail.com
14¦Cliente14¦900-123-0014¦email14@mail.com
15¦Cliente15¦900-123-0015¦email15@mail.com
sqlite>
```

1. Creación de la clase extendida de ContentProvider :

Android, para hacer referencia a un **ContentProvider** lo hará siempre mediante una URI (una cadena de texto, similar a las URL). Para nuestro **ContentProvider** utilizaremos: "content://com.collado.android.contentproviders/clientes". Cada URI se define con sus tres partes:

- El prefijo "content://" que indica que dicho recurso deberá ser tratado por un content provider.
- El identificador del **ContentProvider** , dato que debe ser único (suele usarse el nombre de clase java invertido) com.collado.android.contentproviders.
- La entidad concreta a la que queremos acceder dentro de los datos que proporciona el **ContentProvider** . La tabla de "clientes".

Hay que encapsular la dirección URI en un objeto estático de tipo URI llamado CONTENT_URI. Para ello creamos una clase que extienda de **ContentProvider.** Para facilitar esta tarea Android proporciona una clase llamada UriMatcher, capaz de interpretar determinados patrones en una URI. Esta clase tiene un mediante el método addUri() en el cual indicamos el authority de nuestra URI, el formato de la entidad que estamos solicitando, y el tipo con el que queremos identificar dicho formato

A continuación definir varias constantes con los nombres de las columnas de los datos proporcionados por nuestro **ContentProvider**. Creamos otra clase Clientes que extienda de BaseColumns.
Por último, definir varios atributos privados auxiliares para almacenar el nombre de la base de datos, la versión, y la tabla a la que accederá el **ContentProvider**.

2. Declaración del nuevo ContentProvider :

Métodos abstractos que tendremos que implementar:

- onCreate(),para inicializar todos los recursos necesarios para el funcionamiento del nuevo **ContentProvider**. Apoyándonos SqliteHelper, le pasamos su nombre y versión.
- query(),consultas. Este método recibe como parámetros una URI, una lista de nombres de columna, un criterio de selección, una lista de valores para las variables utilizadas en el criterio anterior, y un criterio de ordenación.
  Devolverá los datos solicitados según la URI indicada y los criterios de selección y ordenación pasados como parámetro.
  Para distinguir entre los dos tipos de URI posibles utilizamos uriMatcher, y su método match(). Para obtener este ID utilizaremos el método getLastPathSegment() del objeto uri que extrae el último elemento de la URI, en este caso el ID del cliente.
  Los resultados se devuelven en forma de Cursor.
- insert(),inserciones. Debe devolver la URI que hace referencia al nuevo registro insertado. Construiremos la nueva URI mediante el método auxiliar ContentUris.withAppendedId() que recibe como parámetro la URI de nuestro **ContentProvider** y el ID del nuevo elemento.
- update(),modificaciones. Devuelven el número de registros afectados
- delete(),eliminaciones. Devuelven el número de registros afectados

- getType(),permitirá conocer el tipo de datos devueltos por el **ContentProvider**. Este tipo de datos se expresará como un MIME Type. Utilizamos una vez más el objeto UriMatcher para determinar el tipo de URI que se está solicitando y en función de ésta devolvemos un tipo MIME u otro porque hay dos tipos MIME distintos para cada entidad del **ContentProvider:**
  - Uno destinado a cuando se devuelve una lista de registros como resultado. "vnd.android.cursor.dir/vnd.xxxxxx"
  - Otro para cuando se devuelve un registro único concreto. "vnd.android.cursor.item/vnd.xxxxxx"

Hemos completado la implementación del nuevo **ContentProvider**. No debemos olvidar declarar el **ContentProvider** en nuestro fichero AndroidManifest.xml de forma que una vez instalada la aplicación en el dispositivo Android conozca la existencia de dicho recurso. Insertaremos el nuevo elemento <provider> dentro de <application> indicando el nombre del **ContentProvider** y su authority.

4. EJEMPLO DE UTILIZACIÓN:

a) UTILIZACIÓN EN NUESTROS CONTENT PROVIDER:

Ahora vamos a hacer uso del **ContentProvider** creado, para acceder a sus datos y a datos del propio sistema Android (logs de llamadas, lista de contactos, agenda telefónica, bandeja de entrada de sms…). El código necesario sería exactamente igual si lo hiciéramos desde la misma aplicación o desde otra distinta.

Lo primero será obtener la referencia a un Content Resolver, el encargado de realizar todas las acciones necesarias sobre el **ContentProvider**. Usaremos el método getContentResolver() desde nuestra actividad para obtener la referencia indicada y así poder usar los métodos de **ContentProvider, como vimos:** query(), update(), insert() y delete().

Vamos a utilizar tres botones en la pantalla principal, para ver cómo funciona:

- Cons → Consulta de todos los clientes.
  - Para ello definimos un array con los nombres de las columnas de la tabla que queremos recuperar (en nuestro caso serán el ID, el nombre, el teléfono y el email).
  - Obtendremos así la referencia al content resolver y utilizaremos su método query() para obtener los resultados en forma de cursor.
  - Recorremos el cursor para procesar los resultados.
- Ins → Para insertar registros nuevos.
  - Rellenamos un objeto ContentValues con los datos del nuevo cliente.
  - Utilizamos el método insert() pasándole la URI del **ContentProvider** en primer lugar, y los datos del nuevo registro como segundo parámetro.
- Elim → Para eliminar todos los registros nuevos insertados con Ins.
  - Utilizando el método delete() del content resolver, indicando como segundo parámetro el criterio de localización de los registros que queremos eliminar (en nuestro caso los de nombre = 'ClienteN', que hemos añadido).

b) UTILIZACIÓN DE CONTENT PROVIDERS GENERADOS POR ANDROID:

Éste mecanismo, es el mismo mecanismo que podemos utilizar para acceder a muchos datos de la propia plataforma Android. Echaremos un vistazo a la documentación oficial del paquete android.provider y lo encontraremos para acceder al historial de llamadas, la agenda de contactos y teléfonos, las bibliotecas multimedia (audio y video), o el historial y la lista de favoritos del navegador.

Por ejemplo, para consultar el historial de llamadas del dispositivo, Android dispone de: content provider android.provider.CallLog. A modo de curiosidad, para emular llamadas entrante, accediendo a la vista del DDMS, en la sección "Emulator Control" veremos un apartado llamado "Telephony Actions". Desde éste, podemos introducir un número de teléfono origen cualquiera y pulsar el botón "Call" para conseguir que nuestro emulador reciba una llamada entrante. Sin aceptar la llamada en elemulador pulsaremos "Hang Up" para teminar la llamada simulando así una llamada perdida.

- Añado un botón más, Calls, para comprobar los registros de llamadas.
- Los nombres de las columnas se almacenan en las constantes Calls.NUMBER (números de teléfono) y Calls.TYPE (si es perdida, recibida o saliente).

Operamos igual que en el apartado a:

- Definimos el array.
- Ejecutamos la consulta llamando al método query().
- Recorremos el cursor obtenido y procesamos los resultados.

Único a diferenciar:

- Para poder acceder al historial de llamadas del dispositivo tendremos que incluir en el fichero AndroidManifest.xml el permiso READ_CONTACTS y READ_CALL_LOG utilizando la cláusula <uses-permission> correspondiente.

APENDICE A

BBDD SQLITE

SQLite es un motor de bases de datos, de pequeño tamaño, no necesita servidor, precisa poca configuración, es transaccional y código libre.

Como ejemplo, nosotros vamos a crear una base de datos llamada BDUsuarios, con una sóla tabla llamada Usuarios que contendrá sólo dos campos: nombre e email. Para ellos, vamos a crear una clase derivada de ***SQLiteOpenHelper*** que llamaremos CreacionSQLite, donde sobrescribiremos los métodos ***onCreate()*** y ***onUpgrade()*** para adaptarlos a la estructura de datos indicada.

```
package com.collado.creacionsqlite;


import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;

public class CreacionSQLite extends SQLiteOpenHelper {
          //SENTENCIA SQL PARA CREAR LA TABLA DE USUARIOS
          String sqlCreate = "CREATE TABLE Usuarios (codigo INTEGER, nombre
TEXT)";
          //CONTEXT ES DE CONTENT
          public CreacionSQLite(Context contexto, String nombre,
                                CursorFactory factory, int version) {
              super(contexto, nombre, factory, version);
          }
          @Override
          public void onCreate(SQLiteDatabase db) {
              //SE EJECUTA LA SENTENCIA SQL DE CREACIÓN DE LA TABLA
              db.execSQL(sqlCreate);      }

          @Override
          public void onUpgrade(SQLiteDatabase db, int versionAnterior, int
versionNueva) {
             //NOTA: POR SIMPLICIDAD DEL EJEMPLO AQUÍ UTILIZAMOS DIRECTAMENTE
LA OPCIÓN DE
              //      ELIMINAR LA TABLA ANTERIOR Y CREARLA DE NUEVO VACÍA
CON EL NUEVO FORMATO.
              //      SIN EMBARGO LO NORMAL SERÁ QUE HAYA QUE MIGRAR DATOS
DE LA TABLA ANTIGUA
              //      A LA NUEVA, POR LO QUE ESTE MÉTODO DEBERÍA SER MÁS
ELABORADO.
              //SE ELIMINA LA VERSIÓN ANTERIOR DE LA TABLA
              db.execSQL("DROP TABLE IF EXISTS Usuarios");
              //SE CREA LA NUEVA VERSIÓN DE LA TABLA
              db.execSQL(sqlCreate);
          }
      }
```

Primero defino una variable llamado sqlCreate donde almacenamos la sentencia SQL para crear una tabla llamada Usuarios con los campos alfanuméricos nombre e email.

El método onCreate() será ejecutado automáticamente por nuestra clase UsuariosDBHelper. Las tareas típicas que deben hacerse en este método serán la creación de todas las tablas necesarias y la inserción de los datos iniciales si son necesarios

Por su parte, el método onUpgrade() se lanzará automáticamente cuando sea necesaria una actualización de la estructura de la base de datos o una conversión de los datos.

Una vez definida nuestra clase helper, la apertura de la base de datos desde nuestra aplicación resulta ser algo de lo más sencillo. Lo primero será crear un objeto de la clase CreacionSQLite al que pasaremos el contexto de la aplicación (en el ejemplo una referencia a la actividad principal), el nombre de la base de datos, un objeto CursorFactory que típicamente no será necesario (en ese caso pasaremos el valor null), y por último la versión de la base de datos que necesitamos. La simple creación de este objeto puede tener varios efectos:

- Si la base de datos ya existe y su versión actual coincide con la solicitada simplemente se realizará la conexión con ella.
- Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método onUpgrade() para convertir la base de datos a la nueva versión y se conectará con la base de datos convertida.
- Si la base de datos no existe, se llamará automáticamente al método onCreate() para crearla y se conectará con la base de datos creada.

Una vez tenemos una referencia al objeto CreacionSQLite, llamaremos a su método getReadableDatabase() o getWritableDatabase() para obtener una referencia a la base de datos, dependiendo si sólo necesitamos consultar los datos o también necesitamos realizar modificaciones, respectivamente.

las bases de datos SQLite creadas por aplicaciones Android utilizando este método se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos

/data/data/com.example.creacionbasedatossqlite/databases/DBUsuarios

Vamos a la perspectiva “DDMS” (Dalvik Debug Monitor Server) de Eclipse y en la solapa “File Explorer” podremos acceder al sistema de archivos del emulador, donde podremos buscar la ruta indicada de la base de datos.

Para comprobar los datos de la BBDD, accedemos de forma remota al emulador a través de su consola de comandos (shell), con el emulador de Android abierto. Abrimos una consola de MS-DOS y utilizaremos la utilidad adb.exe (Android Debug Bridge) situada en la carpeta platform-tools del SDK de Android (en mi caso: c:\Users\Salvador\AppData\Local\Android\android-

sdk\platform-tools\). En primer lugar consultaremos los identificadores de todos los emuladores en ejecución mediante el comando “adb devices“. Se llama “emulator-5554“.

```
C:\pEclipse\sdk\platform-tools>adb devices
List of devices attached
0123456789ABCDEF        device
emulator-5554   device
```

Vamos a acceder a su shell mediante el comando “adb -s identificador-del-emulador shell“. Una vez conectados, ya podemos acceder a nuestra base de datos utilizando el comando sqlite3 pasándole la ruta del fichero. Si todo ha ido bien, podemos escribir las consultas SQL necesarias sobre nuestra base de datos.

```
C:\pEclipse\sdk\platform-tools>adb -s emulator-5554 shell
root@generic:/ # sqlite3 /data/data/com.example.creacionbasedatossqlite/database
s/DBUsuarios
/data/data/com.example.creacionbasedatossqlite/databases/DBUsuarios          <
SQLite version 3.8.6 2014-08-15 11:46:33
Enter ".help" for usage hints.
sqlite> select * from Usuarios;
select * from Usuarios;
1|Usuario1
2|Usuario2
3|Usuario3
4|Usuario4
5|Usuario5
sqlite>
```

/data/data/com.collado.basedatossqlite/databases/DBClientes

BIBLIOGRAFÍA

http://developer.android.com

http://www.sgoliver.net