Content Provider

Profesor: Ana Isabel Vegas

INDICE

1.- INTRODUCCIÓN

Un Content Provider no es más que el mecanismo proporcionado por la plataforma Android para permitir compartir información entre aplicaciones.

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 deberá proporcionar un content provider a través del cual se pueda realizar el acceso a dicha información.

Este mecanismo es utilizado por muchas de las aplicaciones estándar de un dispositivo Android, como por ejemplo la lista de contactos, la aplicación de SMS, o el calendario/agenda.

Esto quiere decir que podríamos acceder a los datos gestionados por estas aplicaciones desde nuestras propias aplicaciones Android haciendo uso de los content providers correspondientes.

2.- CREAR UN CONTENT PROVIDER

El objetivo de este ejemplo es crear una base de datos SQLite con los datos de los clientes. El content provider permitirá el acceso a esos datos desde otras aplicaciones.

Un primer detalle a tener en cuenta es que los registros de datos proporcionados por un content provider deben contar siempre con un campo llamado _ID (no tiene por qué llamarse igual) que los identifique de forma unívoca del resto de registros.

CREAR LA BASE DE DATOS

Lo que haremos será crear una nueva clase que extienda a SQLiteOpenHelper, definiremos las sentencias SQL para crear nuestra tabla de clientes, e implementaremos finalmente los métodos onCreate() y onUpgrade().

```java
public class ClientesSqliteHelper extends SQLiteOpenHelper {

    // Sentencia SQL para crear la tabla de Clientes
    String sqlCreate = "CREATE TABLE Clientes "
            + "(_id INTEGER PRIMARY KEY AUTOINCREMENT, " + " nombre TEXT, "
            + " telefono TEXT, " + " email TEXT )";

    public ClientesSqliteHelper(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);

        // Insertamos 5 clientes de ejemplo
        for (int i = 1; i <= 15; i++) {
            // Generamos los datos de muestra
            String nombre = "Cliente" + i;
            String telefono = "900-123-00" + i;
            String email = "email" + i + "@mail.com";

            // Insertamos los datos en la tabla Clientes
            db.execSQL("INSERT INTO Clientes (nombre, telefono, email) "
                    + "VALUES ('" + nombre + "', '" + telefono + "', '" + email
                    + "')");
        }
    }
```

Gráfico 1. Clase que crea la base de datos

Para añadir un content provider a nuestra aplicación tendremos que:

- Crear una nueva clase que extienda a la clase android ContentProvider.
- Declarar el nuevo content provider en nuestro fichero AndroidManifest.xml

CREAR LA CLASE CONTENT PROVIDER

El acceso a un content provider se realiza siempre mediante una URI. Una URI no es más que una cadena de texto similar a cualquiera de las direcciones web que utilizamos en nuestro navegador

Las direcciones URI de los content providers están formadas por 3 partes.

- 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 utilizar un authority de tipo "nombre de clase java invertido".
- Por último, se indica la entidad concreta a la que queremos acceder dentro de los datos que proporciona el content provider. En nuestro caso será simplemente la tabla de "clientes", ya que será la única existente, pero dado que un content provider puede contener los datos de varias entidades distintas en este último tramo de la URI habrá que especificarlo.

```
// Definición del CONTENT_URI
private static final String uri = "content://com.example.crearcontentprovider/clientes";
```

Gráfico 2. URI del content provider

El siguiente paso será crear la clase que hereda de la clase ContentProvider.

```
public class ClientesProvider extends ContentProvider {
```

Gráfico 3. Clase ClientesProvider

Los métodos abstractos que tendremos que implementar son:

- onCreate()
- query()
- insert()
- update()
- delete()
- getType()

El primero de ellos nos servirá para inicializar todos los recursos necesarios para el funcionamiento del nuevo content provider. Los cuatro siguientes serán los métodos que permitirán acceder a los datos (consulta, inserción, modificación y eliminación, respectivamente) y por último, el método getType() permitirá conocer el tipo de datos devueltos por el content provider

```
@Override
public boolean onCreate() {
    clidbh = new ClientesSqliteHelper(getContext(), BD_NOMBRE, null,
            BD_VERSION);
    return true;
}
```

Gráfico 4. Método onCreate

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

    // Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if (uriMatcher.match(uri) == CLIENTES_ID) {
        where = "_id=" + uri.getLastPathSegment();
    }

    SQLiteDatabase db = clidbh.getWritableDatabase();
    Cursor c = db.query(TABLA_CLIENTES, projection, where, selectionArgs,
            null, null, sortOrder);
    return c;
}
```

Gráfico 5. Método query

```
@Override
public Uri insert(Uri uri, ContentValues values) {

    long regId = 1;
    SQLiteDatabase db = clidbh.getWritableDatabase();
    regId = db.insert(TABLA_CLIENTES, null, values);
    Uri newUri = ContentUris.withAppendedId(CONTENT_URI, regId);
    return newUri;
}
```

Gráfico 6. Método insert

```java
@Override
public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
    int cont;

    // Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if (uriMatcher.match(uri) == CLIENTES_ID) {
        where = "_id=" + uri.getLastPathSegment();
    }
    SQLiteDatabase db = clidbh.getWritableDatabase();
    cont = db.update(TABLA_CLIENTES, values, where, selectionArgs);
    return cont;
}
```

Gráfico 7. Método update

```java
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    int cont;

    // Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if (uriMatcher.match(uri) == CLIENTES_ID) {
        where = "_id=" + uri.getLastPathSegment();
    }
    SQLiteDatabase db = clidbh.getWritableDatabase();
    cont = db.delete(TABLA_CLIENTES, where, selectionArgs);
    return cont;
}
```

Gráfico 8. Método delete

Nos queda implementar el método getType(). Este método se utiliza para identificar el tipo de datos que devuelve el content provider.

Este tipo de datos se expresará como un MIME Type, al igual que hacen los navegadores web para determinar el tipo de datos que están recibiendo tras una petición a un servidor. Identificar el tipo de datos que devuelve un content provider ayudará por ejemplo a Android a determinar qué aplicaciones son capaces de procesar dichos datos.

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.

```java
@Override
public String getType(Uri uri) {

    int match = uriMatcher.match(uri);

    switch (match) {
    case CLIENTES:
        return "vnd.android.cursor.dir/vnd.example.cliente";
    case CLIENTES_ID:
        return "vnd.android.cursor.item/vnd.example.cliente";
    default:
        return null;
    }
}
```

Gráfico 9. Método getType

A continuación vamos a definir varias constantes con los nombres de las columnas de los datos proporcionados por nuestro content provider. Existen columnas predefinidas que deben tener todos los content providers, por ejemplo la columna _ID. Estas columnas estándar están definidas en la clase BaseColumns, por lo que para añadir las nuevas columnas de nuestro content provider definiremos una clase interna pública tomando como base la clase BaseColumns y añadiremos nuestras nuevas columnas.

```java
// Clase interna para declarar las constantes de columna
public static final class Clientes implements BaseColumns {
    private Clientes() {
    }

    // Nombres de columnas
    public static final String COL_NOMBRE = "nombre";
    public static final String COL_TELEFONO = "telefono";
    public static final String COL_EMAIL = "email";
}
```

Gráfico 10. Definiendo las columnas

Por último, vamos a definir varios atributos privados auxiliares para almacenar el nombre de la base de datos, la versión, y la tabla a la que accederá nuestro content provider.

```
// Base de datos
private ClientesSqliteHelper clidbh;
private static final String BD_NOMBRE = "DBClientes";
private static final int BD_VERSION = 1;
private static final String TABLA_CLIENTES = "Clientes";
```

Gráfico 11. Propiedades de la Base de datos

La primera tarea que nuestro content provider deberá hacer cuando se acceda a él será interpretar la URI utilizada. Para facilitar esta tarea Android proporciona una clase llamada UriMatcher, capaz de interpretar determinados patrones en una URI. Esto nos será útil para determinar por ejemplo si una URI hace referencia a una tabla genérica o a un registro concreto a través de su ID.

```
// Inicializamos el UriMatcher
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI("com.example.crearcontentprovider", "clientes",
            CLIENTES);
    uriMatcher.addURI("com.example.crearcontentprovider",
            "clientes/#", CLIENTES_ID);
}
```

Gráfico 12. Inicializamos uriMatcher

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. Para ello, bastará insertar un nuevo elemento <provider> dentro de <application> indicando el nombre del content provider y su authority.

```
<provider android:name="ClientesProvider"
          android:authorities="com.example.crearcontentprovider"/>
```

Gráfico 13. Declaración del content provider en AndroidManifest.xml

3.- UTILIZAR EL CONTENT PROVIDER

Para comenzar, debemos obtener una referencia a un Content Resolver, objeto a través del que realizaremos todas las acciones necesarias sobre el content provider. Esto es tan fácil como utilizar el método getContentResolver() desde nuestra actividad para obtener la referencia indicada. 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.

```
btnInsertar.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View arg0) {
        ContentValues values = new ContentValues();

        values.put(Clientes.COL_NOMBRE, "ClienteN");
        values.put(Clientes.COL_TELEFONO, "999111222");
        values.put(Clientes.COL_EMAIL, "nuevo@email.com");

        ContentResolver cr = getContentResolver();

        cr.insert(ClientesProvider.CONTENT_URI, values);
    }
});
```

Gráfico 14. Insertar clientes

```
btnEliminar.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View arg0) {
        ContentResolver cr = getContentResolver();

        cr.delete(ClientesProvider.CONTENT_URI, Clientes.COL_NOMBRE
                + " = 'ClienteN'", null);
    }
});
```

Gráfico 15. Borrar clientes

```
btnConsultar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        // Columnas de la tabla a recuperar
        String[] projection = new String[] { Clientes._ID,
                Clientes.COL_NOMBRE, Clientes.COL_TELEFONO,
                Clientes.COL_EMAIL };

        Uri clientesUri = ClientesProvider.CONTENT_URI;
        ContentResolver cr = getContentResolver();

        // Hacemos la consulta
        Cursor cur = cr.query(clientesUri, projection, // Columnas a devolver
                null, // Condición de la query
                null, // Argumentos variables de la query
                null); // Orden de los resultados

        if (cur.moveToFirst()) {
            String nombre;
            String telefono;
            String email;

            int colNombre = cur.getColumnIndex(Clientes.COL_NOMBRE);
            int colTelefono = cur.getColumnIndex(Clientes.COL_TELEFONO);
            int colEmail = cur.getColumnIndex(Clientes.COL_EMAIL);

            txtResultados.setText("");
            do {
                nombre = cur.getString(colNombre);
                telefono = cur.getString(colTelefono);
                email = cur.getString(colEmail);
                txtResultados.append(nombre + " - " + telefono + " - "
                        + email + "\n");

            } while (cur.moveToNext());
        }
```

Gráfico 16. Consultar clientes

CONTENT PROVIDER DE ANDROID

Con esto, hemos visto lo sencillo que resulta acceder a los datos proporcionados por un content provider.

Pues bien, éste es el mismo mecanismo que podemos utilizar para acceder a muchos datos de la propia plataforma Android. En la documentación oficial del

paquete android.provider podemos consultar los datos que tenemos disponibles a través de este mecanismo, entre ellos encontramos por ejemplo: el 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 ver un ejemplo de acceso a este tipo de datos, vamos a realizar una consulta al historial de llamadas del dispositivo, para lo que accederemos al content provider android.provider.CallLog.

En primer lugar vamos a registrar varias llamadas en el emulador de Android, de forma que los resultados de la consulta al historial de llamadas contenga algunos registros. Haremos por ejemplo varias llamadas salientes desde el emulador y simularemos varias llamadas entrantes desde el DDMS. Las primeras son sencillas, simplemente ve al emulador, accede al teléfono, marca y descuelga igual que lo harías en un dispositivo físico.

Y para emular llamadas entrantes podremos hacerlo una vez más desde Eclipse, accediendo a la vista del DDMS. En esta vista, si accedemos a 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 el emulador pulsaremos "Hang Up" para terminar la llamada simulando así una llamada perdida.

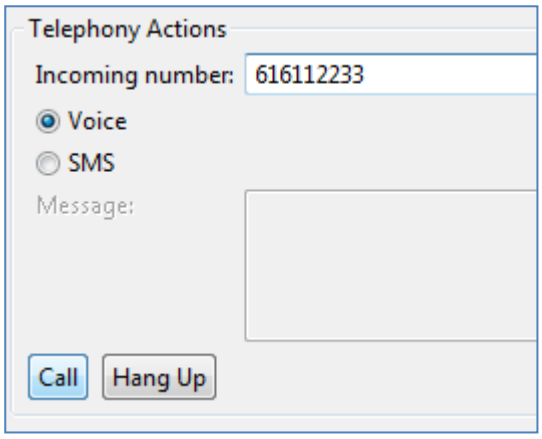

Gráfico 17. Llamadas al emulador

Hecho esto, procedemos a realizar la consulta al historial de llamadas utilizando el content provider indicado.

```
btnLlamadas.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View arg0) {
        String[] projection = new String[] { Calls.TYPE, Calls.NUMBER };
        Uri llamadasUri = Calls.CONTENT_URI;
        ContentResolver cr = getContentResolver();
        Cursor cur = cr.query(llamadasUri, projection, // Columnas a devolver
                null, // Condición de la query
                null, // Argumentos variables de la query
                null); // Orden de los resultados

        if (cur.moveToFirst()) {
            int tipo;
            String tipoLlamada = "";
            String telefono;

            int colTipo = cur.getColumnIndex(Calls.TYPE);
            int colTelefono = cur.getColumnIndex(Calls.NUMBER);
            txtResultados.setText("");
            do {
                tipo = cur.getInt(colTipo);
                telefono = cur.getString(colTelefono);
                if (tipo == Calls.INCOMING_TYPE)
                    tipoLlamada = "ENTRADA";
                else if (tipo == Calls.OUTGOING_TYPE)
                    tipoLlamada = "SALIDA";
                else if (tipo == Calls.MISSED_TYPE)
                    tipoLlamada = "PERDIDA";

                txtResultados.append(tipoLlamada + " - " + telefono
                        + "\n");

            } while (cur.moveToNext());
        }
    }
```

Gráfico 18. Recuperar llamadas

No se nos debe de olvidar agregar los permisos oportuno en el archivo AndroidManifest.xml

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

Gráfico 19. Permisos en AndroidManifest.xml

Tras ejecutar la aplicación y pulsar el botón de llamadas esto es lo que veríamos.

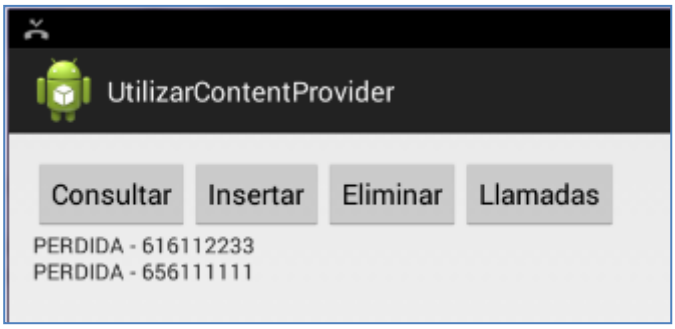

Gráfico 20. Llamadas

ÍNDICE DE GRÁFICOS