Desarrollo Android

Arkaitz Garro

Master en desarrollo de Apps Móviles

Bases de datos

SQLite

SQLite

Es un sistema gestor de base de datos relacional

Ha sido implementado como una librería de C e incluida en Android

Al ser implementada como una librería, cada BD es integrada dentro de la aplicación, lo que elimina las dependencias externas, minimiza la latencia y facilita las transacciones

Content Values y Cursor

`ContentValue` es un objeto que representa una fila en formato NVP. Es utilizado para insertar valores en tablas.

Las consultas a la BD devuelven un `Cursor`. En lugar de extraer los resultados y obtener una copia, el cursor es un puntero a los valores (fila).

Cursor

Cursor incluye estas funciones para la navegación:

`moveToFirst`: mueve el cursor a la primera fila de resultados.

`moveToNext` mueve el cursor a la siguiente fila.

`moveToPrevious`: mueve el cursor a la anterior fila.

`getCount`: número de filas en el resultado.

`getColumnIndexOrThrow`: devuelve el índice la columna que coincide con el nombre.

`getColumnName`: nombre de la columna basado en el índice.

`getColumnNames`: todos los nombres de columnas en el cursor.

`moveToPosition`: mueve el cursor a la posición especificada.

`getPosition`: devuelve la posición actual.

SQLiteOpenHelper

SQLiteOpenHelper es una clase abstracta utilizada para implementar las mejores prácticas con respecto al uso de BD

Permite ocultar la lógica y asegura que cada operación es ejecutada de manera eficiente

Es una buena práctica esperar a crear y abrir BD hasta el momento que sea necesario, ya que son procesos muy costosos.

Debemos asegurar que todas las transacciones se realizan de manera asíncrona, para no afectar a la experiencia de usuario

```
private static class HoardDBOpenHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "myDatabase.db";
    private static final String DATABASE_TABLE = "GoldHoards";
    private static final int DATABASE_VERSION = 1;

    // SQL Statement to create a new database.
    private static final String DATABASE_CREATE = "create table ... ;";

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

    // Called when no db exists in disk
    @Override
    public void onCreate(SQLiteDatabase db) {
      db.execSQL(DATABASE_CREATE);
    }

    // Called when there is a database version mismatch
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      // Simplest case is to drop the old table and create a new one.
      db.execSQL("DROP TABLE IF EXISTS " + DATABASE_TABLE);
      // Create a new one.
      onCreate(db);
    }
}
```

SQLiteOpenHelper

Para acceder a la BD disponemos de `getWritableDatabase` y `getReadableDatabase`, para obtener acceso de escritura o sólo lectura

Si la BD no existe, se crea en este momento (`onCreate`), y si la versión ha cambiado, se actualiza (`onUpgrade`). En cualquier caso, siempre se devuelve la BD abierta y cacheada.

En general, es una buena práctica abrir la BD en modo escritura, ya es la única manera de poder actualizarla.

En caso de fallo, abrir la BD en modo sólo lectura

Consultar una BD

Para realizar una consulta a la BD, disponemos del método `query`, con los siguientes argumentos:

[Opcional] Un `Boolean` que especifica si el resultado contiene valores únicos.

El nombre de la tabla a consultar.

El nombre de las columnas a incluir en el resultado.

Una cláusula `where`.

Una cláusula `groupBy`.

Una cláusula `having`.

Un `String` que describe el orden del resultados.

Un `String` que describe el número máximo de resultados

```
// Specify the result column projection. Return the minimum set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
  KEY_ID, KEY_GOLD_HOARD_ACCESSIBLE_COLUMN, KEY_GOLD_HOARDED_COLUMN };

// Specify the where clause that will limit our results.
String where = KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + "=" + 1;

// Replace these with valid SQL statements as necessary.
String whereArgs[] = null;
String groupBy = null;
String having = null;
String order = null;

SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
Cursor cursor = db.query(HoardDBOpenHelper.DATABASE_TABLE,
                         result_columns, where,
                         whereArgs, groupBy, having, order);

return cursor;
```

Obtener valores del Cursor

El primer paso, es colocar el cursor en la posición deseada, utilizando cualquiera de sus métodos

Una vez en la fila deseada, obtener el valor con los métodos `get<type>(int columnindex)`

```
Cursor cursor = getAccessibleHoard();
float totalHoard = 0f;

// Find the index to the column(s) being used.
int GOLD_HOARDED_COLUMN_INDEX =
    cursor.getColumnIndexOrThrow(KEY_GOLD_HOARDED_COLUMN);

// Iterate over the cursors rows.
while (cursor.moveToNext()) {
    float hoard = cursor.getFloat(GOLD_HOARDED_COLUMN_INDEX);
    totalHoard += hoard;
}

cursor.close();
return totalHoard;
```

Añadir, actualizar y eliminar

La clase SQLiteDatabase dispone de métodos `insert`, `delete` y `update` que encapsulas las sentencias necesarias para ejecutar estas acciones.

Además, existe el método `executeSQL` que permite ejecutar sentencias SQL directamente.

Siempre que se actualicen los datos, es necesario volver a cargar los cursores (con una nueva query)

Añadir datos

```
public void addNewHoard(String hoardName, float hoardValue, boolean
hoardAccessible) {
    // Create a new row of values to insert.
    ContentValues newValues = new ContentValues();

    // Assign values for each row.
    newValues.put(KEY_GOLD_HOARD_NAME_COLUMN, hoardName);
    newValues.put(KEY_GOLD_HOARDED_COLUMN, hoardValue);
    newValues.put(KEY_GOLD_HOARD_ACCESSIBLE_COLUMN, hoardAccessible);
    // [ ... Repeat for each column / value pair ... ]

    // Insert the row into your table
    SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
    db.insert(HoardDBOpenHelper.DATABASE_TABLE, null, newValues);
}
```

Editar datos

```
public void updateHoardValue(int hoardId, float newHoardValue) {
    // Create the updated row Content Values.
    ContentValues updatedValues = new ContentValues();

    // Assign values for each row.
    updatedValues.put(KEY_GOLD_HOARDED_COLUMN, newHoardValue);
    // [ ... Repeat for each column to update ... ]

    // Specify a where clause the defines which rows should be
    // updated. Specify where arguments as necessary.
    String where = KEY_ID + "=" + hoardId;
    String whereArgs[] = null;

    // Update the row with the specified index with the new values.
    SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
    db.update(HoardDBOpenHelper.DATABASE_TABLE, updatedValues,
              where, whereArgs);
}
```

Eliminar datos

```
public void deleteEmptyHoards() {
    // Specify a where clause that determines which row(s) to delete.
    // Specify where arguments as necessary.
    String where = KEY_GOLD_HOARDED_COLUMN + "=" + 0;
    String whereArgs[] = null;

    // Delete the rows that match the where clause.
    SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
    db.delete(HoardDBOpenHelper.DATABASE_TABLE, where, whereArgs);
}
```

Ejercicio

Tareas asíncronas

Utilizando Threads

Todos los componentes de Android, se ejecutan en el proceso principal, por lo tanto, cualquier proceso en cualquier componente puede bloquear el resto de componentes.

En Android, las actividades que no responden a un evento en entrada en 5 segundos, son consideradas como bloqueadas.

Es importante utilizar Threads para todas las operaciones que impliquen tiempo de procesado, para no interferir con la UI.

Consultas asíncronas

Es importante que las operaciones con BD, Content Provider o accesos a Internet se realicen de manera asíncrona

Puede ser complicado sincronizar un proceso en segundo plano con la UI

Para simplificar este proceso, Android ha incluido la clase `Loader`, facilitando el acceso a los datos

Loader

Los elementos Loader está disponibles en todas las actividades y fragmentos a través de `LoadManager`

Permiten el acceso asíncrono a los datos, sin bloquear el proceso principal (UI)

Monitoriza el origen de los datos, actualizando la vista cuando se producen cambios

Es especialmente interesante la clase `CursorLoader`, ya que permite realizar operaciones asíncronas contra Content Providers

Componentes

`LoaderManager:` asociado a la actividad o fragmento, gestiona los **`Loader`** existentes, incluso reconectando con el origen de datos tras cambios de configuración.

`LoaderManager.LoaderCallbacks:` interfaz a implementar en la actividad o fragmento para iniciar/parar los **`Loader`**.

`Loader:` clase base que ejecuta las tareas en segundo plano. Utilizaremos subclases de ésta, como **`CursorLoader`** o **`AsyncTaskLoader`**.

Cómo utilizar Loaders

Para utilizar Loaders, generalmente necesitaremos los siguientes elementos:

- **`Activity o Fragment:`** procesos desde donde se van a lanzar las tareas en segundo plano.
- Una instancia de **`LoaderManager`**
- Un **`CursorLoader,`** encargado de obtener los datos de una BD o un **`ContentProvider`**
- Implementar los métodos **`de LoaderManager.LoaderCallbacks`**
- Un adaptador para mostrar los datos en la vista, como por ejemplo un **`SimpleCursorAdapter`**
- Un proveedor de datos, BD o **`ContentProvider`**

```
public class EarthquakeListFragment extends ListFragment implements
LoaderManager.LoaderCallbacks<Cursor> {

  SimpleCursorAdapter adapter;

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    adapter = new SimpleCursorAdapter(...);

    getLoaderManager().initLoader(id, null, this);
  }

  public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    CursorLoader loader = new CursorLoader(getActivity(),
      EarthquakeProvider.CONTENT_URI, projection, where, null, null);

    return loader;
  }

  public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
  }

  public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
  }
}
```

Tareas sencillas: AsyncTask

`AsyncTask` implementa uno de los mejores patrones para ejecutar Threads y sincronizarlos con la UI.

AsyncTask maneja la creación, gestión y sincronización, permitiendo ejecutar tareas y actualizar la UI cuando éstas se completan.

Es una solución perfecta para tareas cortas que tiene un reflejo en la UI.

Si una actividad es reiniciada, todas las AsyncTask son terminadas.

La tarea es definida por tres argumentos, `Params`, `Progress`, Result y por cuatro métodos `onPreExecute`, `doInBackground`, `onProgressUpdate` y `onPostExecute`.

Tareas sencillas: AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}

new DownloadFilesTask().execute(url1, url2, url3);
```

Ejercicio

Content Provider

Content Provider

Content Provider nos ofrece una interfaz para publicar datos que serán consumidos por un Content Resolver.

Permiten desacoplar componentes de la aplicación, así como crear un mecanismo genérico para compartir datos.

```
public class MyContentProvider extends ContentProvider
```

Registrar Content provider

Al igual que las actividades y servicios, un Content Provider debe ser registrado en el manifiesto.

Es importante definir en el atributo `authorities` el URI, que será utilizado por los Content Resolver para consultar la BD

Este atributo debe ser único, por lo que es una buena práctica que tenga la siguiente sintaxis

```
com.<CompanyName>.provider.<ApplicationName>
```

```
<provider android:name=”.MyContentProvider”
          android:authorities=”com.paad.provider.skeletondatabaseprovider”/>
```

Publicar el URI

Todo Content Provider debe exponer su authority de manera pública. Debe ser el path al contenido principal

```
public static final Uri CONTENT_URI = Uri.parse(“content://
  com.paad.provider.skeletondatabaseprovider/elements”);
```

Esta URI será utilizada por los Content Resolver para realizar las consultas.

```
content://com.paad.provider.skeletondatabaseprovider/elements/5
```

La mejor manera de distinguir que contenido nos están solicitando es hacer uso de la clase `UriMatcher`

URI Matcher

```
//Create the constants used to differentiate between the different URI
//requests.
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;

private static final UriMatcher uriMatcher;

//Populate the UriMatcher object, where a URI ending in
//'elements' will correspond to a request for all items,
//and 'elements/[rowID]' represents a single row.
static {
   uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
   uriMatcher.addURI("com.paad.provider.skeletondatabaseprovider",
                     "elements", ALLROWS);
   uriMatcher.addURI("com.paad.provider.skeletondatabaseprovider",
                     "elements/#", SINGLE_ROW);
}

switch (uriMatcher.match(uri)) {
    case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        queryBuilder.appendWhere(KEY_ID + "=" + rowID);
      default: break;
}
```

Crear Content Provider

Para iniciar el origen de datos al que se desea acceder, sobreescribimos el método `onCreate` del Content Provider.

Este método es llamado al iniciar la aplicación, para todos los Content Provider existentes.

Lo lógico es utilizar SQLiteOpenHelper como hemos visto antes, para diferir la creación de la BD hasta que sea necesario.

```
private MySQLiteOpenHelper myOpenHelper;

@Override
public boolean onCreate() {
  // Construct the underlying database.
  myOpenHelper = new MySQLiteOpenHelper(getContext(),
  MySQLiteOpenHelper.DATABASE_NAME, null, MySQLiteOpenHelper.DATABASE_VERSION);

  return true;
}
```

Consultas a Content Provider

Para recibir consultas en nuestro Content Provider, debemos implementar los métodos `query` y `getType`. Los Content Resolver utilizan estos métodos para acceder a la información

De esta manera, mantenemos una interfaz coherente y no desvelamos la estructura de la BD.

La clave está en definir correctamente las URI que dan acceso a nuestro contenido.

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

    SQLiteDatabase db;
    try {
      db = myOpenHelper.getWritableDatabase();
    } catch (SQLiteException ex) {
      db = myOpenHelper.getReadableDatabase();
    }

    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();

    // If this is a row query, limit the result set to the passed in row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        queryBuilder.appendWhere(KEY_ID + "=" + rowID);
      default: break;
    }

    queryBuilder.setTables(MySQLiteOpenHelper.DATABASE_TABLE);

    // Execute the query.
    Cursor cursor = queryBuilder.query(db, projection, selection,
        selectionArgs, groupBy, having, sortOrder);

    return cursor;
}
```

Content Provider MIME Type

Además de implementar las consultas, debemos especificar el tipo de datos que devuelve cada una de ellas, sobreescribiendo el método `getType`

```
@Override
public String getType(Uri uri) {
    // Return a string that identifies the MIME type
    // for a Content Provider URI
    switch (uriMatcher.match(uri)) {
      case ALLROWS:
        return "vnd.android.cursor.dir/vnd.paad.provider.elemental";
      case SINGLE_ROW:
        return "vnd.android.cursor.item/vnd.paad.provider.elemental";
      default:
        throw new IllegalArgumentException("Unsupported URI: " + uri);
    }
}
```

Transacciones

Tal y como ocurre con las consultas, podemos implementar las operaciones de añadir, modificar y eliminar elementos.

Cuando realizamos transacciones de este tipo, es una buena práctica notificar al Content Resolver que los datos a los que apunta han sido modificados.

```
@Override
public Uri insert(Uri uri, ContentValues values) {
    // Open a read / write database to support the transaction.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();

    // To add empty rows to your database by passing in an empty
    // Content Values object you must use the null column hack
    // parameter to specify the name of the column that can be
    // set to null.
    String nullColumnHack = null;

    // Insert the values into the table
    long id = db.insert(MySQLiteOpenHelper.DATABASE_TABLE,
        nullColumnHack, values);

    // Construct and return the URI of the newly inserted row.
    if (id > -1) {
      // Construct and return the URI of the newly inserted row.
      Uri insertedId = ContentUris.withAppendedId(CONTENT_URI, id);

      // Notify any observers of the change in the data set.
      getContext().getContentResolver().notifyChange(insertedId, null);

      return insertedId;
    }
    else
      return null;
}
```

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

    // Open a read / write database to support the transaction.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();

    // If this is a row URI, limit the deletion to the specified row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        selection = KEY_ID + "=" + rowID
            + (!TextUtils.isEmpty(selection) ?
              " AND (" + selection + ')' : "");
      default: break;
    }

    // Perform the update.
    int updateCount = db.update(MySQLiteOpenHelper.DATABASE_TABLE,
      values, selection, selectionArgs);

    // Notify any observers of the change in the data set.
    getContext().getContentResolver().notifyChange(uri, null);

    return updateCount;
}
```

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    // Open a read / write database to support the transaction.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();

    // If this is a row URI, limit the deletion to the specified row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        selection = KEY_ID + "=" + rowID
            + (!TextUtils.isEmpty(selection) ?
              " AND (" + selection + ')' : "");
      default: break;
    }

    // To delete all rows and return a value pass in "1".
    if (selection == null)
      selection = "1";

    // Perform the deletion.
    int deleteCount = db.delete(MySQLiteOpenHelper.DATABASE_TABLE,
      selection, selectionArgs);

    // Notify any observers of the change in the data set.
    getContext().getContentResolver().notifyChange(uri, null);

    // Return the number of deleted items.
    return deleteCount;
}
```

Content Resolver

Todas las aplicaciones contienen una instancia de `ContentResolver`.

Es la clase utilizada para acceder a los datos a través de los Content Provider.

Un Content Resolver dispone de las operaciones de consulta y modificación, definidas en el Content Provider, pero abstrayendo su funcionamiento.

Únicamente es necesario conocer el URI público del proveedor de contenidos.

Consultando contenidos

A través del método `query` del Content Resolver. Parámetros:

La URI del proveedor a consultar.

Un listado de columnas a incluir en el resultado.

Una cláusula **`where`**. Puede incluirse el comodín **`?`**

Un listado de condiciones que reemplazan los comodines de la condición **`where`**.

El orden del resultado devuelto.

```
// Get the Content Resolver.
ContentResolver cr = getContentResolver();

// Specify the result column projection. Return the minimum set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
    MyHoardContentProvider.KEY_ID,
    MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
    MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN };

// Append a row ID to the URI to address a specific row.
Uri rowAddress =
ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI, rowId);

// Replace these with valid SQL statements as necessary.
String where = null;
String whereArgs[] = null;
String order = null;

// Return the specified rows.
Cursor resultCursor = cr.query(rowAddress, result_columns,
                               where, whereArgs, order);
```

Transacciones: insertar

Content Resolver ofrece dos métodos para insertar datos `insert` y `bulkInsert`

Los dos métodos hacen uso del URI del Content Provider e insertan objetos de la clase `ContentValues`

El método `insert` devuelve la URI del elemento insertado, mientras que `bulkInsert` devuelve el número de registros insertados

Transacciones: insertar

```
private Uri addNewHoard(String hoardName, float hoardValue,
                        boolean hoardAccessible) {
    // Create a new row of values to insert.
    ContentValues newValues = new ContentValues();

    // Assign values for each row.
    newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_NAME_COLUMN, hoardName);
    newValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN, hoardValue);
    newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
                  hoardAccessible);
    // [ ... Repeat for each column / value pair ... ]

    // Get the Content Resolver
    ContentResolver cr = getContentResolver();

    // Insert the row into your table
    Uri myRowUri = cr.insert(MyHoardContentProvider.CONTENT_URI, newValues);

    //
    return myRowUri;
}
```

Transacciones: eliminar

Content Resolver ofrece el método `delete`

Podemos pasarle la URI del elemento a eliminar, o una cláusula `where` si queremos eliminar varios elementos

Transacciones: eliminar

```
private int deleteEmptyHoards() {
    // Specify a where clause that determines which row(s) to delete.
    // Specify where arguments as necessary.
    String where = MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN + "=" + 0;
    String whereArgs[] = null;

  // Get the Content Resolver.
  ContentResolver cr = getContentResolver();

  // Delete the matching rows
  int deletedRowCount =
    cr.delete(MyHoardContentProvider.CONTENT_URI, where, whereArgs);

  return deletedRowCount;
}
```

Transacciones: actualizar

Content Resolver ofrece el método `update`

Podemos pasarle la URI del Content Provider, un objeto `ContentValue` con los datos a modificar y una cláusula `where` que indica que filas actualizar

Todas las filas coincidentes se actualizarán con los mismo valores

Transacciones: actualizar

```
private int updateHoardValue(int hoardId, float newHoardValue) {
    // Create the updated row content, assigning values for each row.
    ContentValues updatedValues = new ContentValues();
    updatedValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN,
                      newHoardValue);
    // [ ... Repeat for each column to update ... ]

    // Create a URI addressing a specific row.
    Uri rowURI =
      ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI, hoardId);

    // Specify a specific row so no selection clause is required.
    String where = null;
    String whereArgs[] = null;

    // Get the Content Resolver.
    ContentResolver cr = getContentResolver();

    // Update the specified row.
    int updatedRowCount =
      cr.update(rowURI, updatedValues, where, whereArgs);

    return updatedRowCount;
}
```

Ejercicio

Búsquedas

Añadiendo búsquedas

Gracias a la implementación de Content Provider, podemos añadir un mecanismo de búsqueda a nuestra aplicación

Android incluye un framework que nos permite añadir un control de búsqueda en nuestras actividades, o desde el widget de la pantalla de inicio

Añadiendo búsquedas

Disponemos de tres tipos de búsqueda:

Search Bar: cuadro de búsqueda sobre el título de la actividad. Se activa al pulsar el botón específico, o a través de software.

Search View: widget de búsqueda que puede ser insertado en cualquier parte de la actividad. Por norma general, se coloca en la Action Bar.

Quick Search Box: widget situado en la pantalla de inicio, que realiza búsqueda por todo el dispositivo.

Añadiendo búsquedas

Para permitir realizar búsquedas en nuestra aplicación, debemos definirlo previamente

`res/xml/searchable.xml`

```
<?xml version=”1.0” encoding=”utf-8”?>
<searchable xmlns:android=”http://schemas.android.com/apk/res/android”
    android:label=”@string/app_name”
    android:hint=”@string/search_hint”>
</searchable>
```

Crear una actividad de búsqueda

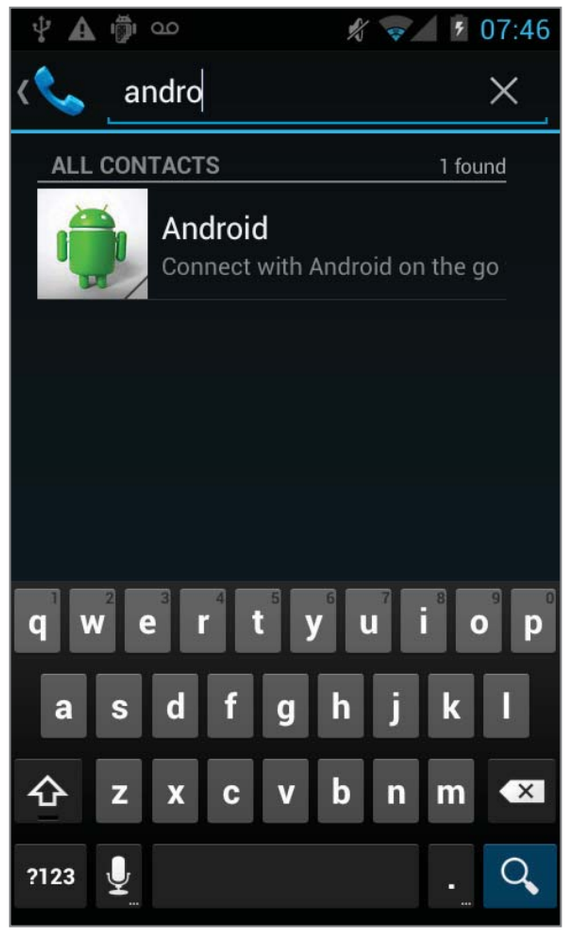
Debemos crear una actividad para mostrar los resultados de la búsqueda, generalmente basada en una `ListView`

```
<activity android:name=".DatabaseSkeletonSearchActivity"
          android:label="Element Search"
          android:launchMode="singleTop">
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
    <meta-data
        android:name="android.app.searchable"
        android:resource="@xml/searchable"
    />
</activity>
```

Search Bar

Para poder añadir el cuadro de diálogo en una actividad en concreto, debemos indicarlo en su nodo del manifiesto

```
<meta-data android:name=”android.app.default_searchable”
           android:value=”.DatabaseSkeletonSearchActivity”
/>
```



Obtener la búsqueda

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Get the launch Intent
    parseIntent(getIntent());
}

@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    parseIntent(getIntent());
}

private void parseIntent(Intent intent) {
    // If the Activity was started to service a Search request,
    // extract the search query.
    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String searchQuery = intent.getStringExtra(SearchManager.QUERY);

        // Perform the search
        performSearch(searchQuery);
    }
}
```

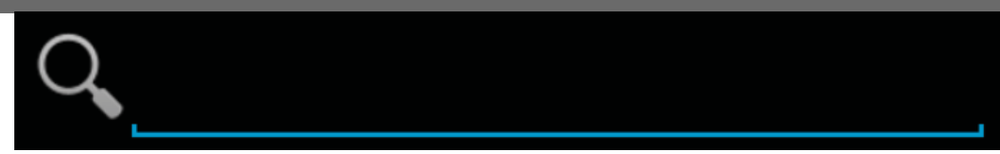
Mostrar los resultados

Una vez que la actividad recibe el texto de búsqueda, hay que ejecutar la búsqueda y mostrar los resultados

El cómo se realice esta búsqueda (el qué y dónde) dependerá de la aplicación y lo que se desee mostrar

Lo normal es acceder al Content Provider que hemos creado previamente, aunque también podemos acceder directamente a la BD

Search View Widget



Se introdujo como una alternativa a Search Box, aunque se comporta de manera muy parecida.

Podemos situarlo en cualquier parte de la interfaz, aunque es recomendable situarlo en la Action Bar.

Debemos conectar esta actividad al campo de búsqueda

```
// Use the Search Manager to find the SearchableInfo related
// to this Activity.
SearchManager searchManager =
    (SearchManager)getSystemService(Context.SEARCH_SERVICE);
SearchableInfo searchableInfo =
    searchManager.getSearchableInfo(getComponentName());

// Bind the Activity's SearchableInfo to the Search View
SearchView searchView = (SearchView)findViewById(R.id.searchView);
searchView.setSearchableInfo(searchableInfo);
```

Sugerencias de búsqueda

Una de las mejoras que incluye Search View es la posibilidad de mostrar sugerencias de resultados en tiempo real

Se muestra una lista de resultados, con la posibilidad de seleccionar cualquiera de ellos y saltarse la pantalla de resultados

Si queremos ofrecer esta posibilidad, debemos modificar el Content Provider asociado para que muestre las sugerencias

Sugerencias de búsqueda

```
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;
private static final int SEARCH = 3;

private static final UriMatcher uriMatcher;

static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(“com.paad.skeletondatabaseprovider”,
                      “elements”, ALLROWS);
    uriMatcher.addURI(“com.paad.skeletondatabaseprovider”,
                      “elements/#”, SINGLE_ROW);

    uriMatcher.addURI(“com.paad.skeletondatabaseprovider”,
                      SearchManager.SUGGEST_URI_PATH_QUERY, SEARCH);
    uriMatcher.addURI(“com.paad.skeletondatabaseprovider”,
                      SearchManager.SUGGEST_URI_PATH_QUERY + “/*”, SEARCH);
    uriMatcher.addURI(“com.paad.skeletondatabaseprovider”,
                      SearchManager.SUGGEST_URI_PATH_SHORTCUT, SEARCH);
    uriMatcher.addURI(“com.paad.skeletondatabaseprovider”,
                      SearchManager.SUGGEST_URI_PATH_SHORTCUT + “/*”, SEARCH);
}
```

Sugerencias de búsqueda

```
@Override
public String getType(Uri uri) {
    // Return a string that identifies the MIME type
    // for a Content Provider URI
    switch (uriMatcher.match(uri)) {
      case ALLROWS:
        return "vnd.android.cursor.dir/vnd.paad.elemental";
      case SINGLE_ROW:
        return "vnd.android.cursor.item/vnd.paad.elemental";
      case SEARCH :
        return SearchManager.SUGGEST_MIME_TYPE;
      default:
        throw new IllegalArgumentException("Unsupported URI: " + uri);
    }
}
```

Sugerencias de búsqueda

Search Manager envía al Content Provider la búsqueda realizada. Debemos devolver un cursor con columnas predefinidas.

`SUGGEST_COLUMN_TEXT_1:` muestra el texto de la busqueda.

`_id:` un identificador único del resultado.

`SUGGEST_COLUMN_INTENT_DATA_ID:` valor que puede ser añadido a un URI específico para mostrar el resultado directamente.

Sugerencias de búsqueda

```
public static final String KEY_SEARCH_COLUMN = KEY_COLUMN_1_NAME;

private static final HashMap<String, String> SEARCH_SUGGEST_PROJECTION_MAP;
static {
    SEARCH_SUGGEST_PROJECTION_MAP = new HashMap<String, String>();
    SEARCH_SUGGEST_PROJECTION_MAP.put("_id", KEY_ID + " AS " + "_id");
    SEARCH_SUGGEST_PROJECTION_MAP.put(
      SearchManager.SUGGEST_COLUMN_TEXT_1,
      KEY_SEARCH_COLUMN + " AS " + SearchManager.SUGGEST_COLUMN_TEXT_1);
    SEARCH_SUGGEST_PROJECTION_MAP.put(
      SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID, KEY_ID +
      " AS " + "_id");
}
```

Sugerencias de búsqueda

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

    // If this is a row query, limit the result set to the passed in row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        queryBuilder.appendWhere(KEY_ID + "=" + rowID);
        break;
      case SEARCH :
        String query = uri.getPathSegments().get(1);
        queryBuilder.appendWhere(KEY_SEARCH_COLUMN +
          " LIKE \"%" + query + "%\"");
        queryBuilder.setProjectionMap(SEARCH_SUGGEST_PROJECTION_MAP);
        break;
      default: break;
    }

    Cursor cursor = queryBuilder.query(db, projection, selection,
        selectionArgs, groupBy, having, sortOrder);

    return cursor;
}
```

Sugerencias de búsqueda

El último paso es especificar que Content Provider va a ser el encargado de especificar el sugerencias para las búsquedas

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
  android:label="@string/app_name"
  android:searchSuggestAuthority=
    "com.paad.skeletonsearchabledatabaseprovider"
  android:searchSuggestIntentAction="android.intent.action.VIEW"
  android:searchSuggestIntentData=
    "content://com.paad.skeletonsearchabledatabaseprovider/elements">
</searchable>
```

Content Provider

Nativos de Android

Content Provider nativos

Android dispone de varios Content Provider:

Media Store: centraliza y gestiona el acceso al contenido multimedia del dispositivo (audio, vídeo e imágenes).

Browser: contenidos del navegador e historial de búsqueda.

Contacts Contract: detalles de los contactos.

Calendar: calendarios, eventos, listas y recordatorios.

Call log: acceso completo al historial de llamadas.