MyTI

Ricardo Queirós

Android™

Bases de Dados e Geolocalização

Acesso a dados em SQLite, MySQL e Realm

Mapeamento objeto-relacional com o Sugar ORM

API: Maps, Location, Geofencing, Activity Recognition e Places

Programação em Java, compatível com o Android 7 Nougat, através do Android Studio 2

FCA

EDIÇÃO
FCA – Editora de Informática, Lda.
Av. Praia da Vitória, 14 A – 1000-247 Lisboa
Tel: +351 213 511 448
fca@fca.pt
www.fca.pt

DISTRIBUIÇÃO
Lidel – Edições Técnicas, Lda.
Rua D. Estefânia, 183, R/C Dto. – 1049-057 Lisboa
Tel: +351 213 511 448
lidel@lidel.pt
www.lidel.pt

LIVRARIA
Av. Praia da Vitória, 14 A – 1000-247 Lisboa
Tel: +351 213 511 448 * Fax: +351 213 522 684
livraria@lidel.pt

ISBN edição impressa: 978-972-722-862-1
1.ª edição impressa: novembro 2016

Impressão e acabamento: Cafilesa – Soluções Gráficas, Lda. – Venda do Pinheiro
Depósito Legal n.º 417590/16
Capa: José M. Ferrão – *Look-Ahead*

ÍNDICE GERAL

O AUTOR

Ricardo Queirós (ricardo.queiros@gmail.com) – Doutorado em Ciências de Computadores pela Faculdade de Ciências da Universidade do Porto (FCUP). Exerce a sua atividade como docente na Escola Superior de Media Artes e Design (ESMAD), onde é responsável por disciplinas na área da Programação de Computadores, focada para os ambientes Web e Mobile. Paralelamente, desenvolve atividade científica na área de Interoperabilidade entre Sistemas de E-learning. Membro efetivo do Center for Research in Advanced Computing Systems (CRACS), uma unidade de investigação do laboratório Associado INESC TEC.

Autor dos livros *Android – Desenvolvimento de Aplicações com Android Studio*, *Desenvolvimento de Aplicações Profissionais em Android* e *Android – Introdução ao Desenvolvimento de Aplicações* e coautor do livro *Introdução ao Desenvolvimento de Jogos em Android*, todos publicados pela FCA.

4 BASES DE DADOS

As plataformas móveis oferecem uma variedade de opções de armazenamento: ficheiros, preferências partilhadas, bases de dados relacionais, entre outras. A escolha da opção de armazenamento ideal não é simples. Mesmo que decida usar uma base de dados relacional, terá ainda que decidir qual API usar. Neste capítulo, apresentam-se algumas bibliotecas de bases de dados relacionais disponíveis para Android.

4.1 INTRODUÇÃO

Até agora, foram apresentadas técnicas de persistência de dados simples. Quando se pretende armazenar dados relacionados, é mais eficiente usar-se uma base de dados. O sistema Android suporta a popular base de dados SQLite. Contudo, apesar das inúmeras vantagens que o SQLite possui, como a não necessidade de dependências, o benefício do uso de um *standard* como o SQL ou o controlo absoluto sobre as *queries*, existem também desvantagens associadas ao uso desta base de dados, tais como:

- Quantidade enorme de código repetitivo e, como tal, ineficiências (também a longo prazo com a manutenção da aplicação);
- Limitação de 1 MB para campos *Binary Large OBject* (BLOB) no Android;
- Necessidade de dominar a linguagem SQL (as *queries* podem ficar longas e complexas);
- Sem verificações em tempo de compilação (por exemplo, em *queries* SQL).

Nesse sentido, poderá estudar várias alternativas ao SQLite. Por exemplo, pode usar um *layer* de abstração em cima do SQLite. Essa abstração é geralmente assegurada através de um *Object-Relational Mapping* (ORM). Alguns bons exemplos são: o GreenDAO, o DBFlow, o Sugar ORM e o ORMLite. No entanto, se desejar substituir completamente o SQLite, existem algumas bases de dados alternativas, como a Couchbase Lite, a Interbase, a LevelDB, a Oracle Berkeley DB, a Realm e a SnappyDB, entre outras.

As próximas três secções destacam a base de dados SQLite, o Sugar ORM e a base de dados Realm, respetivamente.

4.2 SQLITE

O sistema Android suporta a base de dados SQLite. Trata-se de uma pequena biblioteca *open-source* que implementa um amplo subconjunto do standard SQL 92. O SQLite está disponível e é suportado de forma nativa por todos os dispositivos Android. O uso de uma base de dados numa aplicação Android restringe o acesso à aplicação que a criou, ou seja, a base de dados poderá ser acedida apenas pelas classes da aplicação. Para manipular uma base de dados SQLite, usam-se as classes `SQLiteOpenHelper` e `SQLiteDatabase`.

4.2.1 CLASSES `SQLiteOpenHelper` E `SQLiteDatabase`

Para criar e atualizar uma base de dados **SQLite**, recomenda-se a extensão da classe `SQLiteOpenHelper`. No construtor da subclasse invoca-se o método `super` de `SQLiteOpenHelper`, especificando o nome da base de dados e a sua versão atual. Nesta classe, é necessário implementar dois métodos:

- `onCreate` – é invocado se a base de dados não existir. Aqui, pode-se executar, por exemplo, um comando SQL para criar tabelas na base de dados;
- `onUpgrade` – é chamado se a versão da base de dados for aumentada no código da aplicação. Este método permite atualizar o esquema da base de dados.

Ambos os métodos recebem um objeto `SQLiteDatabase` como parâmetro, representando a base de dados.

Pode-se obter uma instância da implementação `SQLiteOpenHelper` usando o construtor definido. Para escrever e ler a partir da base de dados, usam-se os métodos `getWritableDatabase` e `getReadableDatabase`. Ambos os métodos devolvem um objeto `SQLiteDatabase` que, respetivamente, representa a base de dados e fornece métodos de escrita e de leitura.

Para fazer *debugging* de aplicações com base de dados, o Android SDK inclui a ferramenta **sqlite3** que, além de outras funções, permite navegar pelas tabelas e executar comandos SQL.

`SQLiteDatabase` é a classe base para trabalhar com uma base de dados **SQLite**. Esta classe fornece métodos para abrir, consultar, atualizar, remover e fechar uma base de dados. Os principais métodos são apresentados na Tabela 4.1.

MÉTODO	DESCRIÇÃO	RETORNO
`Insert( String table, ContentValues values)`	Insere um registo (linha) numa tabela da base de dados.	Identificador da linha inserida ou -1 em caso de erro.

MÉTODO	DESCRIÇÃO	RETORNO
`Update(String table, ContentValues values, String whereClause, String[] whereArgs)`	Atualiza registos na base de dados.	Número de registos afetados.
`Delete(String table, String whereClause, String[] whereArgs)`	Remove registos da base de dados.	Número de registos afetados.
`execSQL(String sql)`	Executa uma única instrução SQL que não seja um `SELECT` ou qualquer outra instrução SQL que retorne dados. Bons candidatos são a criação e a remoção de tabelas.	Nada retorna. No caso de a instrução SQL ser inválida, lança uma `SQLException`.
`getPath()`	Obtém o caminho para o ficheiro da base de dados.	Caminho para o ficheiro da base de dados.
`isOpen()`	Verifica se a base de dados está aberta.	Verdadeiro, caso a base de dados esteja aberta, ou falso, em caso contrário.
`rawQuery(String sql, String[] selectionArgs)`	Executa uma *query* de acordo com uma instrução SQL passada como parâmetro.	Um objeto `Cursor` posicionado antes do primeiro registo.

TABELA 4.1 – Métodos da classe `SQLiteDatabase`

O objeto `ContentValues` permite definir pares chave-valor. A chave representa o identificador de coluna da tabela e o valor representa o conteúdo para o registo da tabela nessa coluna. O objeto pode ser usado para inserções e atualizações de registos na base de dados. As consultas podem ser criadas através dos métodos `rawQuery` e `query` ou da classe `SQLiteQueryBuilder`. O método `rawQuery` aceita diretamente uma instrução SQL `SELECT` como parâmetro.

O código seguinte fornece um exemplo de uso deste método:

```
Cursor cursor =
  getReadableDatabase().rawQuery("SELECT * FROM student where _id = ?",
  new String[] { id });
```

Este exemplo demonstra o uso do método na invocação de uma instrução SQL para obter um registo da tabela `student` de acordo com o seu identificador. O caráter `?` incluso na instrução SQL é substituído pelos valores do segundo parâmetro.

O método `query` permite também especificar uma consulta SQL. Este método aceita parâmetros de consulta diversos, tais como a identificação do nome da tabela a consultar, as colunas a selecionar, o nível de agrupamento, etc. O método `query` tem os seguintes parâmetros (Tabela 4.2):

PARÂMETRO	DESCRIÇÃO
`String table`	Nome da tabela.
`String[] columns`	Lista de colunas a devolver. Se for nulo, devolve todas.
`String selection`	Filtro definindo os registos a devolver. Equivalente à cláusula `WHERE` (mas sem o `WHERE`). Se for nulo, devolve todos os registos da tabela.
`String[] selectionArgs`	Lista de valores a substituir os carateres `?` encontrados no parâmetro `selection`.
`String[] groupBy`	Filtro indicando como agrupar registos (equivalente à cláusula `GROUP BY` do SQL).
`String having`	Filtro identificando que agrupamentos devolver (equivalente à cláusula `HAVING` do SQL).
`String orderBy`	Tipo de ordenação dos registos (equivalente à cláusula `ORDER BY` do SQL).

TABELA 4.2 – Parâmetros do método `query`

O código seguinte fornece um exemplo de uso deste método:

```
return database.query("student",
  new String[] { "_id", "name" }, null, null, null, null, null);
```

Neste exemplo, são obtidas as colunas `_id` e `name` de todos os registos da tabela `student`.

Ambos os métodos `rawQuery` e `query` retornam um objeto `Cursor`. Os cursores são "apontadores de dados" da base de dados, ou seja, uma interface que permite o acesso aos dados devolvidos por uma *query*. Um objeto `Cursor` aponta basicamente para uma linha do resultado da consulta. Desta forma, podem-se obter os resultados da consulta eficientemente, já que não tem que se carregar todos os dados na memória. Para obter o número de registos devolvidos use o método `getCount`.Para se mover entre registos use os métodos `moveToFirst` e `MoveToNext`. O método `isAfterLast` permite verificar se o fim dos resultados da consulta foi atingido.

O objeto `Cursor` fornece também métodos com o prefixo `get[Type]`, por exemplo, `getLong(columnIndex)` ou `getString(columnIndex)` para aceder aos dados de uma coluna específica (dada pelo `columnIndex`) na posição atual. Outro método importante é

`getColumnIndexOrThrow(String)`, que permite obter o índice de coluna para um dado nome da coluna da tabela. Após o seu uso, um objeto `Cursor` precisa de ser fechado através do seu método `close`.

Para consultas complexas, como as que requerem *aliases* de coluna, deve-se usar a classe `SQLiteQueryBuilder`, que fornece vários métodos para a construção de consultas SQL.

4.2.2 EXEMPLO PRÁTICO

O próximo exemplo explica a criação de uma aplicação de gestão de *bookmarks*. A aplicação vai permitir a criação, a consulta e a remoção de *bookmarks* organizados por categorias. Comece por criar um projeto Android chamado **MyBookmarks**.

A aplicação será baseada numa base de dados **SQLite** chamada **bookmarks**. A base de dados inclui duas tabelas: a tabela **Bookmark**, responsável por armazenar *bookmarks* caracterizados por um título e um URL, e a tabela **Categoria**, que armazena as categorias a que os *bookmarks* podem pertencer. A Figura 4.1 ilustra a relação das duas tabelas da base de dados.

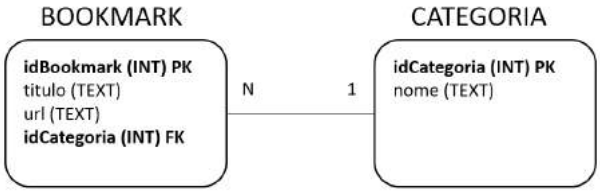

FIGURA 4.1 – Diagrama da base de dados

A relação entre as tabelas é de 1 para N – isto é, um *bookmark* só pode pertencer a uma categoria e uma categoria pode ter vários *bookmarks*.

O exemplo não cobre o desenho gráfico da aplicação, mas explica o desenho da base de dados, bem como a criação das classes de representação e manipulação de dados. Os testes serão feitos recorrendo à ferramenta **LogCat**.

A aplicação será composta por uma classe da atividade e por uma classe manipuladora da base de dados (subclasse de `SQLiteOpenHelper`). A primeira vai comunicar com a classe manipuladora para interagir com a base de dados (adicionar, remover e consultar entradas na base de dados). A segunda fornecerá uma camada abstrata entre a base de dados **SQLite** subjacente e a classe da atividade.

A fim de implementar essa interação de uma forma estruturada, será necessário definir novas classes responsáveis por manter os dados da base de dados em memória. Estas novas classes denominam-se **modelos de dados**.

8.2.3 PEDIDOS DE ATUALIZAÇÃO DE LOCALIZAÇÃO

Em seguida, é criado um objeto `LocationRequest`. Este objeto vai definir o intervalo entre as atualizações e a precisão de localização que se deseja. Os principais métodos são os seguintes:

- `setInterval` – define a taxa, em milissegundos (ms), em que a aplicação prefere receber atualizações de localização;
- `setFastestInterval` – define o intervalo mais rápido, em ms, para atualizações de localização. Isto controla a taxa mais rápida em que a aplicação receberá atualizações de localização, que pode ser mais rápida do que `setInterval` em algumas situações (por exemplo, se outras aplicações estão a receber atualizações de localização);
- `setPriority` – define o parâmetro de precisão. Numa aplicação em primeiro plano, são necessárias atualizações de localização constantes com elevada precisão, pelo que se aconselha o uso do valor `PRIORITY_HIGH_ACCURACY`.

Depois, o objeto é enviado como parte do pedido para iniciar as atualizações através do método `requestLocationUpdates`:

```
@Override
public void onConnected(Bundle bundle) {
  mLastLocation = LocationServices.FusedLocationApi.getLastLocation(
  mGoogleApiClient);
  if(mLastLocation!=null) {
    latLng = new LatLng(
      mLastLocation.getLatitude(),
       mLastLocation.getLongitude());
    updateUI();
  }
  mLocationRequest = new LocationRequest();
  mLocationRequest.setInterval(5000);
  mLocationRequest.setFastestInterval(3000);
  mLocationRequest.setPriority(
    LocationRequest.PRIORITY_BALANCED_POWER_ACCURACY);
  LocationServices.FusedLocationApi.requestLocationUpdates(
    mGoogleApiClient, mLocationRequest, this);
}
```

O serviço de localização envia atualizações de localização para a aplicação através de uma *intent* ou, mais usualmente, como um argumento passado para um método chamado `onLocationChanged(Location)` da interface `LocationListener`. O argumento de entrada é um objeto `Location` com a latitude e a longitude do local. Esta informação raramente é útil aos utilizadores da aplicação, que costumam estar mais familiarizados com nomes de ruas, cidades e lugares. Para encontrar um nome útil para uma determinada latitude e longitude o Android fornece a classe `Geocoder`.

O próximo código mostra a implementação do método `onLocationChanged`. Quando há uma atualização da localização do dispositivo, o mapa é atualizado através do movimento animado da câmara, e o nome da rua atual é exibido depois da conversão do objeto `Location` passada por parâmetro no método `getStreetFromLocation`:

```
@Override
public void onLocationChanged(Location location) {
  street = getStreetFromLocation(this, location);
  latLng = new LatLng(location.getLatitude(), location.getLongitude());
  CameraPosition cameraPosition = new CameraPosition.Builder()
    .target(latLng).zoom(17).build();
  myMap.animateCamera(
    CameraUpdateFactory.newCameraPosition(cameraPosition));
  updateUI();
}
```

O método `getStreetFromLocation` instancia um objeto `Geocoder` e obtém uma lista de endereços baseada na localização recebida como parâmetro. Depois, usam-se os métodos `getAddressLine` e `getThoroughFare` para obter o nome da rua ou da avenida:

```
public static String getStreetFromLocation(Context ctx, Location loc) {
  try {
    Geocoder geocoder = new Geocoder(ctx);
    List<Address> addresses = geocoder.getFromLocation(
      loc.getLatitude(), loc.getLongitude(), 1);
    if(addresses != null && addresses.size() > 0) {
      String streetName = addresses.get(0).getAddressLine(0);
      if(streetName == null) {
        streetName = addresses.get(0).getThoroughfare();
      }
      return streetName;
    } else { return null; }
```

```
  } catch (IOException e) { return null; }
}
```

A atualização da interface do utilizador limita-se a definir o novo conteúdo nos objetos `TextView`:

```
void updateUI() {
  txtOutputLat.setText(String.valueOf(latLng.latitude));
  txtOutputLon.setText(String.valueOf(latLng.longitude));
  txtStreet.setText(street);
}
```

8.2.4 EXECUÇÃO E TESTE

Para simular a atualização da localização e testar a aplicação, inicie o emulador do Android Studio ou outro – como, por exemplo, o Genymotion – e ative o GPS (Figura 8.2).

FIGURA 8.2 – Ativação do GPS no emulador Genymotion

Em seguida, vá alterando a localização na caixa de diálogo do GPS e verificando as atualizações do mapa e os respetivos dados da localização atual, nomeadamente as coordenadas de latitude e longitude e o nome da rua obtido através de geocodificação (Figura 8.3).

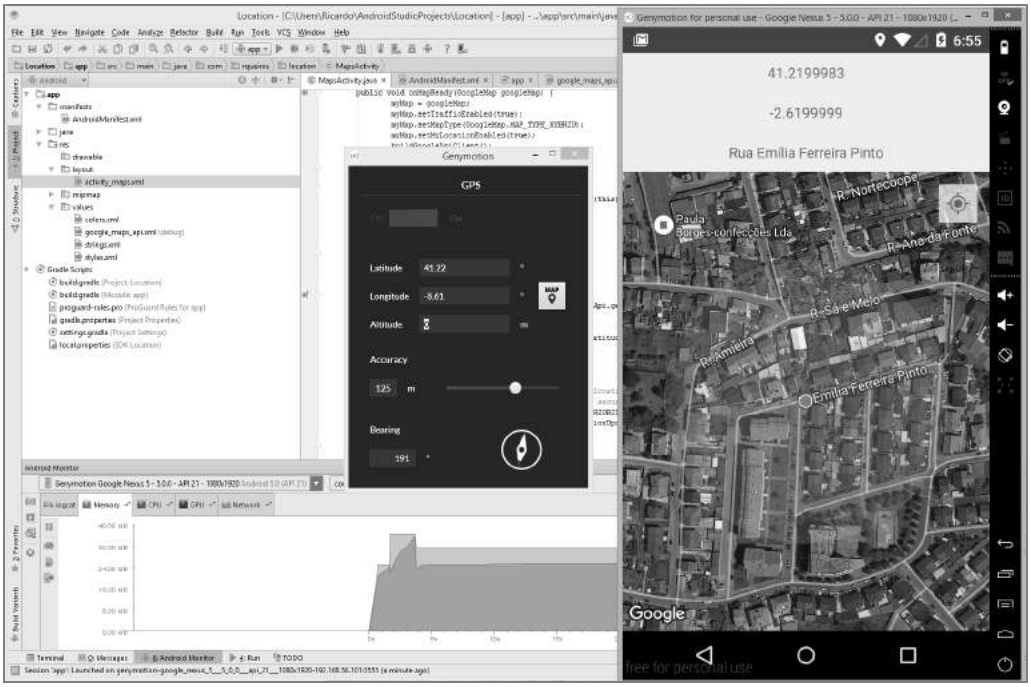

FIGURA 8.3 – Teste de atualização da localização no emulador Genymotion

8.3 API `Geofencing`

Um *geofence* representa uma região geográfica circular definida através de dois parâmetros: a localização (latitude e longitude) e um raio (Figura 8.4).

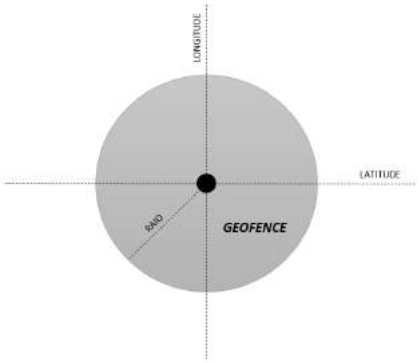

FIGURA 8.4 – *Geofence*

No sistema Android, para trabalhar com *geofences* usa-se a API `Geofencing`. Esta faz parte do conjunto de API `Location` e inclui as seguintes classes e interfaces: `Geofence`, `GeofenceStatusCodes`, `GeofencingEvent` e `GeofencingRequest`.

A interface `Geofence` representa uma área geográfica a ser monitorizada. Esta é criada através do método `build` da classe `Geofence.Builder`.

A Tabela 8.2 enumera os principais métodos desta classe.

MÉTODOS	DESCRIÇÃO
`build()`	Cria um objeto `Geofence`.
`setCircularRegion(double lat, double lgn, float r)`	Define a região do *geofence* em metros (m). A área é definida pela latitude e longitude do local de interesse e por um raio, medido em metros, que permite ajustar quão perto o utilizador precisa de se aproximar do local antes de o *geofence* ser detetado. Quanto maior for o raio, maior será também a probabilidade de o utilizador acionar um alerta de transição *geofence* por se aproximar da área demarcada.
`setExpirationDuration(long durMs)`	Define o tempo de vida do *geofence*, em ms. Assim que o tempo de expiração é atingido, o serviço exclui a área demarcada.
`setLoiteringDelay(int ldMs)`	Define o atraso entre os estados de entrada e de permanência no *geofence*, em milissegundos (ms). Aqui, o atraso significa que a transição `GEOFENCE_TRANSITION_DWELL` será notificada `ldMs` ms depois de o dispositivo entrar no *geofence*.
`setNotificationResponsiveness(int nrMs)`	Define a capacidade de resposta de notificação de melhor esforço da área demarcada.
`setRequestId(String requestId)`	Define um identificador único para o *geofence*. Este identificador pode ser usado posteriormente para, por exemplo, remover um *geofence* do serviço de localização.
`setTransitionTypes(int types)`	Define os tipos de transição que se deseja monitorizar. Com o *geofencing*, a aplicação pode ser notificada quando o dispositivo entra, permanece ou sai de uma área predefinida. Os valores possíveis são os seguintes: – `Geofence.GEOFENCE_TRANSITION_ENTER`; – `Geofence.GEOFENCE_TRANSITION_DWEEL`; – `Geofence.GEOFENCE_TRANSITION_EXIT`.

TABELA 8.2 – Métodos da classe `Geofence.Builder`

O próximo exemplo mostra como criar um objeto `Geofence` definindo um identificador, uma área geográfica, uma data de expiração e os tipos de transições a serem monitorizados:

```
Geofence geofence = new Geofence.Builder()
    .setRequestId(GEOFENCE_REQ_ID)
    .setCircularRegion(LATITUDE, LONGITUDE, RADIUS)
    .setExpirationDuration(DURANTION)
    .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER |
                        Geofence.GEOFENCE_TRANSITION_EXIT)
    .build();
```

Para minimizar o consumo de energia é recomendado o uso de uma área demarcada, com um raio de, pelo menos, 100 metros para a maioria das situações. Se os *geofences* estiverem localizados numa zona rural, deverá aumentar o raio até 500 metros ou mais para garantir que as áreas demarcadas são eficazes.

São três os tipos de transição (Figura 8.5) que se podem monitorizar num *geofence*:

- `GEOFENCE_TRANSITION_ENTER` – o utilizador entrou na região demarcada;
- `GEOFENCE_TRANSITION_DWEEL` – o utilizador permaneceu por um período definido na região demarcada. Essa abordagem pode ajudar a reduzir o *spam alert* resultante do grande número de notificações quando um dispositivo entra e sai, sistematicamente, do *geofence*. Pode-se definir o tempo de permanência através do método `setLoiteringDelay`;
- `GEOFENCE_TRANSITION_EXIT` – o utilizador saiu da região demarcada.

FIGURA 8.5 – Tipos de transições num *geofence*

Por sua vez, a classe `GeofencingRequest` recebe os *geofences* que devem ser monitorizados. É possível criar uma instância através de um `Builder`, passando um objeto ou uma lista de objetos `Geofence`, e o tipo de notificação a ser despoletado quando o *geofence* é criado:

```
GeofencingRequest request = new GeofencingRequest.Builder()
    .setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
    .addGeofence(geofence)
    .build();
```

A classe `GeofencingApi` é o ponto de entrada para todas as interações com a API `Geofencing` dependendo da `GoogleApiClient` para funcionar. Esta classe será usada para adicionar e remover *geofences*.

Para adicionar um *geofence* usa-se o método `addGeofence`. A partir deste momento, a área respetiva do *geofence* é monitorizada usando as definições passadas ao `GeofencingRequest` e um `PendingIntent` invocado quando uma transição (por exemplo, de entrada ou de saída) ocorre:

```
PendingResult<Status> addGeofences (GoogleApiClient client,
    GeofencingRequest geofencingRequest,
    PendingIntent pendingIntent)
```

Para remover um *geofence* usa-se o método `removeGeofences`. É possível remover um *geofence* através do identificador de pedido ou usando o seu `PendingIntent`:

```
PendingResult<Status> removeGeofences(GoogleApiClient client,
    List<String> geofenceRequestIds)
PendingResult<Status> removeGeofences (GoogleApiClient client,
   PendingIntent pendingIntent)
```

8.3.1 EXEMPLO PRÁTICO

Nesta secção, detalha-se a criação de um projeto Android denominado **MyGeofenceApp**. A aplicação (Figura 8.6) vai permitir a criação de *geofences* através das suas coordenadas (latitude e longitude) ou de um simples clique num mapa. A representação gráfica do *geofence* será exibida num mapa. Serão também enviadas notificações assim que o dispositivo entre ou saia da região demarcada.

Esta aplicação é uma adaptação de uma aplicação básica, apresentada por Tin Megali, que pode ser consultada no seguinte *link*: https://github.com/tutsplus/Android-GeofencingBasics.

FIGURA 8.6 – Aplicação MyGeofencingApp

8.3.2 CRIAÇÃO E CONFIGURAÇÃO DO PROJETO

Crie um projeto Android chamado **MyGeofencingApp**. Na seleção da atividade a adicionar ao projeto, escolha a opção **Google Maps Activity**, pois esta simplifica a configuração do projeto. Em seguida, abra o ficheiro **google_maps_api.xml** e siga as instruções para obter a chave para a API `Maps`. Após obter a chave, insira-a no recurso *string* do mesmo ficheiro:

```
<string name="google_maps_key" ...>CHAVE</string>
```

8.3.3 DEFINIÇÃO DO *LAYOUT*

O *layout* da aplicação é composto por um `LinearLayout` que contém duas `TextView`, duas `EditText`, dois `Button` agregados num outro `LinearLayout` e, por fim, um `MapFragment` onde será exibido o mapa. Este último está incluído por omissão após a criação do projeto, pelo que não precisa de inseri-lo. Sendo assim, edite o ficheiro de *layout* **activity_maps.xml** e inclua o seguinte código:

```
<LinearLayout ...
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Latitude"
    android:id="@+id/textview1" />
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/lat" />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Longitude"
    android:id="@+id/textview2" />
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/lng"
    android:layout_gravity="center_horizontal" />
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
 <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="CRIAR GEOFENCE"
    android:id="@+id/button"
    android:onClick="createGeofence"
    />
 <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="REMOVER GEOFENCE"
```