Geolocation

La **geolocalizzazione** è una delle caratteristiche principali che rendono uniche le applicazioni per dispositivi mobili; spesso il termine citato è utilizzato in relazione all'espressione *context-awareness*.

Un qualsiasi progetto è definito *context aware* qualora la possibilità di eseguire la propria computazione dipenda strettamente dal contesto. Pertanto, lo sviluppo di applicazioni mobili, deve conseguire una volontà prestabilita, ossia rendere la computazione indipendente da qualsiasi realtà, descrivendone in questo modo il più ampio spettro d'azione. Il successo di un'applicazione dipende strettamente dalle dipendenze in cui opera.

GPS è l'acronimo di *Global Positioning System*, un sistema complesso basato su flotte di satelliti, i quali trasmettono periodicamente comunicazioni relative alle proprie posizioni ed orari atomici. I segnali sono acquisiti ed elaborati dai device, i quali sono in grado di calcolare la propria posizione posta sulla proiezione del pianeta Terra. Maggiore è il numero di segnali ricevuti migliore risulterà l'accuratezza della posizione.

In *Android* il *location manager* è la figura responsabile del calcolo relativo alla posizione, tale per cui, se dovesse variare, lo stesso meccanismo inoltrerà notifiche all'intero sistema ideato. Trattandosi di un dato estremamente sensibile, occorre definire i permessi a livello *Manifest*. Sono contraddistinte tre tipologie di *localizzazione*, suddivise in:

- **Access_Fine_Location**, abilita l'accesso alla posizione con la migliore accuratezza possibile
- **Access_Coarse_Location**, abilita l'accesso alla posizione del dispositivo calcolata mediante *Wi-fi* oppure dalla *localizzazione cellulare*
- **Access_Background_Location**, abilita l'accesso alla posizione anche in circostanze in cui l'applicazione non sia adoperata

Proseguendo, sono attuati due metodi principali per gestire la *geolocalizzazione*; entrambi sono valide alternative, tuttavia la prima presentata è caratterizzata da un elevato consumo di batteria.

Location Listener è il primo dei due citati. Il proprio funzionamento prevede una richiesta *update* periodica al *location manager*; quest'ultimo, adeguando le operazioni necessarie, comunicherà il risultato alla richiesta avanzata. Un tipico problema riguarda circostanze in cui differenti entità esterne dovessero richiedere la *geolocalizzazione*, aumentando spropositamente il carico computazionale.
Innanzitutto è necessario creare un *listener* che possa attendere notifiche qualora vari la posizione del dispositivo.

```
val locationListener = LocationListener{location: Location ->
  // do your stuff
}
```

Tuttavia, come è stato già espresso prima, il *location manager* potrebbe essere sovraccaricato di richieste inerenti alla *localizzazione*, con conseguente crash dell'applicazione. Proprio per questa ragione è stato introdotto un approccio *opportunistico*, dettato dalla possibilità di

fondere le molteplici *request*.

Un paradigma simile richiede lo sviluppo di un **FusedLocationProvider**, il quale gestisce le richieste e ottimizza l'accesso al *GPS*. Il primo passo richiede l'istanziazione del servizio.

```
val fusedLocationProvideClient =
    LocationServices.getFusedLocationProviderClient(this)
```

Successivamente è necessario ottenere l'ultima *locazione conosciuta*.

```
fusedLocationClient.lastLocation.addOnSuccessListener { location : Location? ->
  // got last known location
}
```

Infine occorre definire una funzione di *callback* che possa comunicare cambiamenti di locazione.

```
val locationCallback = object:LocationCallback() {
  override fun onLocationResult(p0:LocationResult) {
    for(location in p0.locations) {
      // update with location data
    }
  }
}
```

Naturalmente, dato il dispendio computazionale, l'insieme di *request* relative alla *geolocation* sono eseguite su *flussi di esecuzioni indipendenti*, progettati in modo tale che possano risvegllare il *main thread*.

Geofencing

Il termine **Geofencing** rappresenta una determinata metodica di sviluppo, tale per cui l'applicazione è in grado di percepire qualora l'utente dovesse trovarsi in specifiche località. Una caratteristica del genere è capace di rendere un progetto sempre più indipendente, incline alla definizione *context-awareness*.

Adopera il modulo *Location Services*, posto in maniera tale che possa comunicare ad un *Listener* qualora accada un certo evento; secondo questo sviluppo, è necessario abilitare l'accesso alla posizione corrente del dispositivo anche quando l'applicazione installata non sia utilizzata.

Prima di tutto occorre ottenere la posizione corrente dell'utente, mediante:

```
val geofencingClient = LocationServices.getGeofencingClient(this)
```

Proseguendo, il passo successivo consiste nella definizione dei "recinti geografici", implementati tramite l'ausilio del design pattern *Builder*.

```
geofenceList.add(Geofence.Builder()
  .setRequestId(myId)
  .setCircularRegion(myLatitude, myLongiutude, myRadius)
  .seTexpirationDuration(myDuration)
  .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER or
      Geofence.GEOFENCE_TRANSITION_EXIT)
  .build())
```

Infine, è inserita la lista precedente all'interno di una richiesta, rendendola effettivamente attiva.

```
val geofencingRequest = GeofencingRequest.Builder().apply {
  setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
  addGeofences(geofenceList)
}.build()
```

```
geofencingClient?.addGeofences(geofencingRequest, pendingIntent)?.run{
  addOnSuccessListener { /* do your stuff */ }
  addOnFailureListener { /* do your stuff */ }
}
```

Nota a margine è inerente all'intento passato come input alla funzione *addGeofences()*; il fatto che sia un *pendingIntent* è dettato dalla possibilità che l'attività in questione possa operare anche in situazioni in cui l'applicazione è chiusa.

Maps

Le mappe sono uno strumento estremamente importante per rendere pervasiva un'applicazione. Differenti sono le alternative che possono soddisfare una richiesta simile, come *MapBox*, *OsmDroid* oppure *Google Maps*.

Per usufruire dei servizi dati da *Google Maps* è necessario seguire i seguenti step:

- **Step 1**, installare e impostare *Google Play Service SDK*
- **Step 2**, ottenere una chiave *Google Play API* valida per utilizzare la libreria
- **Step 3**, configurare la propria applicazione affinchè possa utilizzare *SDK* di *Google Maps*. Ciò avviene specificando all'interno della sezione *meta* del *Manifest* la *Key API* ottenuta precedentemente.

```
<meta-data
  android:name="com.google.android.gms.version"
  android:value="@integer/google_play_services_version"/>
```

```
<meta-data
  android:name="com.google.android.geo.API_KEY"
  android:value="API_activation_key"/>
```

Tipicamente *Google Maps* all'interno del progetto elaborato si riduce in un **fragment**, ossia una porzione modulare dell'interfaccia utente di una certa attività, il quale implementa la classe *SupportMapFragment*.

Pertanto, è riportato all'interno di un'*attività* un *MapFragment*, come da *XML* sottostante.

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/map"
  android:name="com.google.android.gms.maps.SupportMapFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

Successivamente è formulata la richiesta affinchè il *fragment* possa disegnare la mappa.

```
val mapFragment = supportFragmentManager.findFragmentById(R.id.map)
  as? SupportMapFragment
mapFragment?.getMapAsync(this)
```

Proseguendo, la mappa è poi immessa all'interno di un oggetto *Google Map* dedicato, restituito mediante una funzione di callback.

```
class MainActivity : AppCompatActivity(), OnMapReadyCallback {
```