Developing Location-Aware Android Apps with the Google Location Services API

Catherine Easdon

1 Introduction

There are many different contexts in which you might need to add location-awareness to your Android app. You might be developing an app specifically designed to track location (for example a mapping or car GPS app), or an app where location data is not required but could enhance the user experience (such as a weather forecasting or city tour guide app). Knowing the user's location allows you to deliver more relevant and more useful information to your users.

However, implementing location-awareness on mobile Android devices can be challenging. The sheer variety of implementation strategies is initially overwhelming and, depending on your chosen strategy, there may be many intricate details to consider such as managing run-time permissions, balancing trade-offs in accuracy, speed, and battery-efficiency when choosing a location source, and accounting for user movement and stale cached location fixes. This article focuses on querying the user's location using the Google Play services location APIs, and also discusses alternative implementation strategies.

The full code for the app developed in this article is available on GitHub[1]. Note that this article assumes some basic familiarity with Android Studio and with Android app development; the Android developer site provides comprehensive documentation and a tutorial for beginners [2].

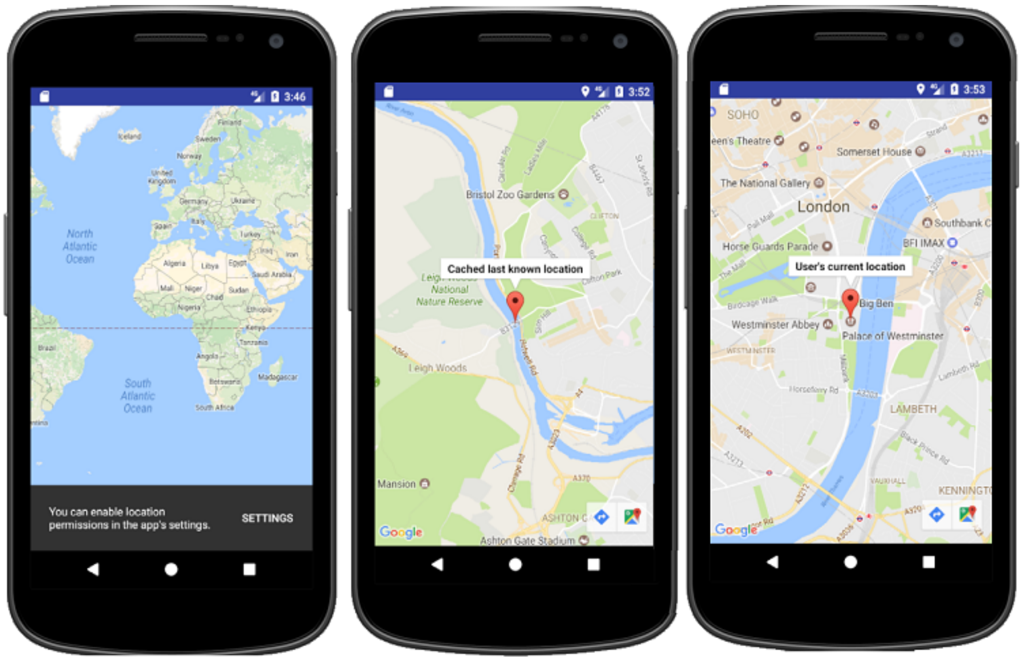
You can enable location permissions in the app's settings.
SETTINGS
Cached last known location
User's current location

Figure 1: Screenshots from the finished app; teleportation ability from Bristol to London kindly provided by the Android emulator!

2 Using the Google Play services location APIs

2.1 Setup - Dependencies

To begin, start a new project in Android Studio using the `MapsActivity` template. For this article, our new activity will be called `LocationActivity`. You'll need to obtain a Google Maps API key[3] and insert this into `google_maps_api.xml`. There is no API key in this article's GitHub repository for security reasons, as each key is uniquely linked to a given developer or project; you should follow the instructions in the Developer Console to restrict access to your key to your new app only to prevent it being abused. Using the Google Maps API will allow us to visualise our obtained location fixes on a map rather than dealing with the raw latitude and longitude data. Now build your app and run it on a mobile device or in the Android emulator; you should see a full-screen Google Map centred on a marker on Sydney, Australia, as produced by Android Studio's demo code. Note that you will need Google Play Services installed in Android Studio for the setup wizard to initialise the app correctly; the API documentation provides a guide[4].

Take a look at the app's `AndroidManifest.xml` and you should see `android.permission.ACCESS_FINE_LOCATION`. Android has two location permissions, `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION`. The `MapsActivity` template has added this permission in by default here, but in a blank project you would need to insert the code for either one of the location permissions in order to access the user's location. Note that with the coarse location permission, GPS location fixes will be accurate to within approximately a city block; consider what level of accuracy your app requires and if a coarse location would be acceptable then use this rather than the fine location permission, as to avoid privacy concerns it is best practice to avoid requesting any more information from your users than necessary. For this demo app, I'll assume we need very accurate data and continue using the fine location permission.

You'll also need to add Google Play location services (`com.google.android.gms:play-services-location:10.2.4`, for location data) and the Android design support library (`com.android.support:design:25.3.1`, to provide us with some Material Design UI components) as dependencies in your app's `build.gradle`. Change the version numbers as needed if Android Studio tells you there is an updated release available.

2.2 Setup - Google API Client

Please see Appendix A for the described code.

The first step is to create a new `GoogleApiClient` object in your activity's `onCreate()` method, and then create override methods of `onStart()` and `onStop()` so that your client connects when the app starts and disconnects when it stops. It's very important to disconnect the client when you no longer need it, as otherwise the continued unnecessary requests could lead to battery drain and excess data usage. Once you've added the `onCreate()` code you'll find you need to make the class implement `GoogleApiClient.ConnectionCallbacks` in order to pass it as an argument to the API Client builder. And of course, now that your class is implementing this interface, it needs to implement its methods, namely `onConnected()` and `onConnectionSuspended()`. We're just setting up the client at the moment, so we'll leave `onConnected()` empty for now, but this is where we'll start requesting a location fix later; `onConnectionSuspended()` should try to reconnect to the client and log a debug message so we can keep track of when this occurs. Finally, we need to make the class also implement `GoogleApiClient.OnConnectionFailedListener` and provide its `onConnectionFailed()` method, simply logging the error code for debugging.

2.3 Getting the User's Last Known Location

Please see Appendix A for the described code.

Now all the dependencies have been resolved, our first step will be to fetch the user's last known location. Location updates can come from a variety of sources on Android, each with their own advantages and disadvantages. For example, WiFi location fixes are less accurate than GPS, but can be more reliably obtained indoors and normally consume less battery power. Figure 2 below is based on the older `android.location` API, but gives a good impression of how location updates vary in source and accuracy.

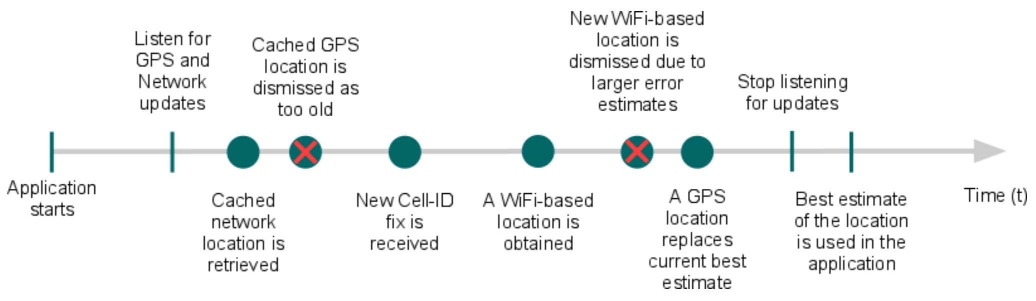
Application starts
Listen for GPS and Network updates
Cached network location is retrieved
Cached GPS location is dismissed as too old
New Cell-ID fix is received
A WiFi-based location is obtained
New WiFi-based location is dismissed due to larger error estimates
A GPS location replaces current best estimate
Stop listening for updates
Best estimate of the location is used in the application
Time (t)

Figure 2: An example timeline of location updates.[5]

Unfortunately, we now need add more boilerplate code, as requesting location data means handling run-time permissions. Beginning in Android 6.0 (API level 23), permissions are granted while an app is running and not during installation. If your users are running API level 22 or below, adding a location permission to your manifest file is all you need to do; users will have to grant that permission during installation, and otherwise the app will not install. However, on newer devices you need to check for permissions, request permissions if needed, and handle the case where the user denies your app location permissions; and this must be done each time your app is run, as users can revoke permissions in the app's settings at any time.

Let's begin by putting a run-time permission check into `onConnected()`. If the user is running API level 23 or higher and we haven't been granted the `ACCESS_FINE_LOCATION` permission, we'll request that permission. For this you need to define a `locationPermissionsRequest` integer constant; this can be any number, and is used to distinguish different permission requests (if you make multiple ones) in `onRequestPermissionsResult()`, the callback called when `requestPermissions()` returns a result. If the user is running API level 22 or below, or we have been granted the run-time permission, we'll call `getLocation()`, the method containing our location update code which we'll define later. In `onRequestPermissionsResult()`, we'll call `getLocation()` if the request was successful, and `askUserForLocationPermission()` if it was unsuccessful. If you define `getLocation()` and `askUserForLocationPermission()` as empty methods for now and run the app now on a device running API level 23 or higher, you'll see a pop-up window asking you to allow the app access to the device's location. But selecting either deny or allow will appear to do nothing yet - we need to define those two empty methods!

For `getLocation()` we'll use the `FusedLocationApi` to get the user's last known (cached) location, create a marker for this location and add it to the app's Google map, and then animate the camera to zoom into this location at a zoom level of 14. Note the check to make sure `cachedLocation` isn't null to avoid an exception - it's surprisingly common for `getLastLocation()` to return null because the phone hasn't yet received a location fix (for example, if the phone has just been restarted). Alternatively, it might instead be several hours out of date, depending on when any apps last requested a location fix. If you test the app now and allow permissions, you can see if your phone currently has a cached last known location. Note if you are testing the app in the Android emulator, you may need to manually send the latitude and longitude of a fake location fix from the emulator's settings menu.

`askUserForLocationPermission()` is called if the user denies location permissions. You can respond to this in any way you wish; the provided code in this example creates a snackbar which tells the user the app won't work properly because they denied location permissions. When they click 'ok' to this snackbar, a new snackbar appears telling them they can enable location permissions in the app's settings, and the only button available is one which opens the app's settings menu. This might seem like quite an aggressive way to ask the user to grant permissions, but if your app depends on location data your users won't be able to use the app unless they allow the permissions anyway.

2.4 Getting Regular Location Updates

Please see Appendix C for the described code.

Of course, most location-aware apps will require much more accurate location tracking than a cached location that is potentially hours out-of-date. We can request regular location updates from the `FusedLocationApi` to get an up-to-date location fix. We'll create a `LocationRequest` object and pass it as a parameter to `requestLocationUpdates()` in `getLocation()` beneath our earlier code with the cached location. This will call an `onLocationChanged()` callback each time a location update is available, which we'll add to the class. The class needs to implement `LocationListener` and then we can pass it as a parameter to `requestLocationUpdates()` to use our callback.

For the same reasons why it's crucial you disconnect the Google API Client when you no longer need to be connected, you should also ensure you call `removeLocationUpdates()` when you no longer require the user's location, and think carefully about which settings to use in your initial `LocationRequest`. In this example, I'm assuming the app needs frequent location updates; using these settings will drain the battery much faster than, for example, setting the priority as `PRIORITY_LOW_POWER` (resulting in accuracy to the nearest 10km) and setting the interval to 60 seconds (note it must be set in milliseconds). `setFastestInterval()` exists because other apps' location requests will affect the rate at which updates are sent; if another app is requesting extremely frequent updates and you don't set the fastest interval appropriately, your app may be unable to handle the frequency of the updates. See the documentation for more details about the available options.[6]

There are many other features available within the Google Play services location APIs, including geofencing (where events are triggered when users are within a specified radius of a location, see Figure 3) and geocoding (converting a street address to geographic coordinates, or vice versa); see the Android developer documentation for more details[7].

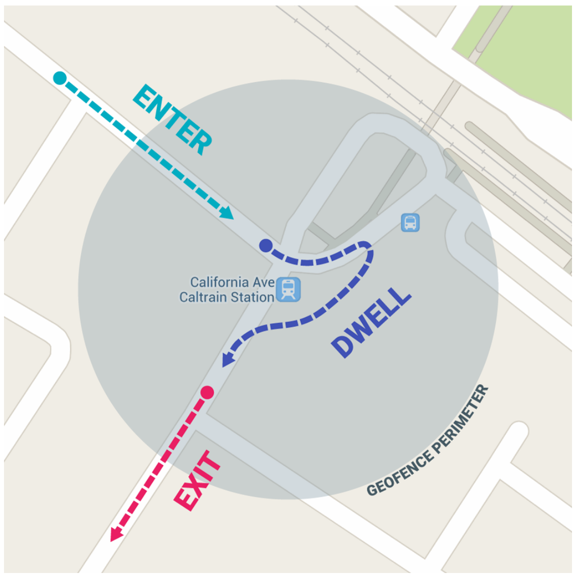

Figure 3: An example geofence.[8]

3 Alternatives

Unfortunately, this location request code can be rather unwieldy. What if your app has several activities which all need to access location data separately - do you need to clutter each activity with all this boilerplate code? One solution is to create a separate class to contain all of the runtime location permissions and Google API Client code, and then in each activity where location data is required initialise an instance of this new class and interact with it via function calls and callbacks.

I've found this works very well in practice on larger projects. You can see an example of this on this article's GitHub repo in the LocationHelper and MapActivity classes[1].

The Google Play services location APIs are the officially recommended implementation strategy for location-awareness, but there are alternatives available, including `android.location` and a variety of third-party libraries offering additional features and/or reducing the amount of boilerplate code required.

`android.location`[5] is the Android framework's location API. It's no longer recommended for development as the Google Play alternative is more fully-featured and easier to use, but it's worth taking a look at its documentation to see how much more complex implementing location-awareness used to be! If you are developing for legacy devices incompatible with Google Play services, you could try using `Android-SimpleLocation`[9], a third-party wrapper library for `android.location` which makes your life easier by hiding the under-the-hood implementation details.

One of the most popular third-party libraries is the `Smart Location Library`[10]. With this library, obtaining a fix for the last known location is as simple as:

```
Location lastLocation =
    SmartLocation.with(this).location().getLastLocation();
```

Note that you still do need to check you have run-time permissions before running this to avoid a security exception, but the library hides most of the Google API client and FusedLocationApi boilerplate. It also adds activity recognition functionality so that you can detect whether the user is walking, running, cycling, or driving and adjust the update accuracy settings accordingly.

References

[1] https://github.com/cattius/Location-Demo

[2] https://developer.android.com/training/index.html

[3] https://developers.google.com/maps/documentation/android-api/signup

[4] https://developers.google.com/android/guides/setup

[5] https://developer.android.com/guide/topics/location/strategies.htm; image used under the CC-BY-2.5 license, no modifications made.

[6] https://developer.android.com/training/location/change-location-settings.html

[7] https://developer.android.com/training/location/index.html

[8] https://developer.android.com/training/location/geofencing.html, image used under the CC-BY-2.5 license, no modifications made.

[9] https://github.com/delight-im/Android-SimpleLocation

[10] https://github.com/mrmans0n/smart-location-lib

Appendix A

This code accompanies the 'Setup - Google API Client' section.

```
public class LocationActivity extends FragmentActivity implements
    OnMapReadyCallback, GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener {
  private GoogleMap mMap;
  protected GoogleApiClient client;
  public static final String TAG = "LocationHelper";    // name of
     the class for debugging
  @Override
  protected void onCreate(Bundle savedInstanceState) {
  ...
    if (client == null) {
      client = new GoogleApiClient.Builder(this)
                          .addConnectionCallbacks(this)
                          .addOnConnectionFailedListener(this)
                          .addApi(LocationServices.API)
                          .build();
    }
  }
  @Override
  protected void onStart() {
    client.connect();
    super.onStart();
  }
  @Override
  protected void onStop() {
    client.disconnect();
    super.onStop();
  }
  @Override
  public void onConnected(Bundle connectionHint) {
    //TODO
  }
  @Override
  public void onConnectionSuspended(int reason){
    Log.d(TAG, "Connection to Google Play services lost, trying to
       reconnect");
    client.connect();
  }
  @Override
  public void onConnectionFailed(ConnectionResult result){
    Log.d(TAG, "Connection failed with error code " +
       result.getErrorCode());
  }
  ...   // leave the rest of the code that was added by default
}
```

Appendix B

This code accompanies the 'Getting the User's Last Known Location' section.

```
  ...
  private static final int locationPermissionsRequest = 20;
     //arbitrary constant
  private Location cachedLocation;
  private Context context = this;
  @Override
  public void onConnected(Bundle connectionHint) {
    if(Build.VERSION.SDK_INT > 22 &&
       ContextCompat.checkSelfPermission(context,
       Manifest.permission.ACCESS_FINE_LOCATION) !=
       PackageManager.PERMISSION_GRANTED){
      requestPermissions(new
         String[]{Manifest.permission.ACCESS_FINE_LOCATION},
         locationPermissionsRequest);
    }
    else { //we have permission - user granted at runtime if SDK
        version > 22 or if <= 22, user must have granted it at
        install time
      getLocation();
    }
  }
  @Override
  public void onRequestPermissionsResult(int requestCode, String
      permissions[], int[] grantResults){
    switch(requestCode) {
      case locationPermissionsRequest: {
        if (grantResults.length > 0 && grantResults[0] ==
            PackageManager.PERMISSION_GRANTED){
          //user gave us runtime permission
          getLocation();
        }
        else { //user denied runtime permission
          askUserForLocationPermission();
        }
        break;
        }
    }
  }
  @SuppressWarnings({"MissingPermission"}) //security exceptions
      ARE handled but in a way that confuses Android Studio
  private void getLocation(){
    cachedLocation =
        LocationServices.FusedLocationApi.getLastLocation(client);
    if(cachedLocation != null){
      if(mMap != null){
        LatLng cachedLocationLatLng = new
            LatLng(cachedLocation.getLatitude(),
            cachedLocation.getLongitude());
        mMap.addMarker(new
            MarkerOptions().position(cachedLocationLatLng).title("Cached
            last known location"));
        mMap.animateCamera(CameraUpdateFactory.newLatLngZoom
            (cachedLocationLatLng, 14.0f));
      }
      else {
        Log.d(TAG, "Map not initialised correctly");
            //this is unlikely to occur as the map initialises very
            quickly
      }
    }
    else{
      Log.d(TAG, "cached location is null");
    }
  }
 private void askUserForLocationPermission() {
   final View.OnClickListener settingsButton = new
       View.OnClickListener() {
     @Override
        public void onClick(View view){
          //open phone settings menu to so they can change the
              permission
          Intent intent = new
              Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS,
             Uri.parse("package:" + context.getPackageName()));
          intent.addCategory(Intent.CATEGORY_DEFAULT);
          intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
         context.startActivity(intent);
        }
     };
    View.OnClickListener okButton = new View.OnClickListener() {
      String userMsg = "You can enable location permissions in the
         app's settings.";
      @Override
      public void onClick(View view){
         Snackbar snackbar = Snackbar.make(view, userMsg,
            Snackbar.LENGTH_INDEFINITE).setAction("Settings",
            settingsButton).setActionTextColor(WHITE);
         snackbar.show();
      }
    };
   View currentParentView = ((Activity)
      context).findViewById(android.R.id.content);
     String userMsg = "You've denied location permissions. This
        app won't function correctly.";
     Snackbar snackbar = Snackbar.make(currentParentView, userMsg,
        Snackbar.LENGTH_INDEFINITE).setAction("OK",
        okButton).setActionTextColor(WHITE);
     snackbar.show();
    }
  ...
```

Appendix C

This code accompanies the 'Getting Regular Location Updates' section.

```
import com.google.android.gms.common.api.Status;
  ...
  private void getLocation(){
    ...
    private LocationRequest locationRequest =
       LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
            .setInterval(10 * 1000)        // 10 seconds, in
               milliseconds
            .setFastestInterval(1 * 1000); // 1 second, in
               milliseconds
    PendingResult<Status> newLocation =
       LocationServices.FusedLocationApi.requestLocationUpdates(client,
       locationRequest, this);
    //if you are testing this in the emulator, you need to
       manually send the phone a GPS fix from the settings menu
       after this code runs
  ...
  @Override
  //Called when we receive a location update from PendingResult
  public void onLocationChanged(Location location){
    if(mMap != null){
      LatLng currentLocation = new LatLng(location.getLatitude(),
         location.getLongitude());
    mMap.addMarker(new
        MarkerOptions().position(currentLocation).title("User's
        current location"));
    mMap.animateCamera(CameraUpdateFactory.newLatLngZoom
        (currentLocation, 14.0f));
  }
  else {
    Log.d(TAG, "Map not initialised correctly"); //this is
        highly unlikely to occur as getting a location fix is
        slow in comparison to initialising the map
  }
}
```