Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Android Google Services Part 2

Location & Geocoding

Google Services Overview

- Overview of Google Play Services and Setup
- Detailed look at
  - Google+ Sign-in and Authentication (Part 1)
  - Location & Geocoding (Part 2)
  - Google Maps (Part 3)

Google Services Overview

- Detailed look at
  - Location & Geocoding (Part 2)

Agenda *

- Finding your Location with Location-Based Services (LBS) & Fused Location Provider
- Overview of GeoFencing & Activitiy Recognition
- Installation & Registration of Google Maps API ‘Key’
- Creating interactive Maps with GoogleMaps, SupportMapFragments & FragmentActivitiys
- Creating & Adding Markers to Maps

Introduction

- One of the defining features of mobile phones is their portability, so it's not surprising that some of the most enticing Android features are the services that let you find, contextualize, and map physical locations
  - Using Location-Based Services / Fused Location Provider
    - you can find the device's current location (GPS, Network Provider etc.)
    - send notifications when the device's location is 'near' some other location, (via proximity alerts or GeoFencing)
  - Using Google Maps (Part 3) you can
    - create map-based Activities as a UI element with full access, allowing you to zoom in/out/pan, control display settings etc.
    - using Markers, you can annotate the map and handle touch/tap events

Overview of Location-Based Services

- Location-based services use real-time location data from a mobile device or smartphone to provide information, entertainment, or security.
- Location-Based services are available on most smartphones, and a majority of smartphone owners use location-based services.
- Many popular applications integrate location-based services. Examples include
  - Google Maps, TripAdvisor, Starbucks, The Weather Channel, Navigation, Facebook Places, CoffeeMate ☺

Overview of Location Providers

- GPS is accurate, but
  - it only works outdoors
  - it quickly consumes battery power
  - it doesn't return the location quickly
- Android's Network (Fused) Location Provider determines user location using Cell Towers and Wi-Fi signals. It is less accurate than GPS, but
  - it works indoors and outdoors
  - it responds faster
  - and it uses less battery power

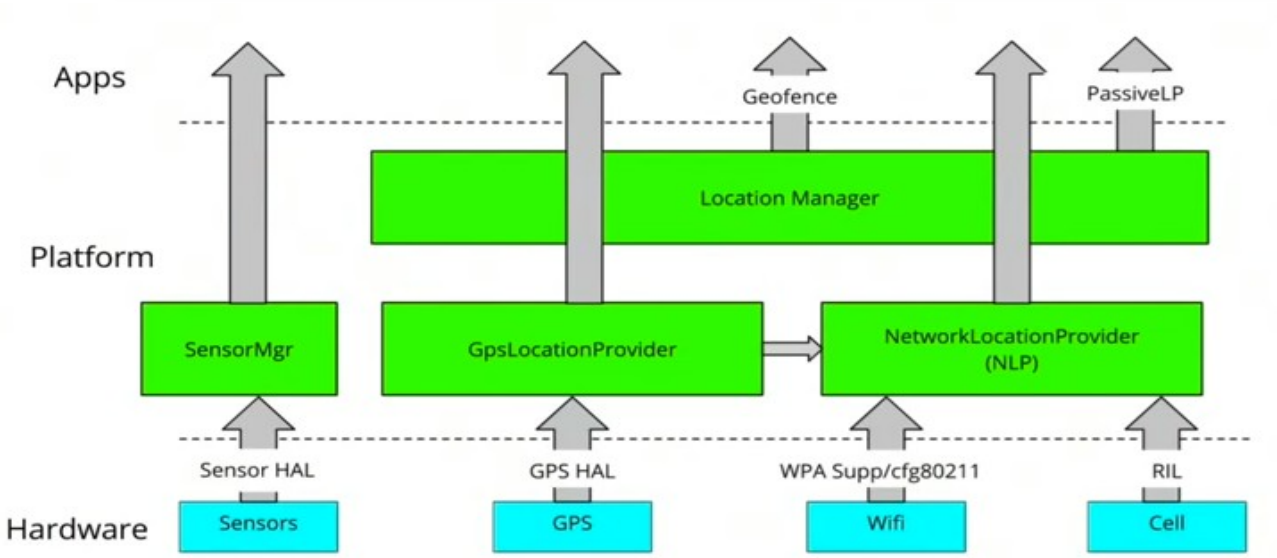
The Fused Location Provider

- The location APIs in Google Play services contains a fused location provider
- The fused location provider manages the underlying location technology and provides a simple API that
  - allows you to specify requirements at a high level, like high accuracy or low power
  - optimizes the device's use of battery power

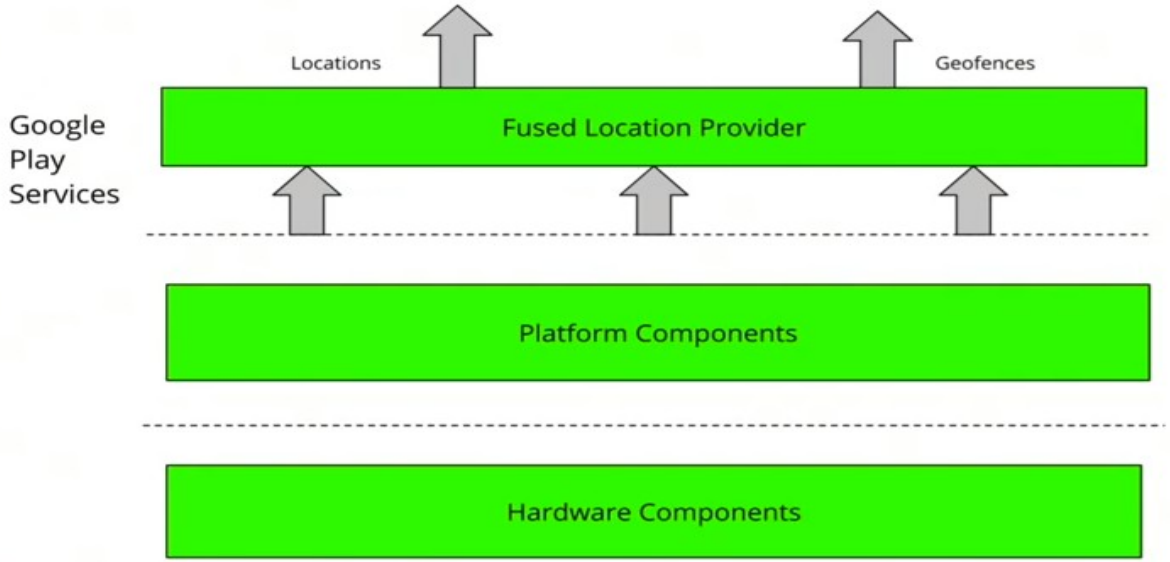
Fused Location Provider

- The goal of Fused Location Provider ('Fused') is to lessen the workload of developers who want to interact with location information
- Provides a single programmable interface
- Google does the hard work in sourcing location, simply feeding it to developers' applications (via Google Play Services)
  - Fused brings together cellular, WiFi, GPS, and Sensor data

Fused Location Provider



Before Android 4.2



After

- Simplified API
  - 3 main aspects were worked on
    - Speed
    - Accuracy
    - Coverge

Fused Location Provider & Priority Modes

- A user can define one of the 3 main fused location provider modes by setting priority:
  - HIGH_ACCURACY, BALANCED_POWER or NO_POWER
- During a Google IO presentation a chart was presented showing effect of different priorities of the recognition algorithm as tested multiple times on a Galaxy Nexus.

Priority	Typical Interval	Battery Drain per Hour (%)	Accuracy*
HIGH_ACCURACY	5 seconds	7.25%	~10 meters
BALANCED_POWER	20 seconds	0.6%	~40 meters
NO_POWER	N/A	small	~1 mile?

Challenges in Determining User Location

- **Multitude of location sources**

  GPS, Cell-ID, and Wi-Fi can each provide a clue to users location. Determining which to use and trust is a matter of trade-offs in accuracy, speed, and battery-efficiency.

- **User Movement**

  Because the user location changes, you must account for movement by re-estimating user location every so often.

- **Varying Accuracy**

  Location estimates from each location source are not consistent in their accuracy. A location obtained 10 seconds ago from one source might be more accurate than the newest location from another or same source.

Part 2
Location & Geocoding

Making Your App Location-Aware

Overview

- One of the unique features of mobile applications is location awareness. Mobile users take their devices with them everywhere, and adding location awareness to your app offers users a more contextual experience.
- The location APIs available in Google Play Services facilitate adding location awareness to your app with automated location tracking, geofencing, and activity recognition.

Overview - Location-Based Services in Android

- Android provides two location frameworks
  - in package `android.location`
  - in package `com.google.android.gms.location`
    (part of Google Play Services)
- The framework provided by Google Play Services is now the preferred way to add location-based services to an application.
  - simpler API – greater accuracy
  - more power efficient – more versatile

Note that some classes in package `android.location` are still used by the Google Play Services API.

Location Awareness - Your “Need to Know”

1. Getting the Last Known Location
   - how to retrieve the last known location of an Android device, which is usually equivalent to the user's current location.
2. Changing Location Settings
   - how to detect and apply system settings for location features.
3. Receiving Location Updates
   - how to request and receive periodic location updates.
4. Displaying a Location Address
   - how to convert a location's latitude and longitude into an address (reverse geocoding).
5. Creating and Monitoring Geofences
   - how to define one or more geographic areas as locations of interest, called geofences, and detect when the user is close to or inside a geofence.

1. Getting the Last Know Location

- Using the Google Play services location APIs, your app can request the last known location of the user's device.
- In most cases, you are interested in the user's current location, which is usually equivalent to the last known location of the device.
- Specifically, use the fused location provider to retrieve the device's last known location. The Steps involved are :
  - Setup Google Play Services (should be done already…)
  - Specify App Permissions
  - Connect to Google Play Services
  - Get the Users Last Known Location

1. Getting the Last Know Location

Specify App Permissions

Apps that use location services must request location permissions. Android offers two location permissions: `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. The permission you choose determines the accuracy of the location returned by the API. If you specify `ACCESS_COARSE_LOCATION`, the API returns a location with an accuracy approximately equivalent to a city block.

This lesson requires only coarse location. Request this permission with the `uses-permission` element in your app manifest, as the following code snippet shows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.basiclocationsample" >

  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
</manifest>
```

1. Getting the Last Know Location *

Connect to Google Play Services

To connect to the API, you need to create an instance of the Google Play services API client. For details about using the client, see the guide to Accessing Google APIs.

In your activity's `onCreate()` method, create an instance of Google API Client, using the `GoogleApiClient.Builder` class to add the `LocationServices` API, as the following code snippet shows.

```
// Create an instance of GoogleAPIClient.
if (mGoogleApiClient == null) {
    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(LocationServices.API)
        .build();
}
```

1. Getting the Last Know Location

Connect to Google Play Services

To connect, call `connect()` from the activity's `onStart()` method. To disconnect, call `disconnect()` from the activity's `onStop()` method. The following snippet shows an example of how to use both of these methods.

```
protected void onStart() {
    mGoogleApiClient.connect();
    super.onStart();
}

protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

1. Getting the Last Know Location *

To request the last known location, call the `getLastLocation()` method, passing it your instance of the `GoogleApiClient` object. Do this in the `onConnected()` callback provided by Google API Client, which is called when the client is ready. The following code snippet illustrates the request and a simple handling of the response:

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
        mLastLocation = LocationServices.FusedLocationApi.getLastLocation(
                mGoogleApiClient);
        if (mLastLocation != null) {
            mLatitudeText.setText(String.valueOf(mLastLocation.getLatitude()));
            mLongitudeText.setText(String.valueOf(mLastLocation.getLongitude()));
        }
    }
}
```

The `getLastLocation()` method returns a `Location` object from which you can retrieve the latitude and longitude coordinates of a geographic location. The location object returned may be null in rare cases when the location is not available.

2. Changing Location Settings

Set Up a Location Request

Create the location request and set the parameters as shown in this code sample:

```
protected void createLocationRequest() {
    LocationRequest mLocationRequest = new LocationRequest();
    mLocationRequest.setInterval(10000);
    mLocationRequest.setFastestInterval(5000);
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
}
```

The priority of `PRIORITY_HIGH_ACCURACY`, combined with the `ACCESS_FINE_LOCATION` permission setting that you've defined in the app manifest, and a fast update interval of 5000 milliseconds (5 seconds), causes the fused location provider to return location updates that are accurate to within a few feet. This approach is appropriate for mapping apps that display the location in real time.

> **Performance hint:** If your app accesses the network or does other long-running work after receiving a location update, adjust the fastest interval to a slower value. This adjustment prevents your app from receiving updates it can't use. Once the long-running work is done, set the fastest interval back to a fast value.

2. Changing Location Settings

Get Current Location Settings

Once you have connected to Google Play services and the location services API, you can get the current location settings of a user's device. To do this, create a `LocationSettingsRequest.Builder`, and add one or more location requests. The following code snippet shows how to add the location request that was created in the previous step:

```
LocationSettingsRequest.Builder builder = new LocationSettingsRequest.Builder()
     .addLocationRequest(mLocationRequest);
```

Next check whether the current location settings are satisfied:

```
PendingResult<LocationSettingsResult> result =
         LocationServices.SettingsApi.checkLocationSettings(mGoogleClient,
                 builder.build());
```

When the `PendingResult` returns, your app can check the location settings by looking at the status code from the `LocationSettingsResult` object. To get even more details about the the current state of the relevant location settings, your app can call the `LocationSettingsResult` object's `getLocationSettingsStates()` method.

3. Receiving Location Updates

- If your app can continuously track location, it can deliver more relevant information to the user.
  - For example, if your app helps the user find their way while walking or driving, or if your app tracks the location of assets, it needs to get the location of the device at regular intervals. As well as the geographical location (latitude and longitude), you may want to give the user further information such as the bearing (horizontal direction of travel), altitude, or velocity of the device.
  - This information, and more, is available in the Location object that your app can retrieve from the fused location provider.

3. Receiving Location Updates

Request Location Updates

Before requesting location updates, your app must connect to location services and make a location request. The lesson on Changing Location Settings shows you how to do this. Once a location request is in place you can start the regular updates by calling `requestLocationUpdates()`. Do this in the `onConnected()` callback provided by Google API Client, which is called when the client is ready.

Depending on the form of the request, the fused location provider either invokes the `LocationListener.onLocationChanged()` callback method and passes it a `Location` object, or issues a `PendingIntent` that contains the location in its extended data. The accuracy and frequency of the updates are affected by the location permissions you've requested and the options you set in the location request object.

This lesson shows you how to get the update using the `LocationListener` callback approach. Call `requestLocationUpdates()`, passing it your instance of the `GoogleApiClient`, the `LocationRequest` object, and a `LocationListener`. Define a `startLocationUpdates()` method, called from the `onConnected()` callback, as shown in the following code sample:

3. Receiving Location Updates *

Request Location Updates

```
@Override
public void onConnected(Bundle connectionHint) {
    ...
    if (mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}

protected void startLocationUpdates() {
    LocationServices.FusedLocationApi.requestLocationUpdates(
            mGoogleApiClient, mLocationRequest, this);
}
```

3. Receiving Location Updates *

Define the Location Update Callback

The fused location provider invokes the `LocationListener.onLocationChanged()` callback method. The incoming argument is a `Location` object containing the location's latitude and longitude. The following snippet shows how to implement the `LocationListener` interface and define the method, then get the timestamp of the location update and display the latitude, longitude and timestamp on your app's user interface:

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener, LocationListener {
    ...
    @Override
    public void onLocationChanged(Location location) {
        mCurrentLocation = location;
        mLastUpdateTime = DateFormat.getTimeInstance().format(new Date());
        updateUI();
    }

    private void updateUI() {
        mLatitudeTextView.setText(String.valueOf(mCurrentLocation.getLatitude()));
        mLongitudeTextView.setText(String.valueOf(mCurrentLocation.getLongitude()));
        mLastUpdateTimeTextView.setText(mLastUpdateTime);
    }
}
```

3. Receiving Location Updates *

Stop Location Updates

Consider whether you want to stop the location updates when the activity is no longer in focus, such as when the user switches to another app or to a different activity in the same app. This can be handy to reduce power consumption, provided the app doesn't need to collect information even when it's running in the background. This section shows how you can stop the updates in the activity's `onPause()` method.

To stop location updates, call `removeLocationUpdates()`, passing it your instance of the `GoogleApiClient` object and a `LocationListener`, as shown in the following code sample:

```
@Override
protected void onPause() {
    super.onPause();
    stopLocationUpdates();
}

protected void stopLocationUpdates() {
    LocationServices.FusedLocationApi.removeLocationUpdates(
            mGoogleApiClient, this);
}
```

3. Receiving Location Updates

Stop Location Updates

Use a boolean, `mRequestingLocationUpdates`, to track whether location updates are currently turned on. In the activity's `onResume()` method, check whether location updates are currently active, and activate them if not:

```
@Override
public void onResume() {
    super.onResume();
    if (mGoogleApiClient.isConnected() && !mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}
```

3. Receiving Location Updates

Save the State of the Activity

A change to the device's configuration, such as a change in screen orientation or language, can cause the current activity to be destroyed. Your app must therefore store any information it needs to recreate the activity. One way to do this is via an instance state stored in a `Bundle` object.

The following code sample shows how to use the activity's `onSaveInstanceState()` callback to save the instance state:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putBoolean(REQUESTING_LOCATION_UPDATES_KEY,
            mRequestingLocationUpdates);
    savedInstanceState.putParcelable(LOCATION_KEY, mCurrentLocation);
    savedInstanceState.putString(LAST_UPDATED_TIME_STRING_KEY, mLastUpdateTime);
    super.onSaveInstanceState(savedInstanceState);
}
```

Define an `updateValuesFromBundle()` method to restore the saved values from the previous instance of the activity, if they're available. Call the method from the activity's `onCreate()` method, as shown in the following code sample:

3. Receiving Location Updates

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

private void updateValuesFromBundle(Bundle savedInstanceState) {
    if (savedInstanceState != null) {
        // Update the value of mRequestingLocationUpdates from the Bundle, and
        // make sure that the Start Updates and Stop Updates buttons are
        // correctly enabled or disabled.
        if (savedInstanceState.keySet().contains(REQUESTING_LOCATION_UPDATES_KEY)) {
            mRequestingLocationUpdates = savedInstanceState.getBoolean(
                    REQUESTING_LOCATION_UPDATES_KEY);
            setButtonsEnabledState();
        }

        // Update the value of mCurrentLocation from the Bundle and update the
        // UI to show the correct latitude and longitude.
        if (savedInstanceState.keySet().contains(LOCATION_KEY)) {
            // Since LOCATION_KEY was found in the Bundle, we can be sure that
            // mCurrentLocationis not null.
            mCurrentLocation = savedInstanceState.getParcelable(LOCATION_KEY);
        }

        // Update the value of mLastUpdateTime from the Bundle and update the UI.
        if (savedInstanceState.keySet().contains(LAST_UPDATED_TIME_STRING_KEY)) {
            mLastUpdateTime = savedInstanceState.getString(
                    LAST_UPDATED_TIME_STRING_KEY);
        }
        updateUI();
    }
}
```

4. Displaying a Location Address

- Getting the Last Known Location and Receiving Location Updates describe how to get the user's location in the form of a Location object that contains latitude and longitude coordinates.
- Although latitude and longitude are useful for calculating distance or displaying a map position, in many cases the address of the location is more useful.
  - For example, if you want to let your users know where they are or what is close by, a street address is more meaningful than the geographic coordinates (latitude/longitude) of the location.

4. Displaying a Location Address

- Using the **Geocoder** class in the Android framework location APIs, you can convert an address to the corresponding geographic coordinates. This process is called *geocoding*. Alternatively, you can convert a geographic location to an address. The address lookup feature is also known as *reverse geocoding*.
- The **getFromLocation()** method to convert a geographic location to an address. The method returns an estimated street address corresponding to a given latitude and longitude.

4. Displaying a Location Address

- The steps necessary are as follows:
  - Get a Geographic Location
  - Define an Intent Service to Fetch the Address
    - Define the Intent Service in your App Manifest
    - Create a Geocoder
    - Retrieve the street address data
    - Return the address to the requestor
  - Start the Intent Service
  - Receive the Geocoding Results
- For a Full discussion (and examples) visit https://developer.android.com/training/location/display-address.html

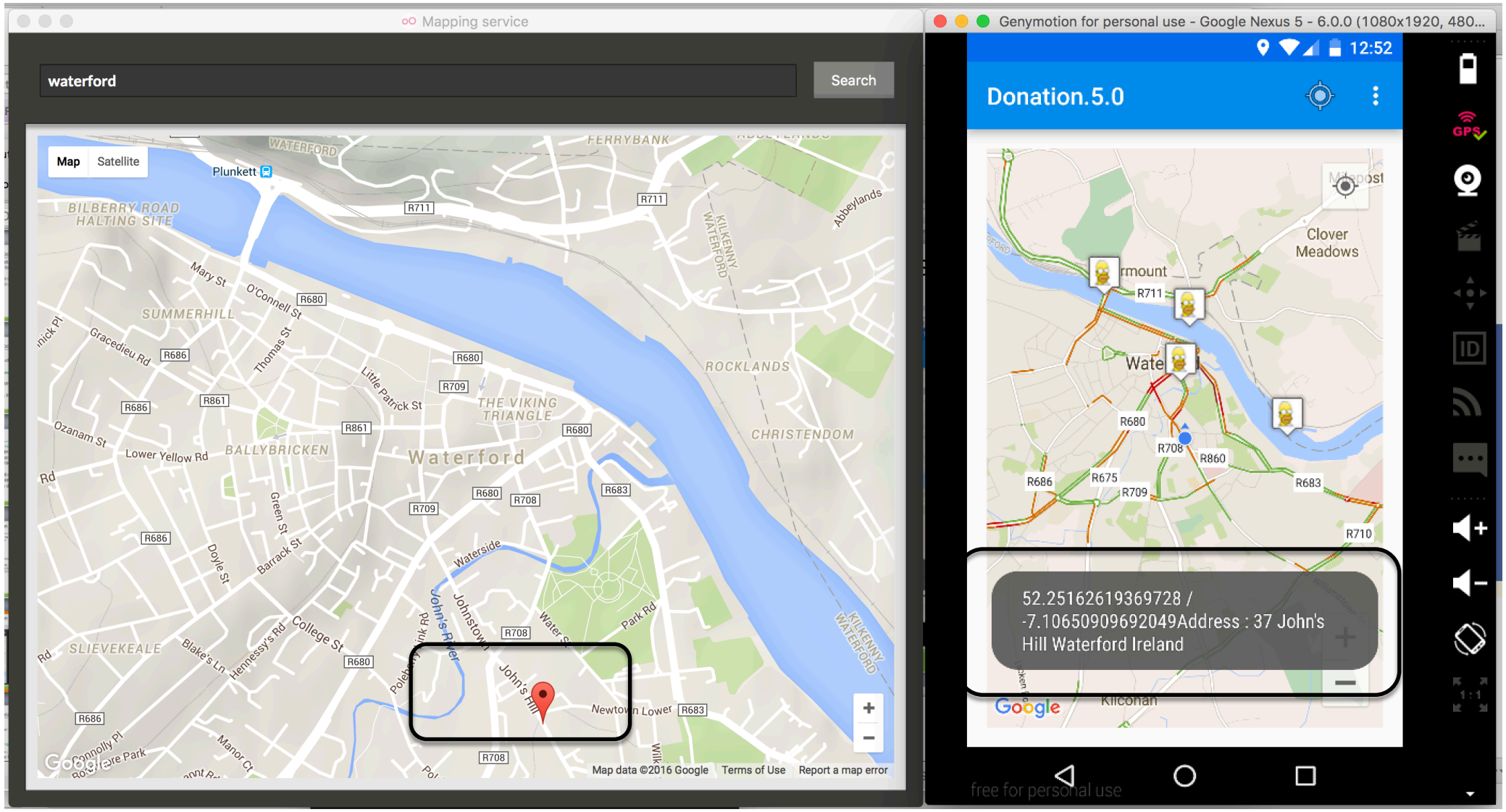
Example: Translating a Location to an Address
(Reverse Geocoding)

```
private String getAddressFromLatLng( LatLng latLng ) {
        Geocoder geocoder = new Geocoder( this );

        String strAddress = "";
        Address address;
        try {
            address = geocoder
                    .getFromLocation( latLng.latitude, latLng.longitude, 1 )
                    .get( 0 );
            strAddress = address.getAddressLine(0) +
                    " " + address.getAddressLine(1) +
                    " " +address.getAddressLine(2);
        }
        catch (IOException e ) {
        }

        return strAddress;
    }
```

Example: Translating a Location to an Address
(Reverse Geocoding)



Translating an Address to a Location (Geocoding)

- Create a string with the address

```
String addressStr =
    "171 Moultrie Street, Charleston, SC, 29409";
```

- Create a `Geocoder` instance

```
Geocoder geocoder = new Geocoder(this);
```

- Call the `Geocoder` method `getFromLocationName()`

```
List<Address> addresses =
    geocoder.getFromLocationName(addressStr, 1);
```

- Retrieve the latitude and longitude from the first address

```
Address address = addresses.get(0);
// call address.getLatitude() and
// address.getLongitude() as needed
```

Example: Geocoding

5. Creating and Monitoring Geofences

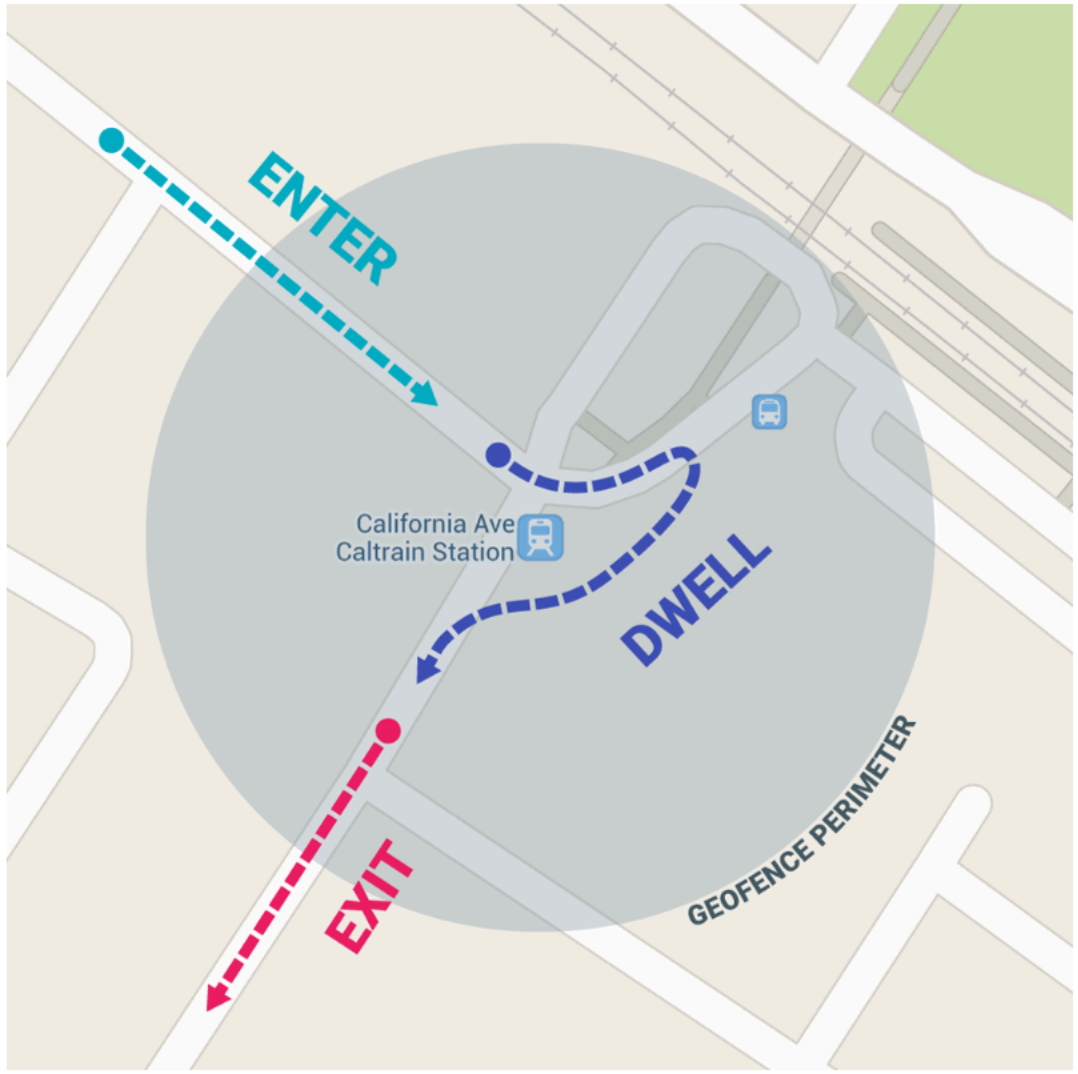
- Geofencing combines awareness of the user's current location with awareness of the user's proximity to locations that may be of interest.
- To mark a location of interest, you specify its latitude and longitude. To adjust the proximity for the location, you add a radius. The latitude, longitude, and radius define a geofence, creating a circular area, or fence, around the location of interest.

5. Creating and Monitoring Geofences

- You can have multiple active geofences, with a limit of 100 per device user.
- For each geofence, you can ask Location Services to send you entrance and exit events, or you can specify a duration within the geofence area to wait, or *dwell*, before triggering an event.
- You can limit the duration of any geofence by specifying an expiration duration in milliseconds. After the geofence expires, Location Services automatically removes it.

5. Creating and Monitoring Geofences

- Entrance
- Dwell
- Exit events



5. Creating and Monitoring Geofences

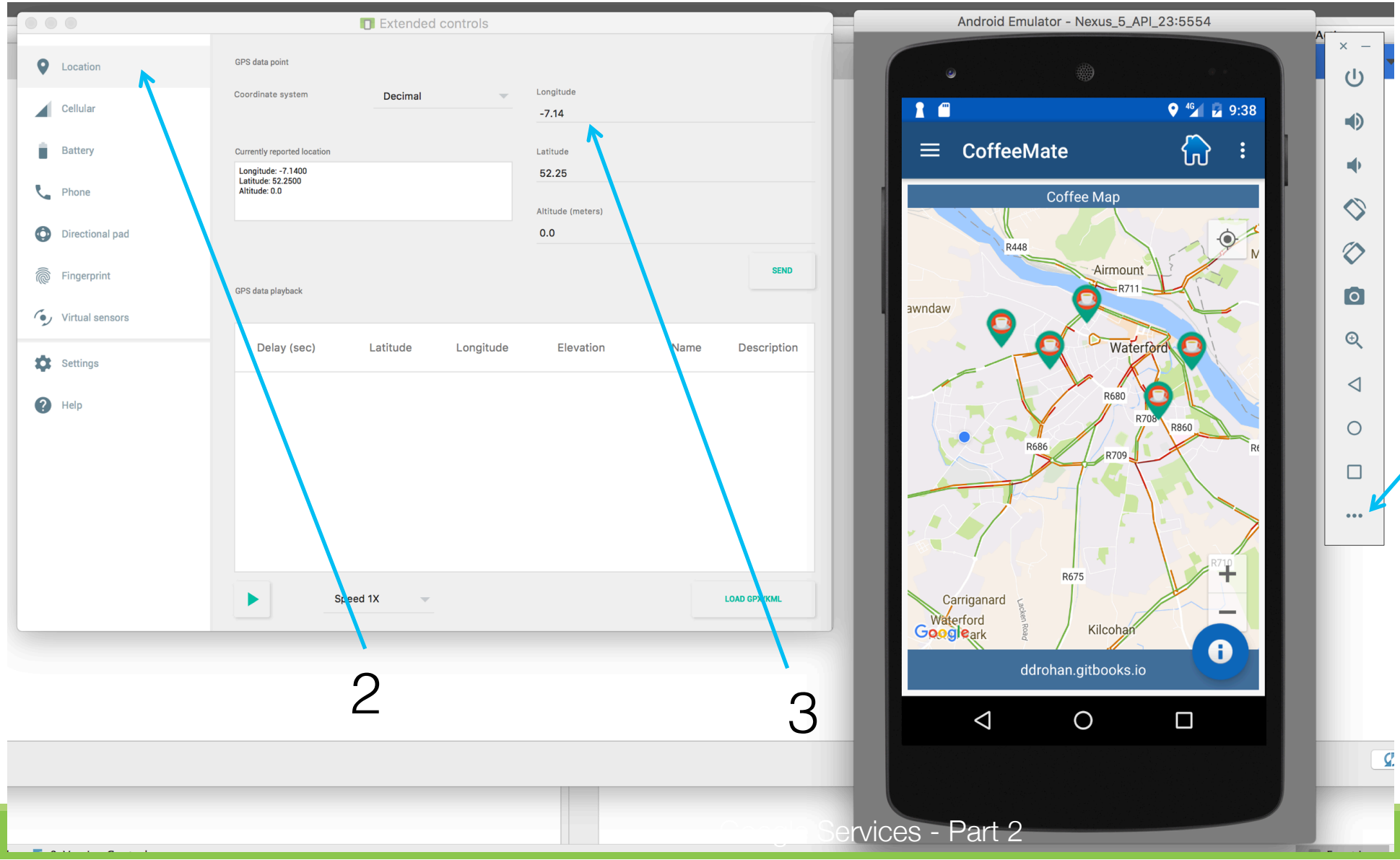
- The steps necessary are as follows:
  - Set up for Geofence Monitoring
  - Create and Add Geofences
    - Create geofence objects
    - Specify geofences and initial triggers
    - Define an Intent for geofence transitions
    - Add geofences
  - Handle Geofence Transitions
  - Stop Geofence Monitoring
- For a Full discussion (and examples) visit https://developer.android.com/training/location/geofencing.html

Testing Google Play Services

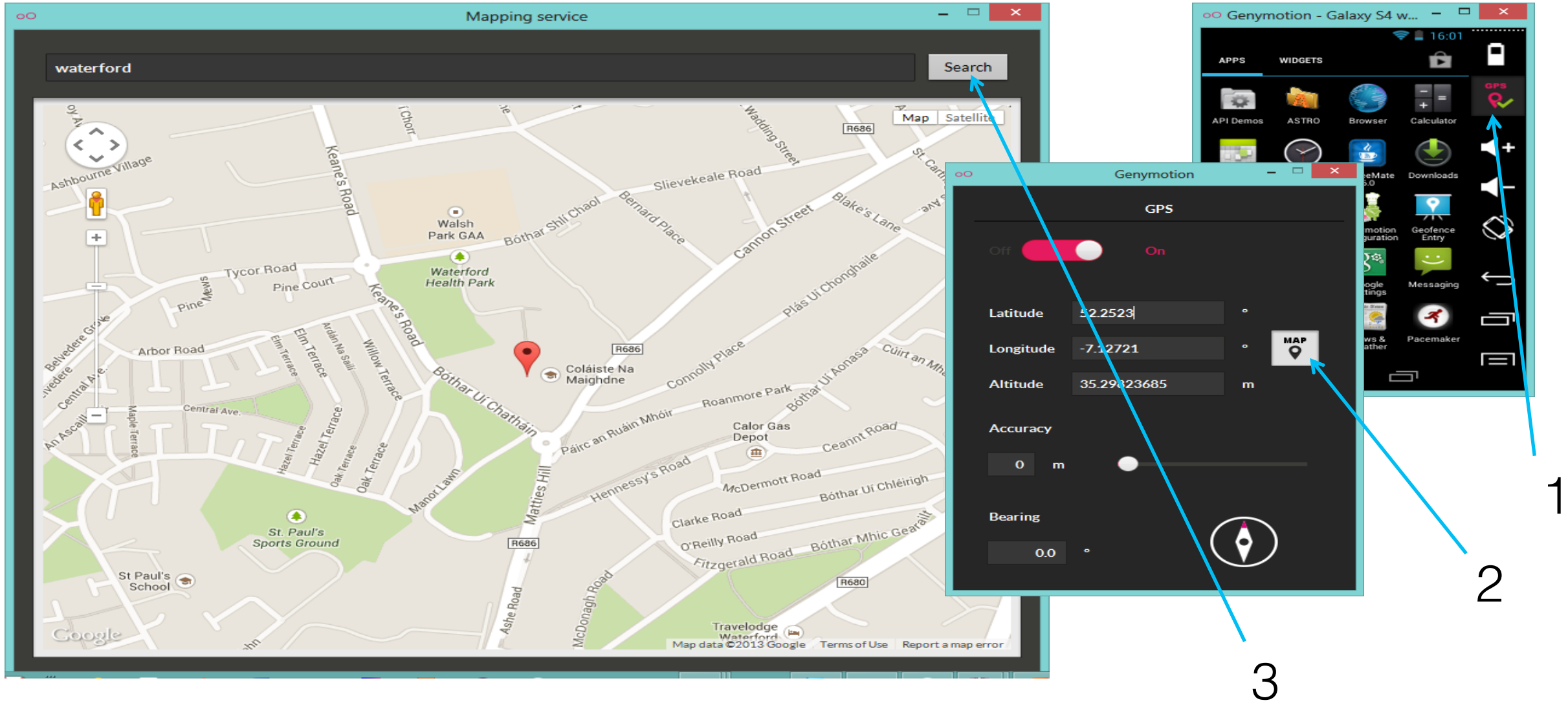
To test an application using the Google Play services SDK, you must use either

- A compatible Android device that runs Android 2.3 or higher and includes Google Play Store
- An Android emulator (virtual device) that runs the Google APIs platform based on Android 4.2.2 or higher (Genymotion is a good one to use and Android Studio has improved quite a lot in the last few releases – next few slides)

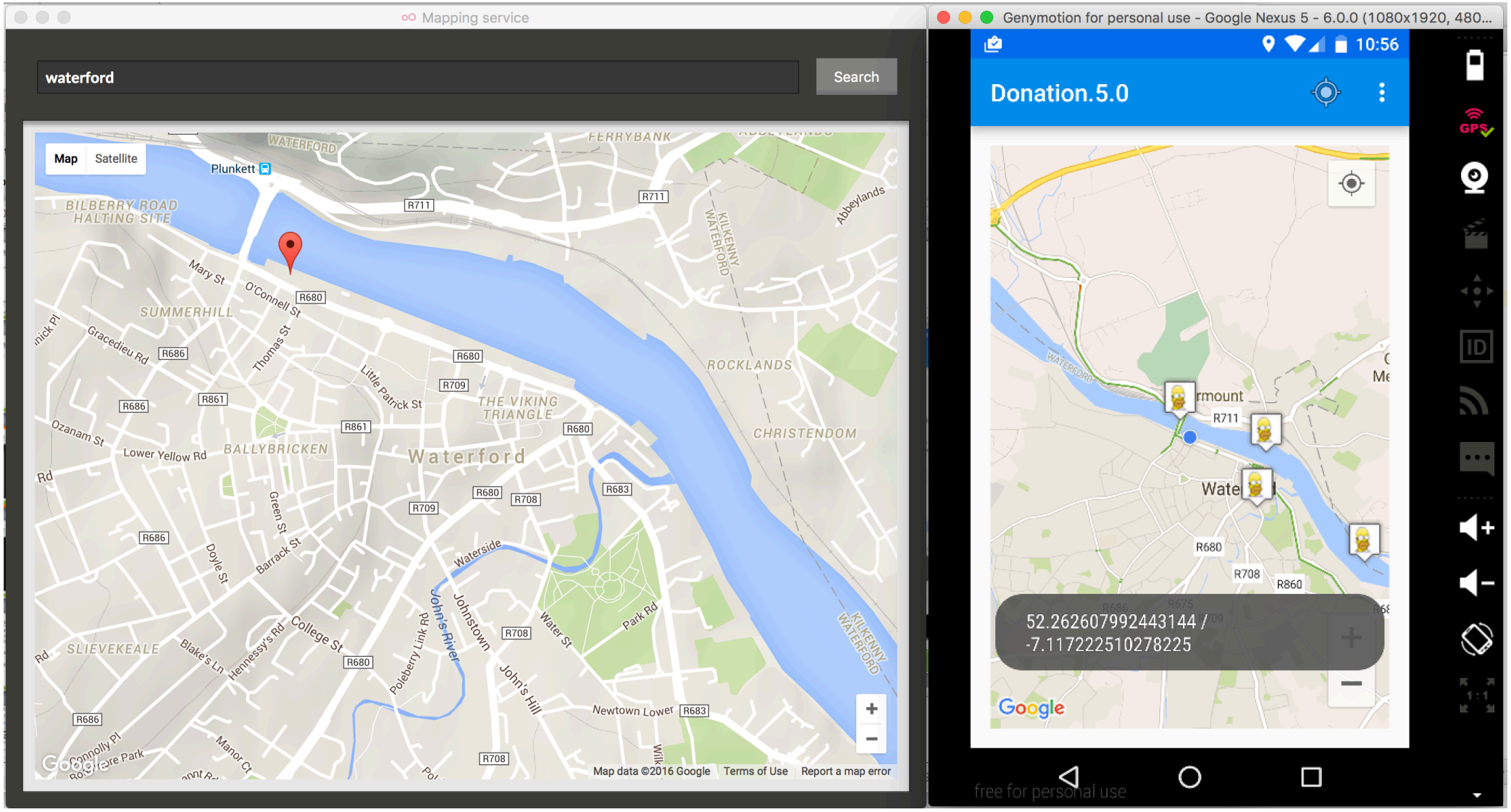
Aside : Android Studio Emulator Setup



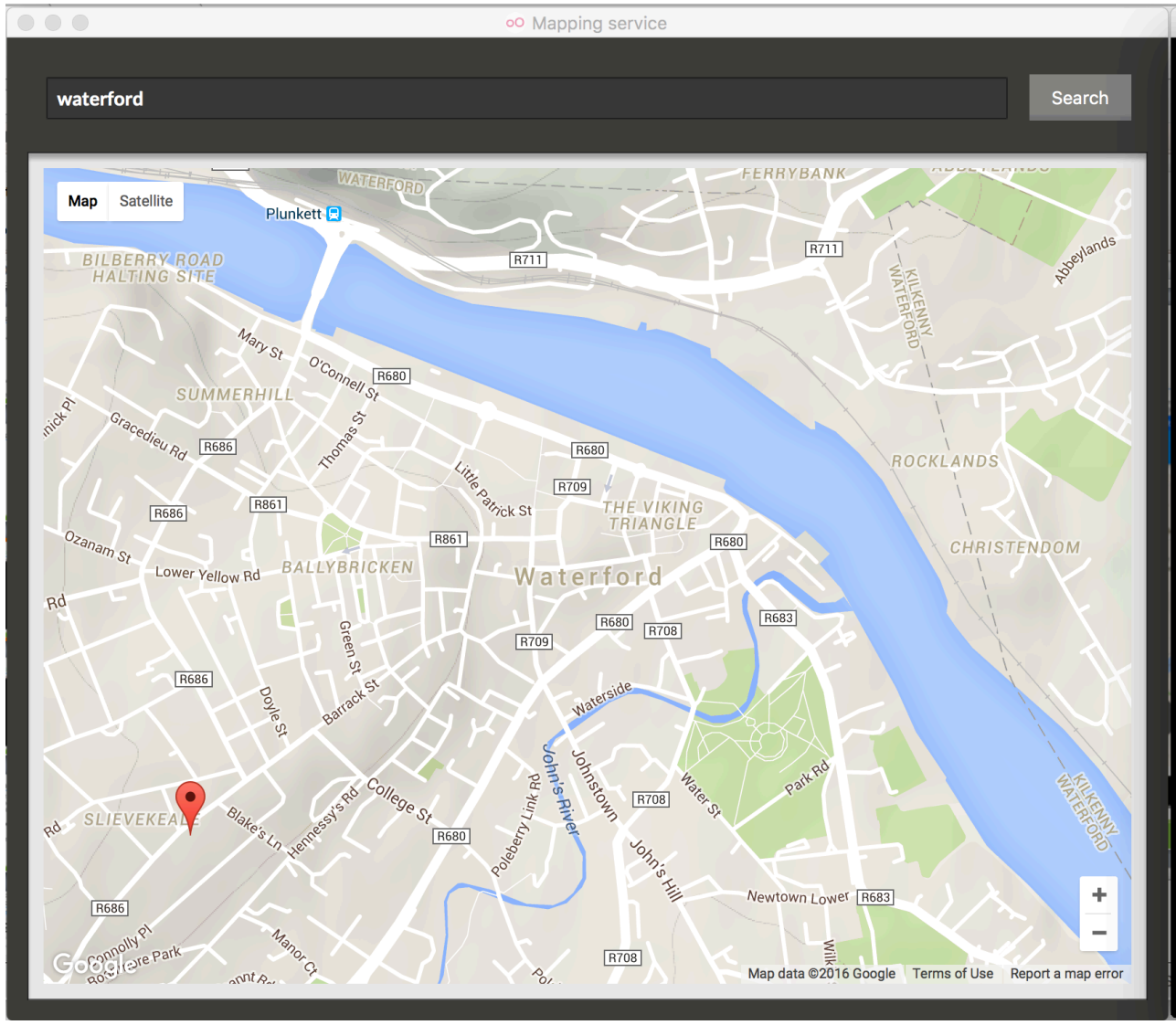
Aside : Genymotion Emulator Setup

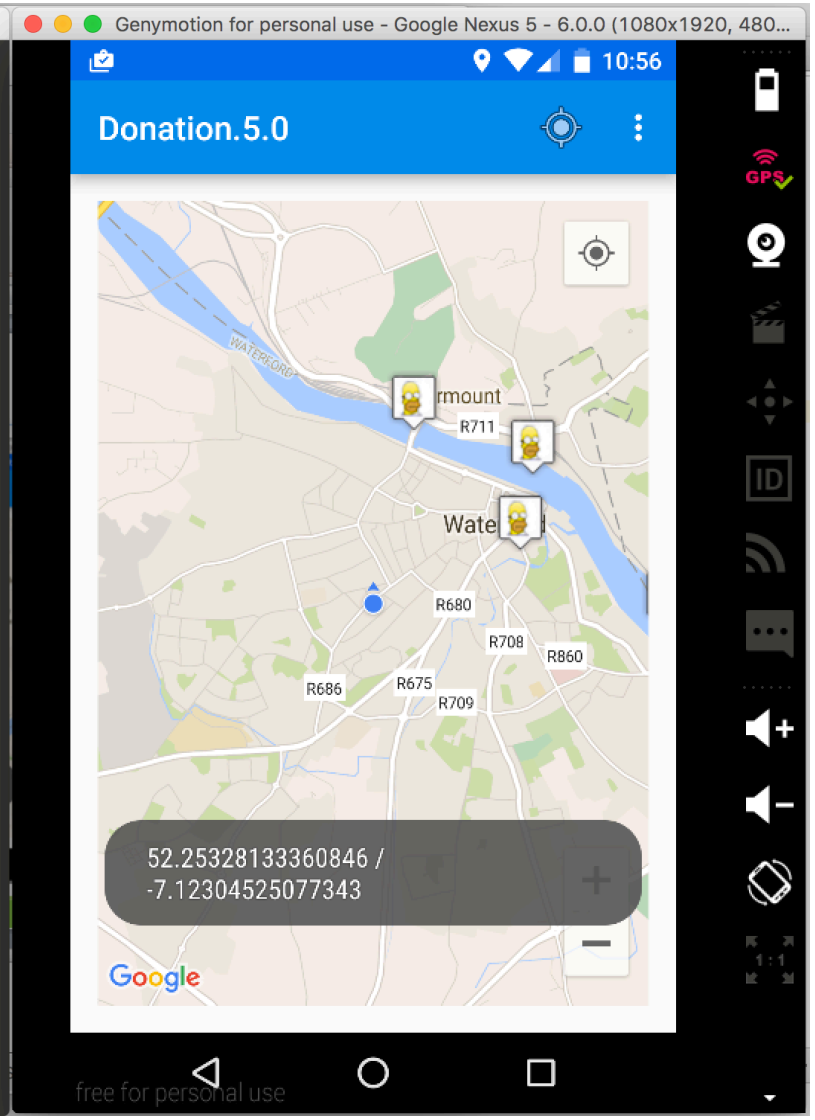


Example: Using `LocationListener` *



Example: Using `LocationListener` (continued)





Key Location Classes and Interfaces

In package `android.location`

- Class `Location`
  - represents a geographic location sensed at a particular time
- Class `Address`
  - represents an address as a set of strings describing a location.
- Class `Geocoder`
  - translates between locations and addresses

Key Location Classes and Interfaces (continued)

In package `com.google.android.gms.location`

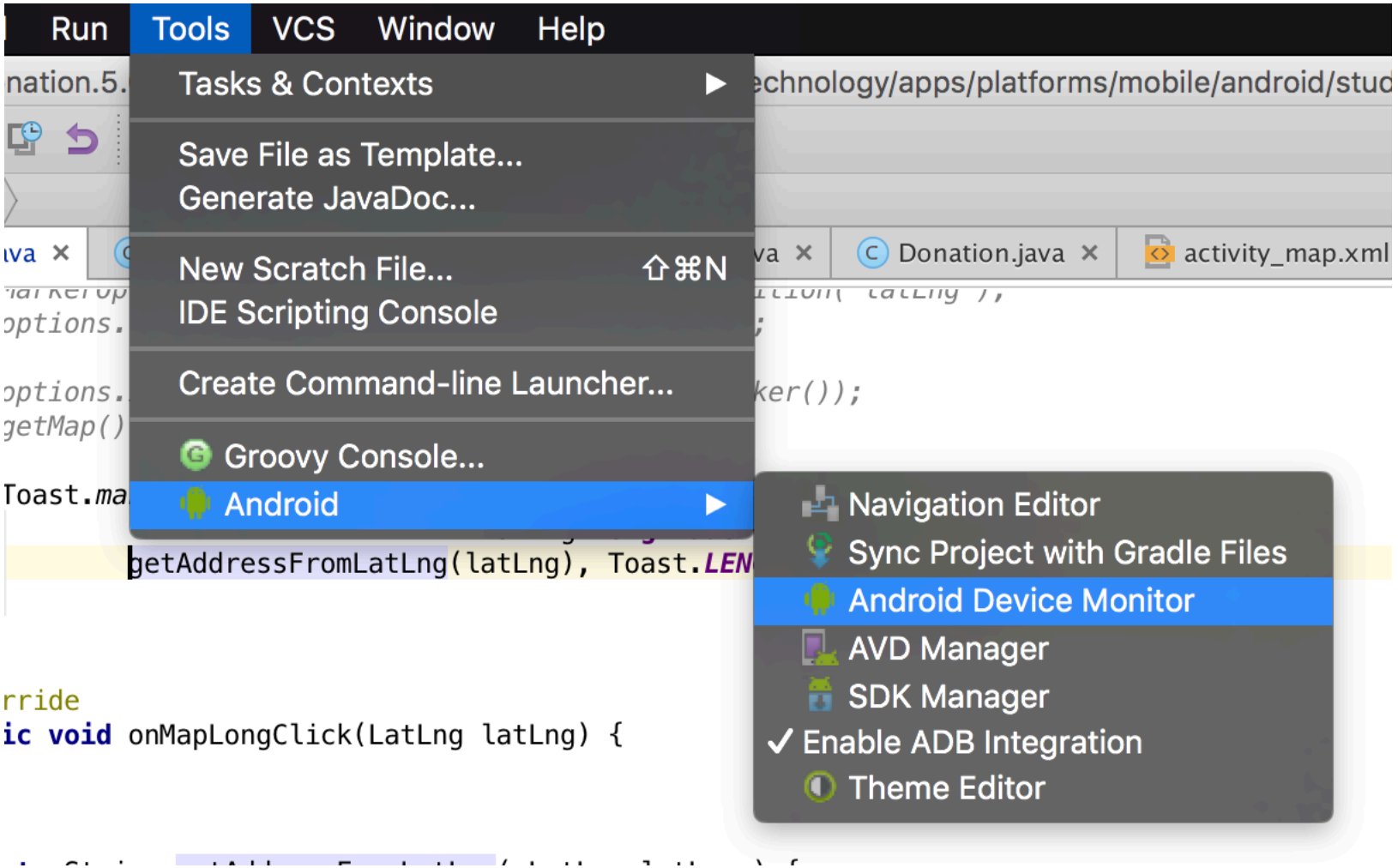
- Class `LocationServices`
  - main entry point for location services integration
- Interface `FusedLocationProviderApi`
  - main entry point for interacting with the fused location provider
- Interface `LocationListener`
  - receives notifications when the location has changed
- Class `LocationRequest`
  - contains quality-of-service parameters for requests to the `FusedLocationProviderApi`

Location Services on an Emulator

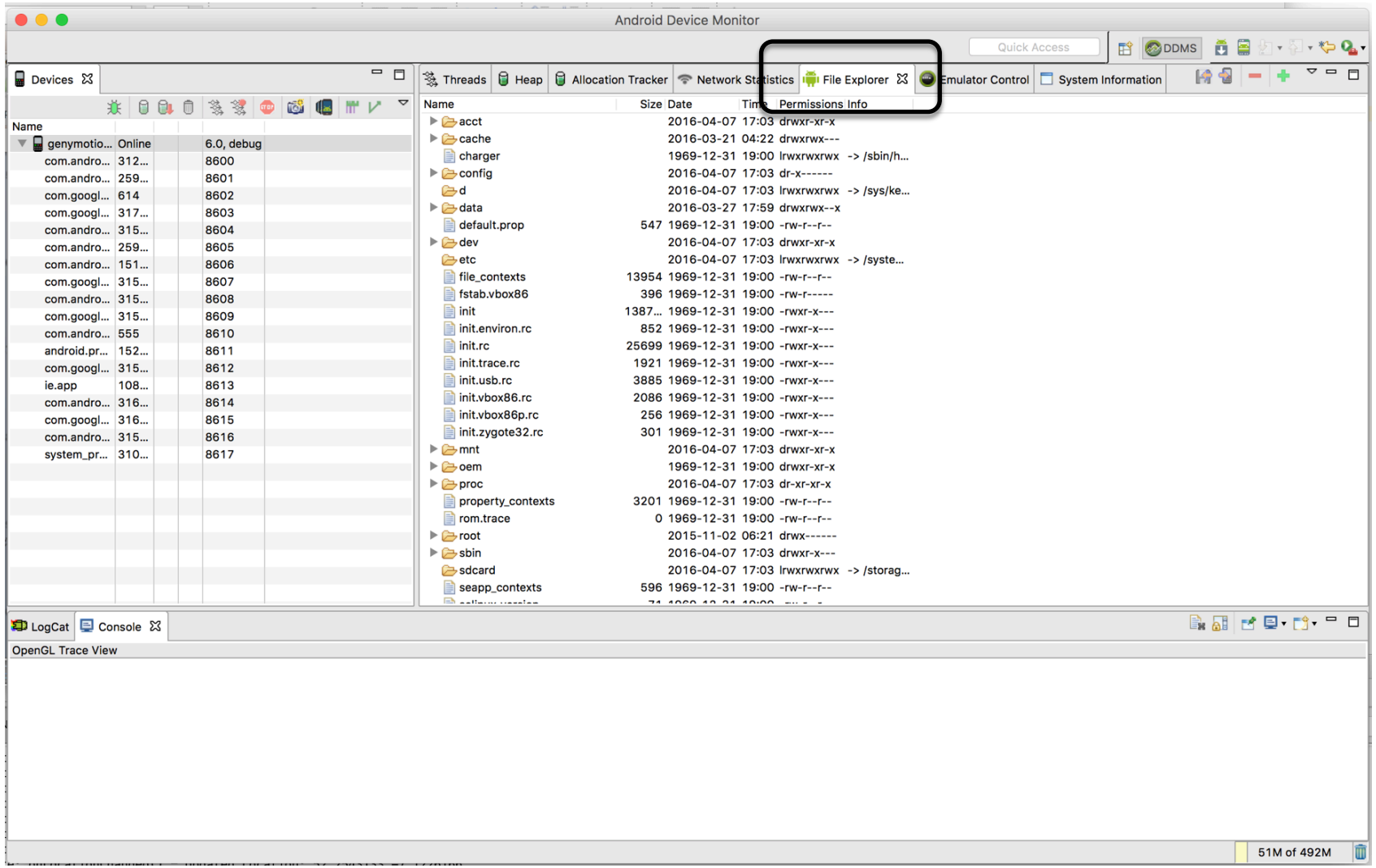
- A virtual device (emulator) does not have GPS or real location providers, so it uses a “mock” GPS provider that always returns the same position unless it is changed manually. (Like we can using Genymotion)
- If you’re not using Genymotion/Android Studio, the location on the emulator can be changed using
  - the Android Device Monitor
  - the “geo” command in the emulator console; e.g.,

```
geo fix -79.960138 32.797917
```

Using the Android Device Monitor

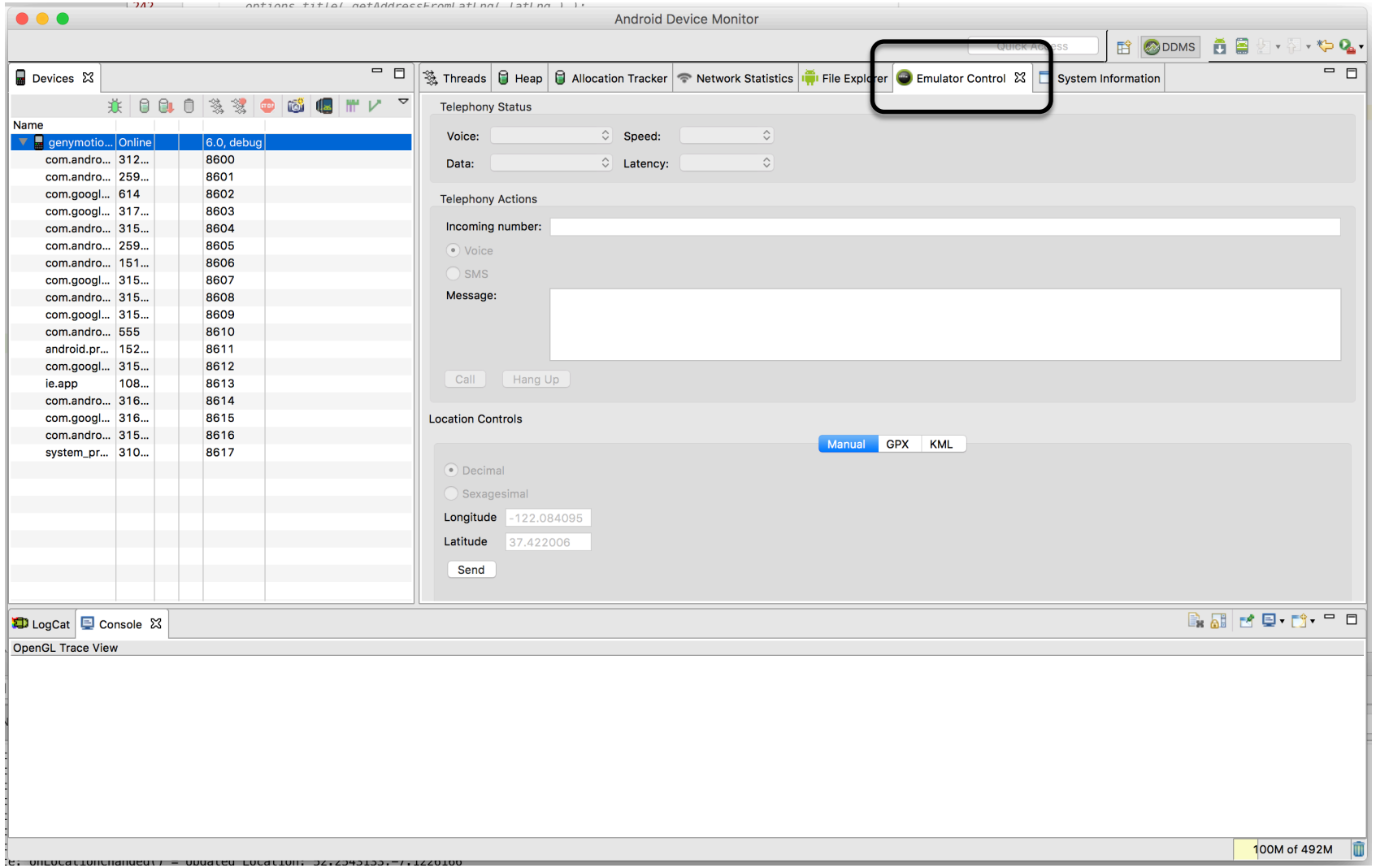


Using the Android Device Monitor



Emulator Control Panel

Using the Android Device Monitor

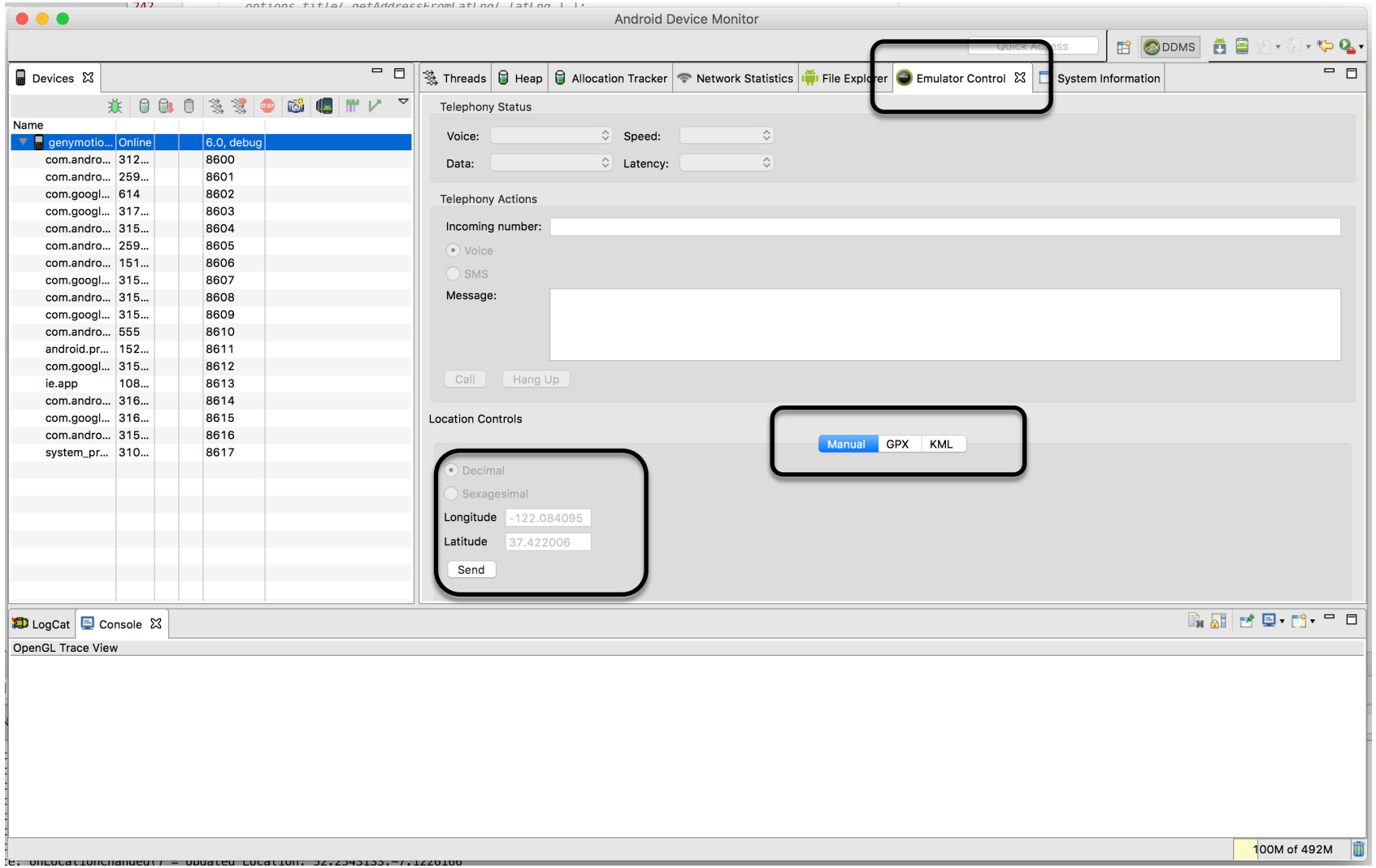


Emulator Control Panel

Using the Emulator Control Panel

- The Emulator Control panel can send simulated location data in three different ways:
  - Manually send individual longitude/latitude coordinates to the device.
  - Use a GPX file describing a route for playback to the device.
  - Use a KML file describing individual place marks for sequenced playback to the device.
- See the following for details of GPX and KML files:
  - GPX: The GPS Exchange Format
    http://www.topografix.com/gpx.asp
  - KML Tutorial
    http://code.google.com/apis/kml/documentation/kml_tut.html

Setting a Mock Location on an Emulator

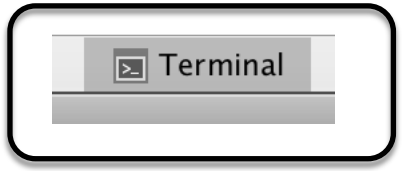


Emulator Control Panel

Setting a Mock Location Using the “geo” Command

To send mock location data from the command line:

- In Android Studio, click on the “Terminal” tab near the bottom.



- Connect to the emulator console:

```
telnet localhost 5554
```

5554 is the console port (check emulator screen)

- Send the location data:

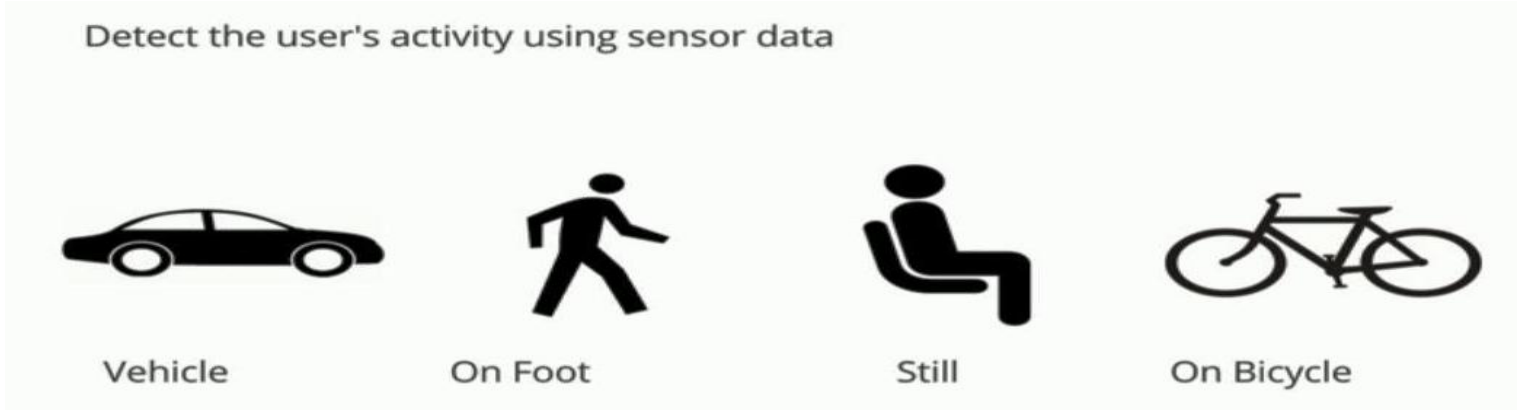
```
geo fix -121.45356 46.51119 4392
```

The “geo fix” command accepts a longitude and latitude in decimal degrees, and an optional altitude in meters.

Note that a telnet client is not installed automatically in Windows. Use Control Panel → Programs and Features → Turn Windows features on or off

Activity Recognition

- Makes it easy to check the user's current activity
  - still, walking, cycling, and in-vehicle, with very efficient use of the battery.



- Sensor data to find the type of action the user is performing

All Available via Google Play Services

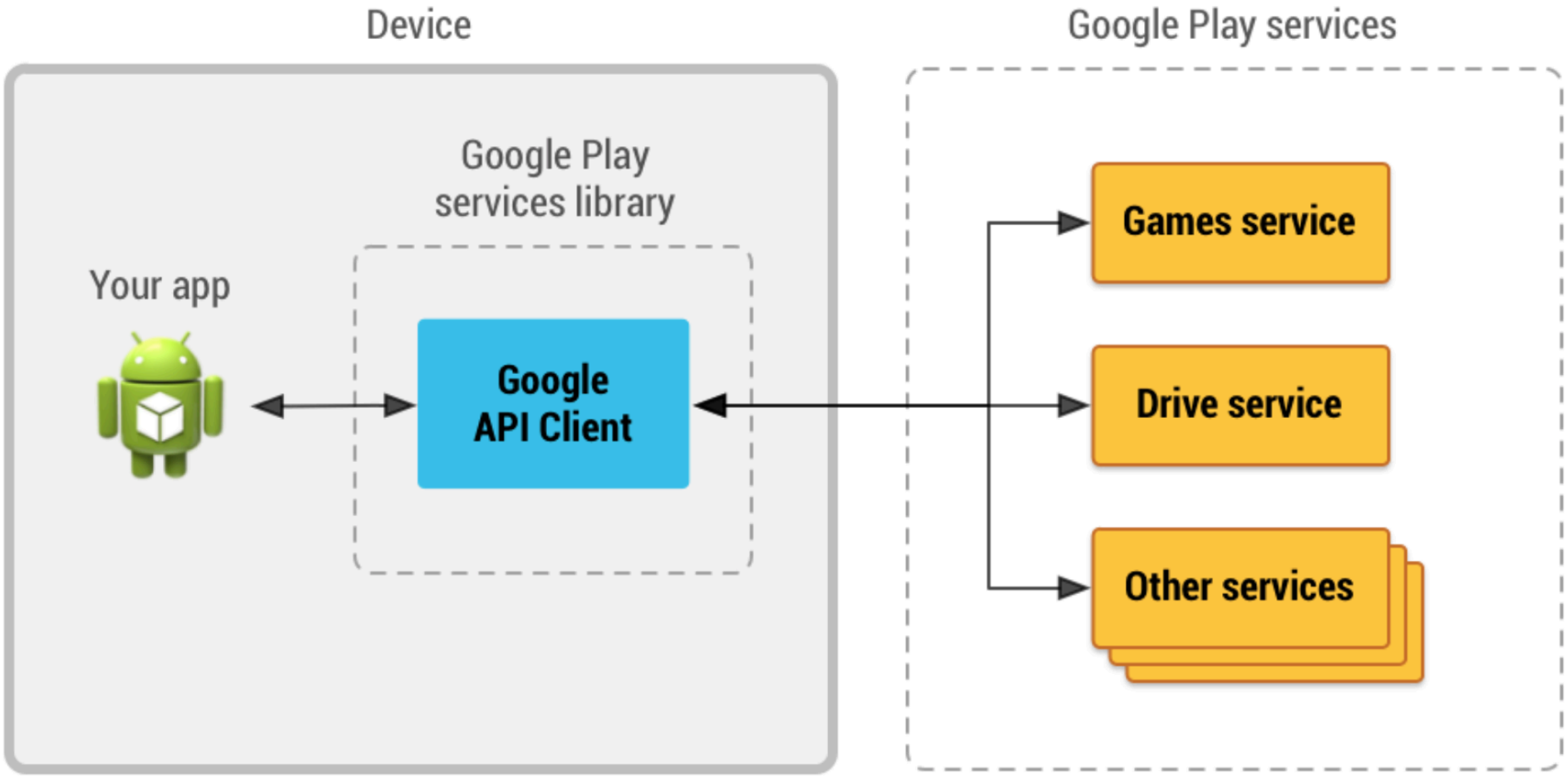


Figure 1: An illustration showing how the Google API Client provides an interface for connecting and making calls to any of the available Google Play services such as Google Play Games and Google Drive.

CoffeeMate 6.0

Code Highlights

MapsFragment – interfaces/instance variables *

```
public class MapsFragment extends MapFragment implements
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener,
        GoogleMap.OnInfoWindowClickListener,
        GoogleMap.OnMapLongClickListener,
        GoogleMap.OnMapClickListener,
        GoogleMap.OnMarkerClickListener,
        OnMapReadyCallback,
        LocationListener,
        VolleyListener {

    private GoogleApiClient mGoogleApiClient;
    private Location mCurrentLocation;
    private LocationRequest mLocationRequest;
    private List<Coffee> mCoffeeList;
    private long UPDATE_INTERVAL = 30000; /* 30 secs */
    private long FASTEST_INTERVAL = 1000; /* 5 secs */
    private GoogleMap mMap;
    private float zoom = 13f;
```

- Here we declare the interfaces our custom `MapFragment` (MapsFragment) implements.
- Interfaces for `Volley` & Location Updates.
- Variables to keep track of the users current location, location requests etc.

MapsFragment – GoogleApiClient Setup *

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    setHasOptionsMenu(true);

    TextView titleBar = (TextView) getActivity().findViewById(R.id.recentAc
    titleBar.setText("Coffee Map");

    Home.app.mGoogleApiClient = new GoogleApiClient.Builder(getActivity())
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .addApi(LocationServices.API)
            .build();
}
```

- Here we build our `GoogleApiClient` specifying the LocationServices API.
- It's common practice to 'rebuild' your api client (can actually improve performance)

MapsFragment – onStart() / onStop() *

```
@Override
public void onStart() {
    super.onStart();
    Home.app.mGoogleApiClient.connect();
    CoffeeApi.attachListener(this);
}

@Override
public void onStop() {
    super.onStop();
    if (Home.app.mGoogleApiClient != null && Home.app.mGoogleApiClient.isConnected())
        Home.app.mGoogleApiClient.disconnect();

    CoffeeApi.detachListener();
}
```

- Here we try and 'connect' to our `GoogleApiClient` and 'attach' the Fragment.
- Disconnect when the Fragment is stopped.

And detach the listener

MapsFragment – onConnected() *

```
@Override
public void onConnected(Bundle dataBundle) {

    getMapAsync(this);
    // Display the connection status
    try {
        mCurrentLocation = LocationServices
                .FusedLocationApi
                .getLastLocation(Home.app.mGoogleApiClient);
    }
    catch(SecurityException se)
    {
        Toast.makeText(getActivity(),"Check Your Permissions",Toast.LENGTH_SHORT).show();
    }

    if (mCurrentLocation != null) {
        Toast.makeText(getActivity(), "GPS location was found!", Toast.LENGTH_SHORT).show();
        //LatLng latLng = new LatLng(mCurrentLocation.getLatitude(), mCurrentLocation.getLongitude());

    } else {
        Toast.makeText(getActivity(), "Current location was null, Setting Default Values!", Toast.LENGTH_SHORT).show();
        mCurrentLocation = new Location("Waterford City Default (WIT)");
        mCurrentLocation.setLatitude(52.2462);
        mCurrentLocation.setLongitude(-7.1202);

    }
}
```

- Acquire `GoogleMap`
  (automatically initializes the maps system and the view)
- Get Current Location
- Set Location if necessary
  (e.g. on emulator)

MapsFragment – onMapReady() *

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;

    mMap.setMapType(MAP_TYPES[curMapTypeIndex]);
    if (!checkPermission())
        requestPermission();
    else
        mMap.setMyLocationEnabled(true);

    mMap.getUiSettings().setMapToolbarEnabled(true);
    mMap.getUiSettings().setCompassEnabled(true);
    mMap.getUiSettings().setMyLocationButtonEnabled(true);
    mMap.getUiSettings().setAllGesturesEnabled(true);
    mMap.setTrafficEnabled(true);
    mMap.setBuildingsEnabled(true);
    mMap.getUiSettings().setZoomControlsEnabled(true);

    initListeners();
    initCamera(mCurrentLocation);
    startLocationUpdates();
    CoffeeApi.attachListener(this);
    CoffeeApi.getAll("/coffees/" + Home.app.googleToken, null);
}
```

- Bind to our `GoogleMap` and set its initial type.
- Check for the necessary permissions.
- Set the Map (*mMap*) properties
- Initialise Listeners & Camera
- Start Location Updates (next slide) & get all the users coffees.

MapsFragment – Permissions *

```
//http://www.journaldev.com/10409/android-handling-runtime-permissions-example
private boolean checkPermission() {
    int result = ContextCompat.checkSelfPermission(getActivity(), ACCESS_FINE_LOCATION);
    int result1 = ContextCompat.checkSelfPermission(getActivity(), CAMERA);

    return result == PackageManager.PERMISSION_GRANTED && result1 == PackageManager.PERMISSION_GRANTED;
}

private void requestPermission() {
    ActivityCompat.requestPermissions(getActivity(), new String[]{ACCESS_FINE_LOCATION, CAMERA},
                                                                    PERMISSION_REQUEST_CODE);
}
```

- Checking to see if Location & Camera permissions are allowed
- Requesting Location & Camera permissions

MapsFragment – Permissions *

```
@Override
public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
    switch (requestCode) {
        case PERMISSION_REQUEST_CODE:
            if (grantResults.length > 0) {

                boolean locationAccepted = grantResults[0] == PackageManager.PERMISSION_GRANTED;
                boolean cameraAccepted = grantResults[1] == PackageManager.PERMISSION_GRANTED;

                if (locationAccepted && cameraAccepted)
                    Snackbar.make(getView(), "Permission Granted, Now you can access location data and camera.",
                                                                        Snackbar.LENGTH_LONG).show();
                else {

                    Snackbar.make(getView(), "Permission Denied, You cannot access location data and camera.",
                                                                        Snackbar.LENGTH_LONG).show();

                    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                        if (shouldShowRequestPermissionRationale(ACCESS_FINE_LOCATION)) {
                            showMessageOKCancel("You need to allow access to both the permissions",
                                    new DialogInterface.OnClickListener() {
                                        @Override
                                        public void onClick(DialogInterface dialog, int which) {
                                            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                                                requestPermissions(new String[]{ACCESS_FINE_LOCATION, CAMERA},
                                                        PERMISSION_REQUEST_CODE);
                                            }
                                        }
                                    });
                            return;
                        }
                    }
                }
            }
            break;
    }
}
```

- Retrieving permission status
- Updating the User

MapsFragment – Tracking Location (1) *

```
protected void startLocationUpdates() {
    mLocationRequest = new LocationRequest();
    mLocationRequest.setPriority(LocationRequest.PRIORITY_BALANCED_POWER_ACCURACY);
    mLocationRequest.setInterval(UPDATE_INTERVAL);
    mLocationRequest.setFastestInterval(FASTEST_INTERVAL);
    try {
        LocationServices.FusedLocationApi.requestLocationUpdates(Home.app.mGoogleApiClient,
                mLocationRequest, this);
    }
    catch(SecurityException se)
    {
        Toast.makeText(getActivity(),"Check Your Permissions on Location Updates",Toast.LENGTH_SHORT).show();
    }
}
```

- Create a new `LocationRequest` and set values
- Use the `FusedLocationApi` to *requestLocationUpdats*

MapsFragment – Tracking Location (2) *

```
public void onLocationChanged(Location location) {
    // Report to the UI that the location was updated
    String msg = "Updated Location: " +
            Double.toString(location.getLatitude()) + "," +
            Double.toString(location.getLongitude());
    //Toast.makeText(getActivity(), msg, Toast.LENGTH_SHORT).show();
    Log.v("coffeemate", "onLocationChanged() = " + msg);
    mCurrentLocation = location;
    initCamera(mCurrentLocation);
}
```

- Get individual latitude and longitude whenever there's a location change
- Update our current location (`mCurrentLocation`) and initialise/reposition the camera

MapsFragment – Helper Methods *

```
private void initListeners() {
    mMap.setOnMarkerClickListener(this);
    mMap.setOnMapLongClickListener(this);
    mMap.setOnInfoWindowClickListener(this);
    mMap.setOnMapClickListener(this);
}
```

```
private void initCamera(Location location) {

    if (zoom != 13f && zoom != mMap.getCameraPosition().zoom)
            zoom = mMap.getCameraPosition().zoom;

    CameraPosition position = CameraPosition.builder()
            .target(new LatLng(location.getLatitude(),
                    location.getLongitude()))
            .zoom(zoom)
            .bearing(0.0f)
            .tilt(0.0f)
            .build();

    mMap.animateCamera(CameraUpdateFactory
            .newCameraPosition(position), null);
}
```

- Adding necessary listeners to our `GoogleMap` reference.
- Position/reposition the Camera based on current location and set zoom ratio.

MapsFragment – Adding Coffee Markers *

```
@Override
public void setList(List list) {
    Home.app.coffeeList = list;
    addCoffees(Home.app.coffeeList);
}
```

- Triggered by our `CoffeeApi` callback
- Traversing our list of coffees and adding a location marker to the map

```
public void addCoffees(List<Coffee> list)
{
    for(Coffee c : list)
        mMap.addMarker(new MarkerOptions()
                .position(new LatLng(c.marker.coords.latitude, c.marker.coords.longitude))
                .title(c.name + " €" + c.price)
                .snippet(c.shop + " " + c.address)
                .icon(BitmapDescriptorFactory.fromResource(R.drawable.coffee_icon)));

}
```

Relevant Links

- Location APIs
  https://developer.android.com/google/play-services/location.html
- Setting Up Google Play Services
  https://developer.android.com/google/play-services/setup.html
- Getting the Last Known Location
  http://developer.android.com/training/location/retrieve-current.html
- Receiving Location Updates
  http://developer.android.com/training/location/receive-location-updates.html
- Displaying a Location Address
  http://developer.android.com/training/location/display-address.html

Questions?