COMP-304 Winter 2023

Review of Lecture 9

- **Firebase Realtime Database**
  - Cloud-hosted NoSQL database**.**
  - **This database is stored locally** on each device **as JSON files**, which are synchronized in real time to the cloud-host.
  - You need to register your app with Firebase, add Firebase configuration file, and modify project and module Gradle files
  - Use **setValue** to insert
  - Use **getValue** to read
  - Use **removeValue** to delete an entry
- **Option 1**: Add Firebase using the Firebase console (https://firebase.google.com/docs/android/setup)
- **Option 2**: Android Studio includes a **Firebase Assistant** to simplify adding Firebase components to your app
- The Firebase Realtime Database uses **a declarative rules language** to define how your data should be accessed.

Review of Lecture 9

- By default, Firebase Databases require **Firebase Authentication**, and grant full read and write permissions to all authenticated users.
  - To set the **access rules to public**, switch to the rules tab and set the read and write elements to true.
- **Firestore** is a highly-scalable NoSQL cloud database that, like Firebase Realtime Database, can be **used to sync application data across servers and client apps in real time.**
- Firestore is designed specifically to be highly scalable and supports more expressive and efficient querying, including shallow queries that don't require retrieving the entire collection, and support for sorting, filtering, and limiting query returns.

Location-Based Services

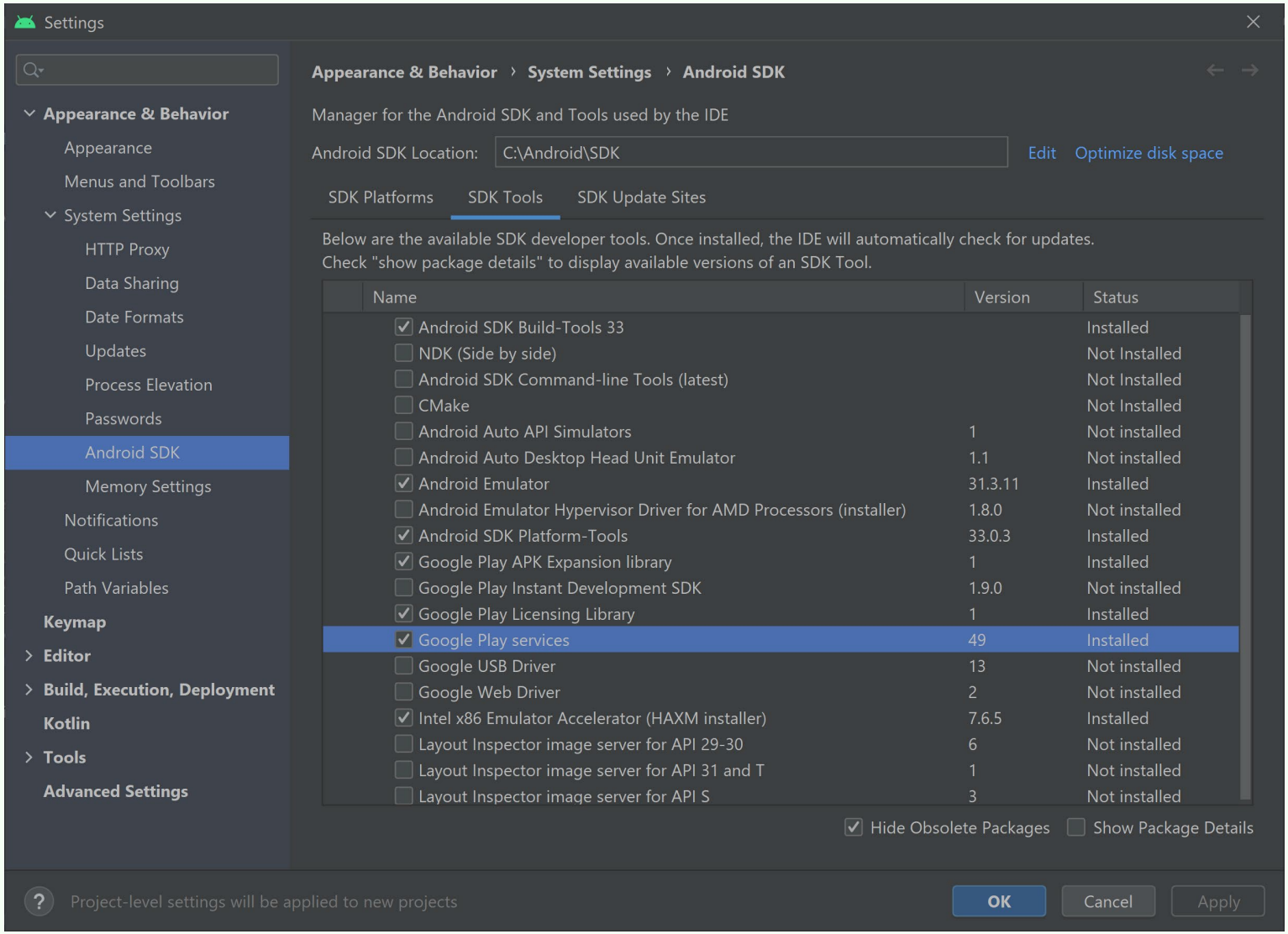
Objectives:

- Install and use **Google Play services**.
- Determine and update the **device's physical location** using the emulator.
- Add **interactive maps** to your application.
- Display **user location on a map**.
- Find addresses and address locations with the **Geocoder**.
- Set and monitor **Geofences**.

Location Services

- **Location Services** enable you to **find the device's current location**, and **get updates** as it changes.
- **Google Play Services** are a **set of libraries** that you can include in your projects to access over 20 Google-proprietary features including **Location Services**, **Google Maps**, and the **Awareness APIs**.
- **Maps and location-based services** use **latitude** and **longitude** to pinpoint geographic locations.
  - A **geocoder** that you can use to convert back and forth between **latitude/longitude** values **and real-world addresses**.
- The **Awareness API** that helps you understand and react to changes in your **user's context**.

Google Play Services



Adding Google Play services as app dependencies

```
dependencies {
...
implementation 'com.google.android.gms:play-services-awareness:19.0.1"
implementation 'com.google.android.gms:play-services-maps:18.1.0'
implementation 'com.google.android.gms:play-services-location:20.0.0'
}
```

Checking if Google Play services is available

```
val googleApiAvailability = GoogleApiAvailability.getInstance()
    val status = googleApiAvailability.isGooglePlayServicesAvailable(unwrappedActivity)
    if (status != ConnectionResult.SUCCESS) {
        if (googleApiAvailability.isUserResolvableError(status) && displayError) {
        googleApiAvailability.getErrorDialog(unwrappedActivity, status, 2404)!!.show()
```

Using the Location Service

- Using the Location Service, you can do the following:
  - Obtain your current **location**
  - Follow **movement**
  - Set **geofences** for detecting movement into and out of a specified area
- Obtaining the current device location and specifying the **degree of location accuracy,** requires one of two **uses-permission** tags in your manifest:

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

Receive location updates in Android - Example

- This example shows **how to receive location updates**.
- The key components include the following:
  - **MainActivity -** for the user to allow the app to access the device's location
  - **ForegroundLocationService -** service that subscribes and unsubscribes to location changes, and promotes itself to a foreground service (with a notification) if the user navigates away from the app's activity. You add location code here.
  - **Util -** Adds extension functions for the Location class and saves location in **SharedPreferences** (simplified data layer).

https://codelabs.developers.google.com/codelabs/while-in-use-location/#1

Receive location updates in Android - Example

- There are four different options for location access:
  - **Allow only while using the app**
    - This option is the recommended option for most apps. Also known as "while-in-use" or "foreground only" access, this option was added in Android 10 and allows developers to retrieve location only while the app is actively being used. An app is considered to be active if either of the following is true:
  - **An activity is visible**.
    - A foreground service is running with an ongoing notification.
  - **One time only**
    - Added in Android 11, this is the same as Allow only while using the app, but for a limited amount of time. For more information, see One-time permissions.
  - **Deny**
    - This option prevents access to location information.
  - **Allow all the time** (requires an extra permission for Android 10 and higher)

https://codelabs.developers.google.com/codelabs/while-in-use-location/#1

Receive location updates in Android - Example

- To fully support **Allow only while** using the app location updates, you need to account for when the user navigates away from your app.
- If you wish to continue receiving updates in that situation, you need to **create a foreground Service** and associate it with a Notification.
- In addition, if you want to use the same Service to request location updates when your app is visible and when the user navigates away from your app, you need to bind/unbind that Service to the UI element.

Receive location updates in Android - Example

- Your app can access the set of supported location services through classes in the **com.google.android.gms.location** package.
- **FusedLocationProviderClient -** the central component of the location framework used to request location updates and get the last known location.
- **LocationRequest -** a data object that contains quality-of-service parameters for requests (intervals for updates, priorities, and accuracy).
  - This is passed to the **FusedLocationProviderClient** when you request location updates.
- **LocationCallback** - is used for receiving notifications when the device location has changed or can no longer be determined.
  - This is passed a **LocationResult** where you can get the **Location** to save in your database.

Review the key variables needed for location updates

```
// TODO: Step 1.1, Review variables (no changes).
// FusedLocationProviderClient - Main class for receiving location updates.
private lateinit var fusedLocationProviderClient: FusedLocationProviderClient
// LocationRequest - Requirements for the location updates, i.e., how often you
// should receive updates, the priority, etc.
private lateinit var locationRequest: LocationRequest
// LocationCallback - Called when FusedLocationProviderClient has a new Location.
private lateinit var locationCallback: LocationCallback
// Used only for local storage of the last known location. Usually, this would be saved to your database, but because this is a simplified sample without a full database, we only need the last location to create a Notification if the user navigates away from the app.
private var currentLocation: Location? = null
```

Review the FusedLocationProviderClient initialization

- In the base module, search for TODO: Step 1.2, Review the **FusedLocationProviderClient** in the **ForegroundOnlyLocationService.kt** file:

- // TODO: Step 1.2, Review the FusedLocationProviderClient.
- fusedLocationProviderClient = LocationServices.getFusedLocationProviderClient(this)
- As mentioned in the previous comments, this is the main class for getting location updates.
- The variable is already initialized for you, but it's important to review the code to understand how it is initialized.
  - You add some code here later to request location updates.

Initialize the LocationRequest

ForegroundOnlyLocationService.kt file:

```
// TODO: Step 1.3, Create a LocationRequest.
locationRequest = LocationRequest.create().apply {
    // Sets the desired interval for active location updates. This interval is inexact. You
    // may not receive updates at all if no location sources are available, or you may
    // receive them less frequently than requested. You may also receive updates more
    // frequently than requested if other applications are requesting location at a more
    // frequent interval.
    //
    // IMPORTANT NOTE: Apps running on Android 8.0 and higher devices (regardless of
    // targetSdkVersion) may receive updates less frequently than this interval when the app
    // is no longer in the foreground.
    interval = TimeUnit.SECONDS.toMillis(60)

    // Sets the fastest rate for active location updates. This interval is exact, and your
    // application will never receive updates more frequently than this value.
    fastestInterval = TimeUnit.SECONDS.toMillis(30)

    // Sets the maximum time when batched location updates are delivered. Updates may be
    // delivered sooner than this interval.
    maxWaitTime = TimeUnit.MINUTES.toMillis(2)

    priority = LocationRequest.PRIORITY_HIGH_ACCURACY
}
```

Initialize the LocationCallback

```
// TODO: Step 1.4, Initialize the LocationCallback.
locationCallback = object : LocationCallback() {
    override fun onLocationResult(locationResult: LocationResult) {
        super.onLocationResult(locationResult)
        // Normally, you want to save a new location to a database. We are simplifying
        // things a bit and just saving it as a local variable, as we only need it again
        // if a Notification is created (when the user navigates away from app).
        currentLocation = locationResult.lastLocation
        // Notify our Activity that a new location was added. Again, if this was a
        // production app, the Activity would be listening for changes to a database
        // with new locations, but we are simplifying things a bit to focus on just
        // learning the location side of things.
        val intent = Intent(ACTION_FOREGROUND_ONLY_LOCATION_BROADCAST)
        intent.putExtra(EXTRA_LOCATION, currentLocation)
        LocalBroadcastManager.getInstance(applicationContext).sendBroadcast(intent)
        // Updates notification content if this service is running as a foreground
        // service.
        if (serviceRunningInForeground) {
            notificationManager.notify(
                NOTIFICATION_ID,
                generateNotification(currentLocation))
        }
    }
}
```

Initialize the LocationCallback

- The **LocationCallback** you create here is the callback that the **FusedLocationProviderClient** will call when a new location update is available.
- In your callback, you first get the latest location using a **LocationResult** object.
- After that, you notify your Activity of the new location using a local broadcast (if it is active) or you update the Notification if this service is running as a foreground Service.

Subscribe to location changes

- In the **ForegroundOnlyLocationService.kt** file:
- // TODO: Step 1.5, Subscribe to location changes.

fusedLocationProviderClient.**requestLocationUpdates**(locationRequest, locationCallback, Looper.getMainLooper())

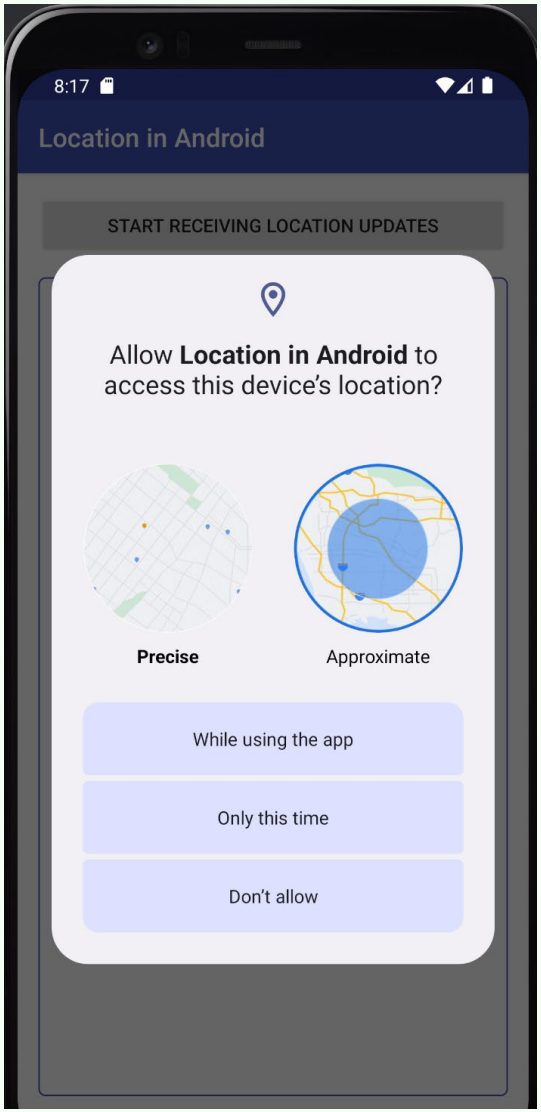
- The **requestLocationUpdates()** call lets the **FusedLocationProviderClient** know that you want to receive location updates.
- **LocationRequest** and **LocationCallback** let the FusedLocationProviderClient know the quality-of-service parameters for your request and what it should call when it has an update.
- Finally, the **Looper** object specifies the thread for the callback.
- This code is within a try/catch statement.
- This method requires such a block because a **SecurityException** occurs when your app doesn't have permission to access location information.

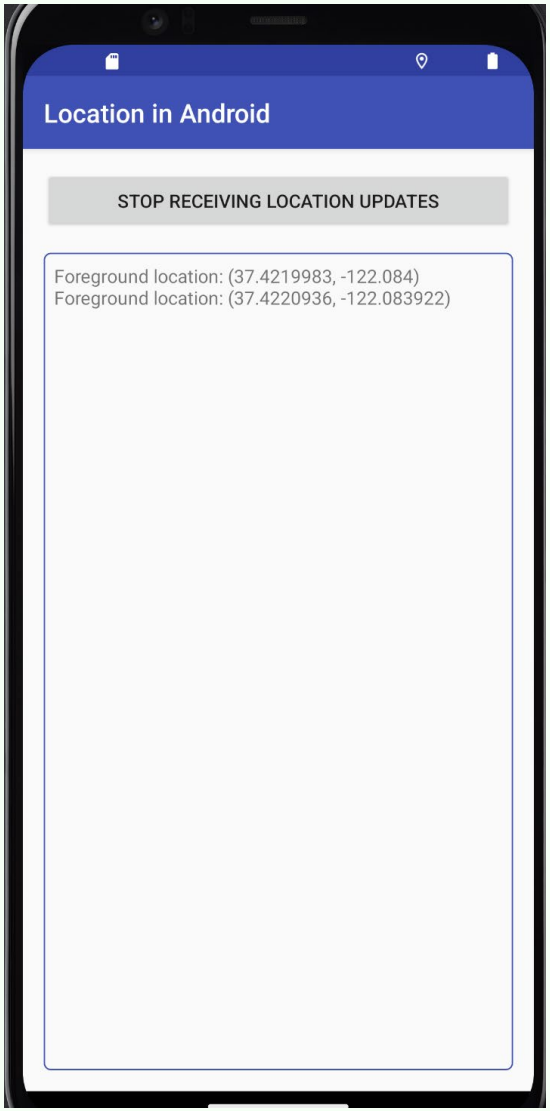
Unsubscribe from location changes

- When the app no longer needs access to location information, it's important to unsubscribe from location updates.
- In the **ForegroundOnlyLocationService.kt** file:

```
// TODO: Step 1.6, Unsubscribe to location changes.
//set up aan execute a task to let the FusedLocationProviderClient know that you no longer want to receive location updates for your LocationCallback
val removeTask = fusedLocationProviderClient.removeLocationUpdates(locationCallback)
removeTask.addOnCompleteListener { task ->
    if (task.isSuccessful) {
        Log.d(TAG, "Location Callback removed.")
        stopSelf()
    } else {
        Log.d(TAG, "Failed to remove Location Callback.")
    }
}
```

while-in-use-location app





Best Practices When Using Location

- **Battery life versus accuracy** - carefully consider how accurate your location updates need to be.
- **Minimize update rate** - slower updates can reduce battery drain at the price of less timely updates.
- **Modify the fastest interval -** useful when your application is performing a time-consuming operation that will prevent it from processing further location updates.
- **Unsubscribe when appropriate** - your app should always unsubscribe from updates whenever they aren't needed.

Using the Geocoder

- Geocoding enables you to **translate in both directions between street addresses and longitude/latitude** map coordinates.
- This can give you a recognizable context for the locations and coordinates used in location-based services and map-based Activities.
- The **Geocoder** class provides access to two geocoding functions:
  - **Forward geocoding** - Finds the latitude and longitude of an address
  - **Reverse geocoding** - Finds the street address for a given latitude and longitude

Using the Geocoder

- Geocoder geocoder = new Geocoder(this, Locale.getDefault());
- The Geocoder uses a web service to implement its lookups that may not be included on all Android devices.
- Use the isPresent method to determine if a Geocoder implementation exists on a given device:

```
boolean geocoderExists = Geocoder.isPresent();
```

- As the geocoding lookups are done on the server, your app also requires the Internet uses-permission in your manifest:
- <uses-permission android:name="android.permission.INTERNET"/>
- Google limits to the number and frequency of requests.
- The limits of the Google Maps-based service include:
  - A maximum of 2,500 requests per day per device
  - No more than 50 QPS (queries per second)

Reverse Geocoding

- **Reverse** geocoding returns street addresses for physical locations specified by latitude/longitude pairs.
- It's a useful way to get a recognizable context for the Locations returned by location-based services.

```
private void reverseGeocode(Location location) {
double latitude = location.getLatitude();
double longitude = location.getLongitude();
List<Address> addresses = null;
Geocoder gc = new Geocoder(this, Locale.getDefault());
try {
addresses = gc.getFromLocation(latitude, longitude, 10);
} catch (IOException e) { Log.e(TAG, "Geocoder I/O Exception", e);}
}
```

Forward Geocoding

- **Forward** geocoding (or just geocoding) determines map coordinates for a given location.

```
List<Address> result =
geocoder.getFromLocationName(streetAddress, 5);
```

- Geocoding an address:

```
Geocoder geocoder = new Geocoder(this, Locale.US);
String streetAddress = "160 Riverside Drive, New York, New
York";
List<Address> locations = null;
try {
locations = geocoder.getFromLocationName(streetAddress,
5);
} catch (IOException e) {
    Log.e(TAG, "Geocoder I/O Exception", e);
}
```

Creating Map-based Activities

- Using a GoogleMap from within a MapFragment, you can create Activities that include an interactive map.

```
dependencies {
...
implementation 'com.google.android.gms:play-services-maps:17.0.0'
}
```

- You can **embed a map** into your own applications and use it for different purposes
- A **SupportMapFragment** is the simplest way to place a map in an application.
- It's a wrapper around a view of a map to automatically handle the necessary life cycle needs.

Displaying Maps

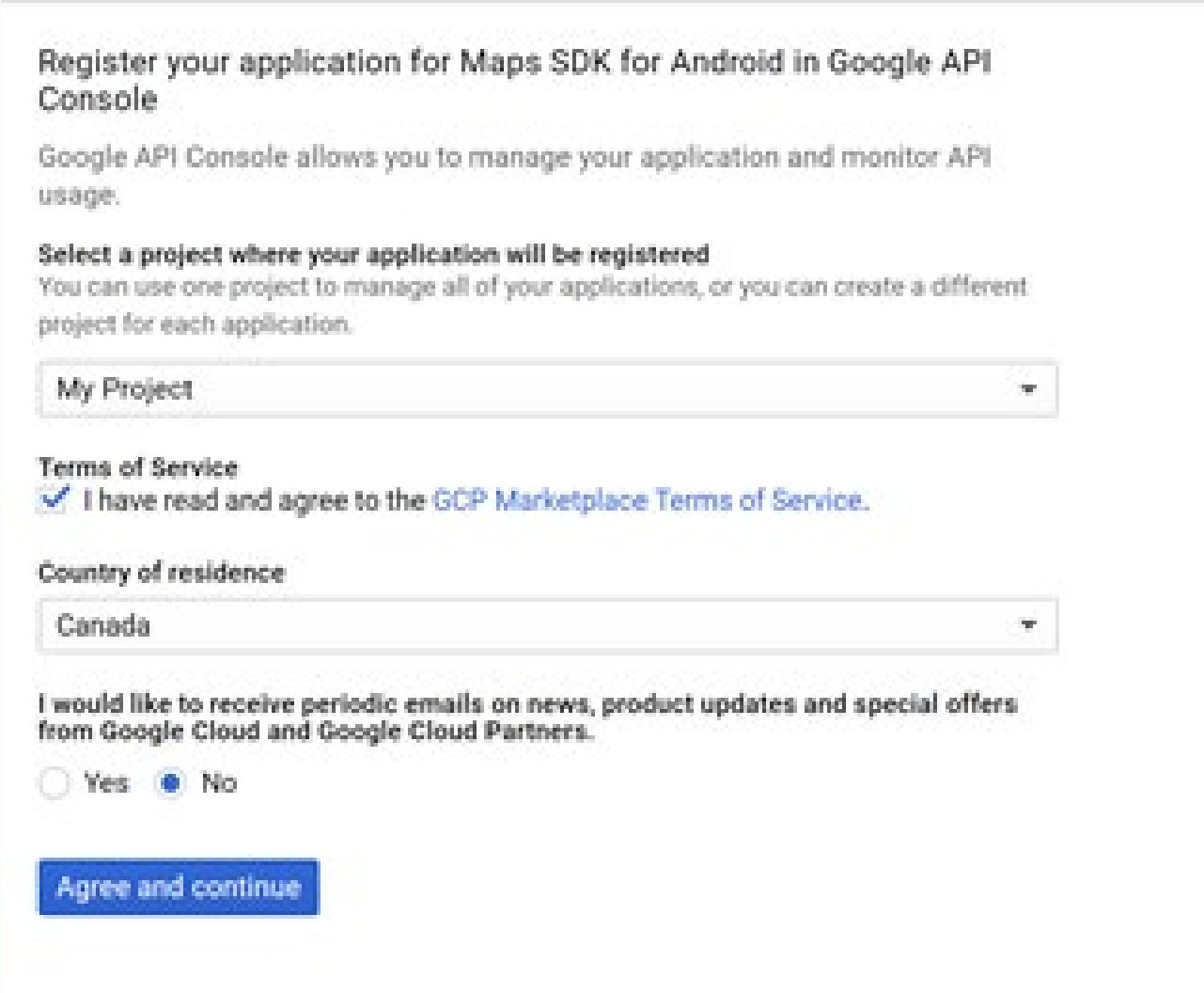
- The steps:
  1. Install and/or update the **Google Play services SDK**

  1. Create a **Google Maps project**
     - **It creates** google_maps_api.xml file
  2. Get a **Google Maps API key**

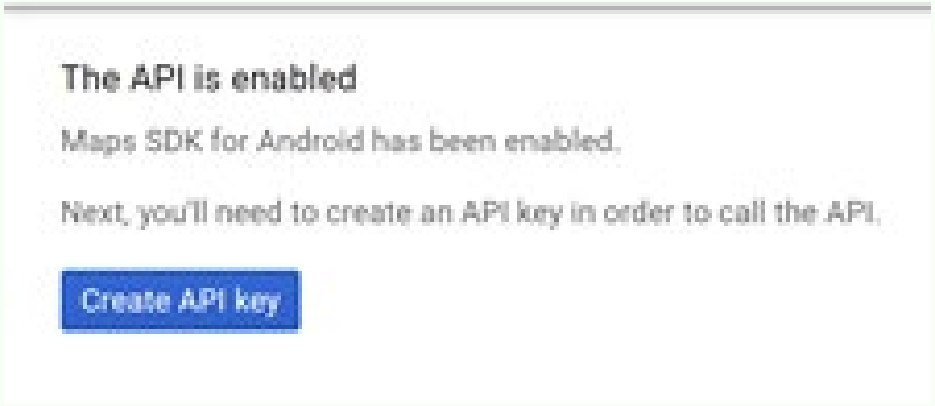
Getting a Maps API Key

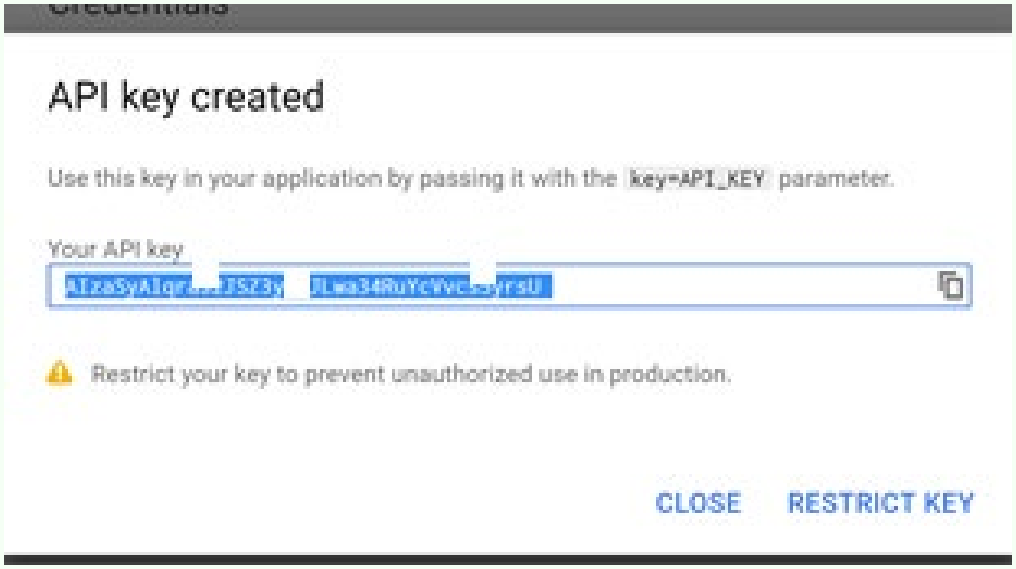
- Use the link provided in the google_maps_api.xml file that Android Studio created for you:
  - Copy the link provided in the google_maps_api.xml file and paste it into your browser.
  - The link takes you to the **Google Cloud Platform Console** and supplies the required information to the Google Cloud Platform Console via URL parameters, thus reducing the manual input required from you.
- Follow the instructions to **create a new project** on the Google Cloud Platform Console **or select an existing project**.

Getting a Maps API Key



Getting a Maps API Key





Adding the API Key to your application

- **To add the key to your application:**
  - **Add it to** google_maps_api.xml file, or:
    - In AndroidManifest.xml, add the following element as a child of the <application> element, by inserting it just before the closing tag </application>:

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

substituting your API key for *API_KEY*.

- This element sets the key com.google.android.geo.API_KEY to the value *API_KEY* and makes the API key visible to any **MapFragment** in your application.

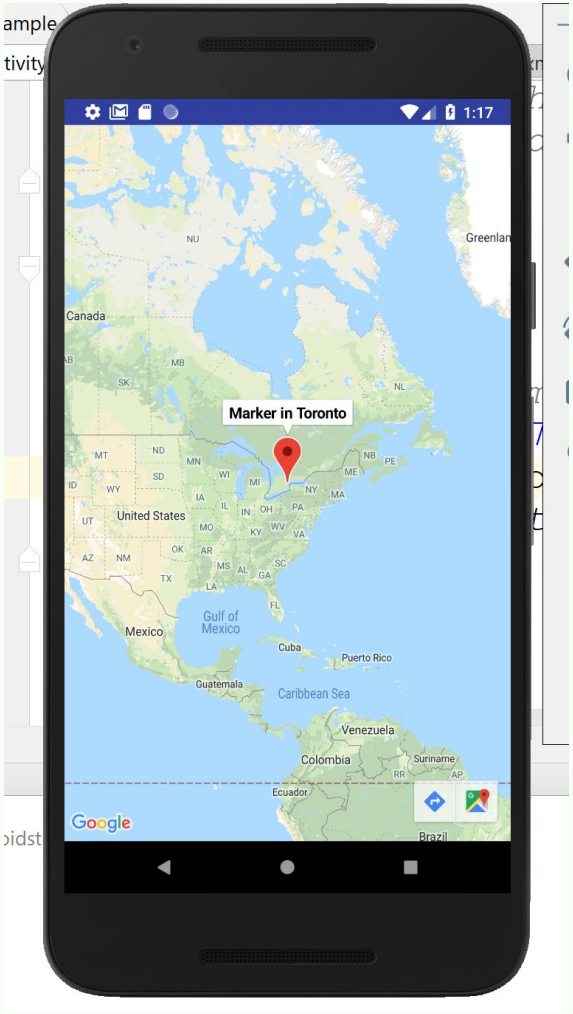
Google Maps project

```
public class MapsActivity extends FragmentActivity implements OnMapReadyCallback {
    private GoogleMap mMap;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_maps);
        // Obtain the SupportMapFragment and get notified when the map is ready to be used.
        SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager()
                .findFragmentById(R.id.map);
        mapFragment.getMapAsync( onMapReadyCallback: this);
    }
    @Override
    public void onMapReady(GoogleMap googleMap) {
        mMap = googleMap;

        // Add a marker in Sydney and move the camera
        LatLng toronto = new LatLng( v: 43.6,  v1: -79.4);
        mMap.addMarker(new MarkerOptions().position(toronto).title("Marker in Toronto"));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(toronto));
    }
}
```

Running the app

- Change the Latitude, Longitude to Toronto values
- Run the app



Adding built-in zoom control

- To add a parameter to activity_maps.xml that sets the **uiZoomControls to true**:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment“
    map:uiZoomControls="true“
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MapsActivity" />
```

- You can also programmatically zoom in or out of the map using the animateCamera() method of the **GoogleMap** class.

Programmatically zoom in or out

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_3:
            mMap.animateCamera(CameraUpdateFactory.zoomIn());
            break;
        case KeyEvent.KEYCODE_1:
            mMap.animateCamera(CameraUpdateFactory.zoomOut());
            break;
    }
    return super.onKeyDown(keyCode, event);
}
```

Changing Map Views

- By default, Google Maps is displayed in ***map view***, which is basically drawings of streets and places of interest.
- You can also set Google Maps to display in ***satellite view*** using the setMapType() method of the **GoogleMap** class:

```
mMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
// mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
// mMap.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
// mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
mMap.setBuildingsEnabled(true);
mMap.setIndoorEnabled(true);
mMap.setTrafficEnabled(true);
```

Getting the Location That Was Touched

- To get the latitude and longitude of a point on the Google Map that was touched, you must set a onMapClickListener:

```
mMap.setOnMapClickListener(new GoogleMap.OnMapClickListener()
{
@Override
public void onMapClick(LatLng point) {
Log.d("DEBUG","Map clicked [" + point.latitude +
" / " + point.longitude + "]");
}
});
}
```

Displaying Interactive Map Markers

- You can add interactive, customizable markers to a Google Map using the addMarker method:

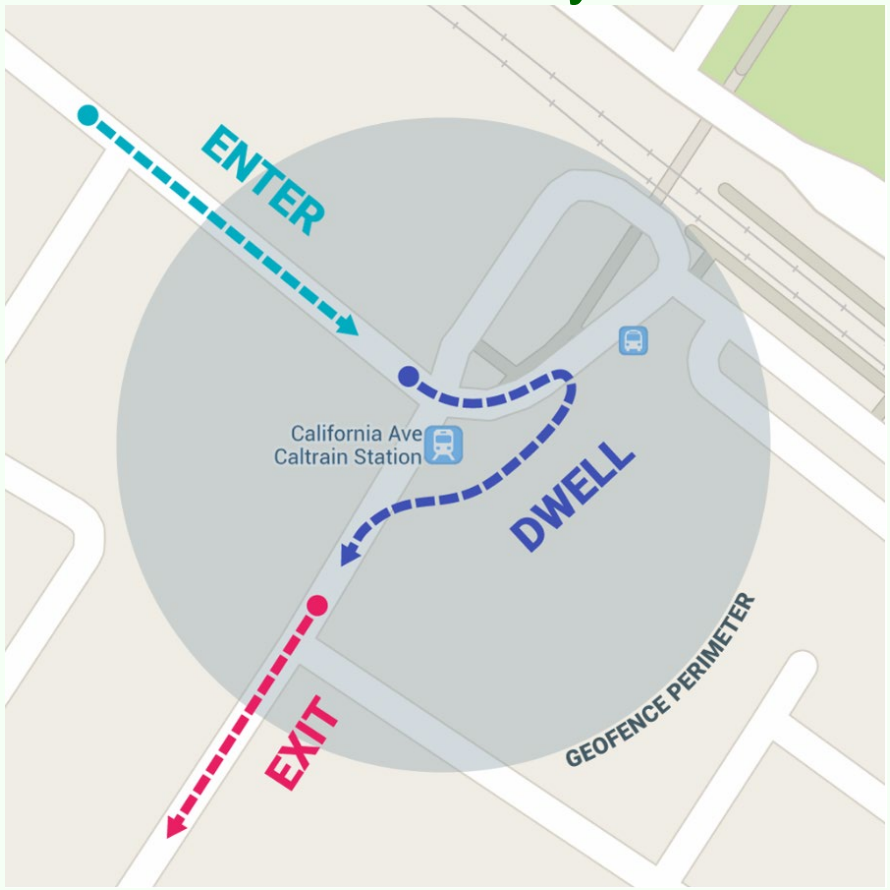
```
LatLng position = new LatLng(lat, lng);
Marker newMarker = mMap.addMarker(new
MarkerOptions().position(position));
```

- By providing a title and snippet text, markers can become interactive:

```
Marker newMarker = mMap.addMarker(new
MarkerOptions().position(latLng).title("Honeymoon
Location").snippet("This is where I had my
honeymoon!"));
```

Geofences

- **Geofencing** combines awareness of the user's current location with awareness of the user's proximity to locations that may be of interest.



Setting and Managing Geofences

- **Geofences** are defined by a given **latitude** and **longitude**, combined with an effective **radius**.
- Using Geofences, you can **set Pending Intents that are fired based on the user's proximity** to specified locations.
- Your app can specify up to 100 Geofences per device user.
- The **Geofence API** is part of the Google Play service Location Services library, which **must be added as a dependency** to your app module's **build.gradle** file after you've installed Google Play services as described earlier in this chapter:

```
dependencies {
……
implementation 'com.google.android.gms:play-services-location:15.0.1'
}
```

Setting and Managing Geofences

- The Geofence API **requires the *fine* location permission** to be defined in your application manifest:

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

- As a dangerous permission, fine location access must also be requested at run time prior to setting a Geofence:

```
// Check if we have permission to access high accuracy fine location.
int permission = ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION);
// If permission is granted, fetch the last location.
if (permission == PERMISSION_GRANTED) {
setGeofence();
} else {
// If permission has not been granted, request permission.
ActivityCompat.requestPermissions(this,
new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
LOCATION_PERMISSION_REQUEST);
}
```

Setting and Managing Geofences

1. Create a GeofencingClient object
2. Create a Geofence object
3. Create a GeofencingRequest object and **add the geofence** to it
4. Create a PendingIntent
5. Initiate a Geofencing Request
6. Receive information in onReceive method of a GeofenceBroadCastReceiver.

```
GeofencingClient geofencingClient =
LocationServices.getGeofencingClient(this);
```

- You can define a Geofence around a given location using the Geofence.Builder class.
- Specify a **unique ID**, the **center point** (using longitude and latitude values), a **radius** around that point, an **expiry time-out**, and the **transition types** that will cause the Pending Intent to fire: **entry**, **exit**, and/or dwell.

Setting and Managing Geofences

```
Geofence newGeofence = new Geofence.Builder()
.setRequestId(id) // unique name of geofence
.setCircularRegion(location.getLatitude(),
location.getLongitude(), 30) // 30 meter radius.
.setExpirationDuration(Geofence.NEVER_EXPIRE) // Or
expiration time in ms
.setLoiteringDelay(10*1000) // Dwell after 10 seconds
.setNotificationResponsiveness(10*1000) // Notify within 10
seconds
.setTransitionTypes(Geofence.GEOFENCE_TRANSITION_D
WELL)
.build();
```

- **To add** a Geofence you need to pass a **GeofencingRequest** and a **Pending Intent** to fire to the Geofencing Client.

Setting and Managing Geofences

```
// create a geofencing request
GeofencingRequest geofencingRequest = new
GeofencingRequest.Builder()
.addGeofence(newGeofence)
.setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_DW
ELL)
.build();
```

- To specify the Intent to fire, you use a PendingIntent, a class that wraps an Intent in a kind of method pointer:

```
// create a pending intent to fire to the geofencing client
Intent intent = new Intent(this,
GeofenceBroadcastReceiver.class);
PendingIntent geofenceIntent =
PendingIntent.getBroadcast(this, -1, intent, 0);
```

Setting and Managing Geofences

- Initiating a Geofencing Request

```
geofencingClient.addGeofences(geofencingRequest, geofenceIntent)
.addOnSuccessListener(this, new OnSuccessListener<Void>() {
@Override
public void onSuccess(Void aVoid) {
// TODO Geofence added.
}
})
.addOnFailureListener(this, new OnFailureListener() {
@Override
public void onFailure(@NonNull Exception e) {
Log.d(TAG, "Adding Geofence failed", e);
// TODO Geofence failed to add.
}
});
```

Setting and Managing Geofences

- When the Location Service detects that you have **crossed the Geofence radius boundary**, the **Pending Intent fires**.
- Depending on your Pending Intent, the Intent fired when the Geofence is triggered can trigger a Broadcast Receiver.

```
public class GeofenceBroadcastReceiver extends BroadcastReceiver {

  private static final String TAG = "GeofenceReceiver";

  @Override
  public void onReceive(Context context, Intent intent) {
    GeofencingEvent geofencingEvent = GeofencingEvent.fromIntent(intent);
    if (geofencingEvent.hasError()) {
      int errorCode = geofencingEvent.getErrorCode();
      String errorMessage =
        GeofenceStatusCodes.getStatusCodeString(errorCode);
      Log.e(TAG, errorMessage);
    } else {
      // Get the transition type.
      int geofenceTransition = geofencingEvent.getGeofenceTransition();

      // A single event can trigger multiple geofences.
      // Get the geofences that were triggered.
      List<Geofence> triggeringGeofences =
        geofencingEvent.getTriggeringGeofences();

      // TODO React to the Geofence(s) transition(s).
    }
  }
}
```

Adding Contextual Awareness

- The Awareness API **combines multiple signals** including location, user context, and the environment to provide a mechanism that allows you **to add context-based functionality to your app**, with minimal impact on system resources.
- The Awareness API currently supports up to seven different context signals:
  - Time
  - Location – physical user location
  - User Activity
  - Nearby Beacons
  - Places – nearby businesses
  - Device state – headphone connection state
  - Environmental conditions – local weather

Adding Contextual Awareness

```
dependencies {
[... Existing Dependencies ...]
implementation 'com.google.android.gms:play-services-awareness:17.0.1'
}
```

- Requires you to create and connect an instance of the GoogleApiClient:

```
mGoogleApiClient = new GoogleApiClient.Builder(this)
.addApi(Awareness.API)
.enableAutoManage(this, // MainActivity
this) // OnConnectionFailedListener
.build();
```

- When auto-managed, your Google API Client will automatically connect during onStart and disconnect after onStop.

Adding Contextual Awareness

- You'll need to obtain a key for the Awareness API from developers.google .com/awareness/android-api/get-a-key.
- add it to your application manifest immediately before the closing application tag enclosed within a meta-data node as shown in the following snippet:

```
<meta-data
android:name="com.google.android.awareness.API_KEY"
android:value="[YOUR_API_KEY]"
/>
```

- Add also meta-data nodes for other APIs.
- Use static get methods of the **Snapshot API** to obtain information about context signals available.

Retrieving Snapshot context signal results

```
// Each type of contextual information in the snapshot API has a corresponding "get"
// method. For instance, this is how to get the user's current Activity.
    Awareness.getSnapshotClient(this).getDetectedActivity()
        .addOnSuccessListener(new
OnSuccessListener<DetectedActivityResponse>() {
            @Override
            public void onSuccess(DetectedActivityResponse dar) {
                ActivityRecognitionResult arr = dar.getActivityRecognitionResult();
                // getMostProbableActivity() is good enough for basic Activity detection.
                // To work within a threshold of confidence,
                // use ActivityRecognitionResult.getProbableActivities() to get a list of
                // potential current activities, and check the confidence of each one.
                DetectedActivity probableActivity = arr.getMostProbableActivity();
                int confidence = probableActivity.getConfidence();
                String activityStr = probableActivity.toString();
                mLogFragment.getLogView().println("Activity: " + activityStr
                    + ", Confidence: " + confidence + "/100");
            }
        });
```

References

- Textbook
- Android Documentation:
  - https://developers.google.com/maps/documentation/android-sdk/start
  - https://developer.android.com/guide/topics/location/index.html
  - https://developer.android.com/guide/topics/location/strategies.html
  - https://developer.android.com/training/location/
  - https://developers.google.com/maps/documentation/android-api/
  - https://developers.google.com/maps/documentation/android-api/start
  - https://developers.google.com/maps/documentation/javascript/geolocation
  - https://developers.google.com/maps/faq
  - https://developers.google.com/awareness
  - https://developers.google.com/maps/documentation/android-sdk/start#maps_android_mapsactivity-kotlin
  - https://developers.google.com/codelabs/maps-platform/maps-platform-101-android#0
  - https://developer.android.com/training/location/geofencing
  - https://developer.android.com/codelabs/advanced-android-kotlin-training-geofencing#0