Android Development with Kotlin

Location Map and Sensors

Building Location Aware Apps:

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.

So, to develop the **location aware apps** we follow this workflow:

1) Request the proper permissions
2) Receive location updates
3) Optimise power usage
4) Add maps to your app

Request the proper permissions:

To protect user privacy, apps that use location services must request location permissions.

When you request location permissions, follow the same best practices as you would for any **other runtime permission**. One important difference when it comes to location permissions is that the system includes multiple permissions related to location. Which permissions you request, and how you request them, depend on the location requirements for your app's use case.

Android's location permissions deal with the following categories of location access:

1) Foreground location
2) Background location

Foreground location:

It's recommended that you declare a **foreground service type** of location, as shown in the following code snippet. On Android 10 (API level 29), you must declare this foreground service type.

```
<!-- Recommended for Android 9 (API level 28) and lower. -->
<!-- Required for Android 10 (API level 29). -->
<service
    android:name="MyNavigationService"
    android:foregroundServiceType="location" ... >
    <!-- Any inner elements would go here. -->
</service>
```

You declare a need for foreground location when your app requests either the ACCESS_COARSE_LOCATION permission or the ACCESS_FINE_LOCATION permission, as shown in the following snippet:

```
<manifest ... >
  <!-- To request foreground location access, declare one of these
permissions. -->
  <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Background location:

On Android 10 (API level 29), the ACCESS_BACKGROUND_LOCATION permission in your app's manifest in order to request background location access at runtime. On earlier versions of Android, when your app receives foreground location access, it automatically receives background location access as well.

```
<manifest ... >
  <!-- Required only when requesting background location access on
       Android 10 (API level 29). -->
  <uses-permission
android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
</manifest>
```

Request location updates:

Appropriate use of location information can be beneficial to users of your app. 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.

Get the last known location:

The last known location of the device provides a handy base from which to start, ensuring that the app has a known location before starting the periodic location updates.

In your activity's `onCreate()` method, create an instance of the Fused Location Provider Client as the following code snippet shows.

```
private lateinit var fusedLocationClient: FusedLocationProviderClient

override fun onCreate(savedInstanceState: Bundle?) {
```

```
    // ...

    fusedLocationClient =
LocationServices.getFusedLocationProviderClient(this)
}
```

To request the last known location, call the `getLastLocation()` method. The following code snippet illustrates the request and a simple handling of the response:

```
fusedLocationClient.lastLocation
        .addOnSuccessListener { location : Location? ->
            // Got last known location. In some rare situations this
can be null.
        }
```

Making a location request:

```
override fun onResume() {
    super.onResume()
    if (requestingLocationUpdates) startLocationUpdates()
}

private fun startLocationUpdates() {
    fusedLocationClient.requestLocationUpdates(locationRequest,
            locationCallback,
            Looper.getMainLooper())
}
```

Defining the location update callback:

```
private lateinit var locationCallback: LocationCallback

// ...

override fun onCreate(savedInstanceState: Bundle?) {
    // ...

    locationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult?)
{
            locationResult ?: return
            for (location in locationResult.locations){
                // Update UI with location data
                // ...
            }
        }
```

```
    }
}
```

Stopping the location updates:

```
override fun onPause() {
    super.onPause()
    stopLocationUpdates()
}

private fun stopLocationUpdates() {
    fusedLocationClient.removeLocationUpdates(locationCallback)
}
```

Optimise the power usage:

Understanding the battery drain:

➔ Location gathering and battery drain are directly related in the following aspects:

1) **Accuracy:** The precision of the location data. In general, the higher the accuracy, the higher the battery drain.
2) **Frequency:** How often location is computed. The more frequent location is computed, the more battery is used.
3) **Latency:** How quickly location data is delivered. Less latency usually requires more battery.

Accuracy:

You can specify location accuracy using the **setPriority()** method, passing one of the following values as the argument:

➔ **PRIORITY_HIGH_ACCURACY** provides the most accurate location possible, which is computed using as many inputs as necessary (it enables GPS, Wi-Fi, and cell, and uses a variety of Sensors), and may cause significant battery drain.
➔ **PRIORITY_BALANCED_POWER_ACCURACY** provides accurate location while optimizing for power. Very rarely uses GPS. Typically uses a combination of Wi-Fi and cell information to compute device location.
➔ **PRIORITY_LOW_POWER** largely relies on cell towers and avoids GPS and Wi-Fi inputs, providing coarse (city-level) accuracy with minimal battery drain.
➔ **PRIORITY_NO_POWER** receives locations passively from other apps for which location has already been computed.

The location needs of most apps can be satisfied using the balanced power or low power options. High accuracy should be reserved for apps that are running in the foreground and require real time location updates (for example, a mapping app).

Frequency:

You can specify location frequency using two methods:

- ➔ Use the setInterval() method to specify the interval at which location is computed for your app.
- ➔ Use the setFastestInterval() to specify the interval at which location computed for other apps is delivered to your app.

You should pass the largest possible value when using setInterval(). This is especially true for background location gathering, which is often a source of unwelcome battery drain. Using intervals of a few seconds should be reserved for foreground use cases. The background location limits introduced in Android 8.0 enforce these strategies, but your app should strive to enforce them on Android 7.0 or lower devices.

Latency:

You can specify latency using the **setMaxWaitTime()** method, typically passing a value that is several times larger than the interval specified in the **setInterval()** method. This setting delays location delivery, and multiple location updates may be delivered in batches. These two changes help minimize battery consumption.

If your app doesn't immediately need a location update, you should pass the largest possible value to the **setMaxWaitTime()** method, effectively trading latency for more data and battery efficiency.

When using geofences, apps should pass a large value into the **setNotificationResponsiveness()** method to preserve power. A value of five minutes or larger is recommended.

Adding maps to your app:

Add a fragment:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:id="@+id/map"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Add a fragment:

You can also add a **MapFragment** to an **Activity** in code. To do this, create a new MapFragment instance, and then call **FragmentTransaction.add()** to add the **Fragment** to the current **Activity**.

```
val mapFragment: SupportMapFragment = SupportMapFragment.newInstance()
val fragmentTransaction = supportFragmentManager.beginTransaction()
fragmentTransaction.add(R.id.my_container, mapFragment)
fragmentTransaction.commit()
```

Adding map code:

```
class BasicMapDemoActivity : AppCompatActivity(),
  OnMapReadyCallback {
...
}
```

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

Setting a marker:

```
override fun onMapReady(googleMap: GoogleMap?) {
  googleMap ?: return
  with(googleMap) {
      addMarker(MarkerOptions()
          .position(LatLng(0.0, 0.0))
          .title("Marker"))
  }
}
```

Sensors in Android:

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

Following are the type of sensors:

- ➔ Sensors
- ➔ Motion Sensors
- ➔ Position Sensors
- ➔ Environment Sensors

The Android platform supports three broad categories of sensors:

a) Motion sensors

These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

b) Environmental sensors

These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

c) Position sensors

These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

Accessing an accelerometer in android:

```
 mSensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
 mAccelerometer =
mSensorManager!!.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
```

Getting values:

```
private void getAccelerometer(SensorEvent event)
{
var values = event.values;
// Movement
var x = values[0];
var y = values[1];
 var z = values[2];
}
```

We have covered all the concepts related to location, maps and sensors in the lecture. Feel free to explore more using the following links:

Location access: https://developer.android.com/training/location

Battery: https://developer.android.com/guide/topics/location/battery

Google Maps Documentation: https://developers.google.com/maps/documentation/android-sdk/map#kotlin

Request Updates: https://developer.android.com/training/location/request-updates

Current Location Access: https://developer.android.com/training/location/retrieve-current

Permission Access: https://developer.android.com/training/location/permissions