# UNIT - IV: MESSAGING, LOCATION-BASED SERVICES, AND NETWORKING

SMS messaging, Sending e-mail, Displaying maps, Getting location data, Monitoring a location, Consuming web services using HTTP.

## INTRODUCTION

In this unit, students will explore essential mobile app functionalities that enhance user interaction and data exchange. The topics include SMS messaging for seamless text communication and email integration for handling user correspondence. Additionally, the unit covers displaying maps, which is crucial for location-based services, and obtaining real-time location data to personalize user experiences. Students will also learn how to monitor a location for tracking purposes and consume web services via HTTP to enable efficient data transfer between client and server. These skills form the foundation for building versatile, data-driven mobile applications.

## SMS MESSAGING

SMS messaging is one of the main killer applications on a mobile phone today — for some users as necessary as the phone itself. Any mobile phone you buy today should have at least SMS messaging capabilities, and nearly all users of any age know how to send and receive such messages.

### 1. Setting up Permissions

For any SMS functionality, your app needs to request the appropriate permissions. In Android 13, you must request **SEND_SMS**, **RECEIVE_SMS**, and optionally **READ_SMS** permissions.

Add the following permissions to AndroidManifest.xml:

```
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
```

Since Android 6.0 (API level 23), apps must also request these permissions at runtime. Here's an example of how to handle runtime permission requests for SMS in Kotlin:

```
if (ContextCompat.checkSelfPermission(this,
Manifest.permission.SEND_SMS) !=
PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this,
arrayOf(Manifest.permission.SEND_SMS), REQUEST_SMS_PERMISSION)
}
```

**2. Sending SMS using SmsManager**

We can send SMS messages using the SmsManager class. It provides simple methods to send text messages.

```
val phoneNumber = "1234567890"
val message = "Hello, this is a test message!"

val smsManager = SmsManager.getDefault()
smsManager.sendTextMessage(phoneNumber, null, message, null, null)
```

In this example:

- **phoneNumber**: The recipient’s phone number.
- **message**: The text you want to send.

- The other parameters (null values) are for service center address, delivery intent, and sent intent, which are optional.

**3. Receiving SMS Messages**

To receive an SMS, we need to create a BroadcastReceiver that listens for SMS broadcasts from the system. Android automatically sends a broadcast when a new SMS is received.

**Step-by-Step Guide**:

- **Create a BroadcastReceiver**: This class will capture the broadcast that the system sends when an SMS is received.
- **Declare the receiver in the manifest**: The receiver must be declared in the manifest with the action android.provider.Telephony.SMS_RECEIVED.

**Example: SMS Receiver**

**SmsReceiver.kt**

```
class SmsReceiver : BroadcastReceiver() {
  override fun onReceive(context: Context?, intent: Intent?) {
    if (intent != null && Telephony.Sms.Intents.SMS_RECEIVED_ACTION ==
intent.action) {
      val bundle = intent.extras
      if (bundle != null) {
        val pdus = bundle.get("pdus") as Array<*>
        for (pdu in pdus) {
          val format = bundle.getString("format")
          val smsMessage = SmsMessage.createFromPdu(pdu as ByteArray,
format)

          // Extract sender and message content
          val sender = smsMessage.originatingAddress
          val messageBody = smsMessage.messageBody

          // Example: Display the message in a Toast
          Toast.makeText(context, "Message from $sender: $messageBody",
Toast.LENGTH_LONG).show()
            }
        }
    }
```

**AndroidManifest.xml** Add the receiver and permission declaration in your manifest:

```
  <uses-permission android:name="android.permission.RECEIVE_SMS"/>
  <uses-permission android:name="android.permission.READ_SMS"/>
  <application ... >
  <receiver android:name=".SmsReceiver" android:enabled="true"
android:exported="true">
      <intent-filter android:priority="999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"
/>
      </intent-filter>
  </receiver>
</application>
```

Here:

- Telephony.Sms.Intents.SMS_RECEIVED_ACTION is the action indicating that an SMS has been received.
- The **pdus** array contains Protocol Data Units (PDUs), which represent the received SMS message.

**Key Points:**

- **SmsManager**: Use it for sending SMS messages programmatically.
- **BroadcastReceiver**: Use it to listen for incoming SMS messages.
- **Permissions**: Always request runtime permissions and handle permission denial gracefully.

## SENDING EMAIL IN ANDROID

In Android, sending emails can be efficiently accomplished using intents. Intents in Android are a messaging object used to request an action from another app component. In the context of sending emails, we use **implicit intents**, which allow communication with components provided by other applications, like email clients.

### Implicit Intents for Email

To send an email, an implicit intent with the action Intent.ACTION_SENDTO is commonly used. The ACTION_SENDTO action indicates that the intent is for sending data to a specific recipient. The **URI scheme** mailto: is used to specify that the intent is directed toward email-related activities.

By using this method, the Android system displays a chooser dialog to let the user select their preferred email application, thus leveraging the existing email client on the device. This is highly beneficial as it allows the application to avoid handling the complexities of composing and sending emails directly, while also providing a seamless user experience.

### Key Intent Extras for Sending Emails

When composing an email using an intent, there are certain key data fields (extras) you can include:

- **EXTRA_EMAIL**: Specifies the recipient(s) of the email in the form of an array of strings.
- **EXTRA_SUBJECT**: Provides the subject of the email.
- **EXTRA_TEXT**: Represents the body content of the email.
- **EXTRA_CC** and **EXTRA_BCC**: Optionally, these can be used to add CC (Carbon Copy) and BCC (Blind Carbon Copy) recipients.

**PROCESS OF SENDING E MAIL IN ANDROID**

- **Creating the Intent**: An Intent object is created with the action ACTION_SENDTO and the data URI mailto: to ensure only email clients respond.
- **Adding Email Details**: The recipient email address, subject, and body are added as extras.
- **Launching the Intent**: The intent is triggered using startActivity(), which launches the email client for the user to send the email.

**Step 1: Set Up the Email Intent**

The Intent class is used to start an email client. You'll need to specify the action ACTION_SENDTO and the recipient, subject, and body of the email.

**Step 2: Add Email Sending Code**

```
import android.content.Intent
import android.net.Uri

fun sendEmail(recipient: String, subject: String, body: String) {
    val intent = Intent(Intent.ACTION_SENDTO).apply {
        data = Uri.parse("mailto:")
        putExtra(Intent.EXTRA_EMAIL, arrayOf(recipient))
        putExtra(Intent.EXTRA_SUBJECT, subject)
        putExtra(Intent.EXTRA_TEXT, body)
    }
    if (intent.resolveActivity(packageManager) != null) {
        startActivity(intent)
    }
}
```

**Step 3: Call the Function**

Invoke the sendEmail function from your activity or fragment by passing the recipient, subject, and body text.

```
sendEmail(
    recipient = "example@example.com",
    subject = "Hello from Android",
    body = "This is a test email from my app."
)
```

---

## DISPLAYING MAPS IN ANDROID

Displaying maps in Android is primarily achieved using the **Google Maps API**, which allows developers to embed Google Maps into their applications. This service provides interactive and customizable map views that support functionalities such as zoom, markers, user location, and more.

Android offers two main libraries for displaying maps:

- **Google Maps SDK for Android**: Allows embedding maps powered by Google Maps into Android apps.
- **OSMDroid**: An open-source alternative to Google Maps, useful for offline mapping or when developers prefer not to use Google's services.

**Google Maps API Overview**

The **Google Maps SDK for Android** provides a rich set of APIs to add maps and various features, such as markers, polylines, and polygons. It supports interactive features like zooming, scrolling, and adding overlays for better visual representation.

**Steps to Implement Google Maps in Android**

1. **Set Up Google Maps API**: To start with Google Maps, you need to register your app in the Google Cloud Console and obtain an API key.
   - Enable the **Google Maps Android API** service.
   - Generate the **API key** that will be used in your Android project.
2. **Modify the Manifest**: Include the required permissions and API key in the AndroidManifest.xml file. The permissions ensure the app can access the device's location and internet to display the maps.

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

3. **Add Google Play Services Dependency**: Add the required dependencies for Google Maps in your build.gradle file.

   *implementation 'com.google.android.gms:play-services-maps:18.1.0'*
4. **Create a Map Fragment**: A MapFragment or SupportMapFragment is used to embed the map in the layout. You define this fragment in your activity's layout file.

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

5. **Initialize the Map in Activity**: In your activity or fragment, you need to initialize the map and manage its lifecycle. You can set up various features such as adding markers, zoom controls, and gestures.

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.google.android.gms.maps.CameraUpdateFactory
class MapsActivity : AppCompatActivity(), OnMapReadyCallback {
  private lateinit var map: GoogleMap
  override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_maps)

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

  override fun onMapReady(googleMap: GoogleMap) {
      map = googleMap
      val location = LatLng(37.7749, -122.4194) // Coordinates for San
Francisco
      map.addMarker(MarkerOptions().position(location).title("Marker in
San Francisco"))
      map.moveCamera(CameraUpdateFactory.newLatLngZoom(location,
12.0f))
  }
}
```

**Important Components**

1. **SupportMapFragment**: This is a specialized fragment to display a map using Google Maps.
2. **GoogleMap**: The object that controls the map and allows customization (e.g., setting markers, adding overlays).
3. **MarkerOptions**: Used to add markers on the map at specific latitude/longitude coordinates.
4. **CameraUpdateFactory**: Manages the camera (view) on the map to zoom in or out and centre on certain locations.

---

## GETTING LOCATION DATA IN ANDROID

In Android, obtaining location data allows your app to provide features such as navigation, location-based recommendations, and other location-related services. Android offers APIs through the **Fused Location Provider (FLP)**, which intelligently combines data from GPS, Wi-Fi, and mobile networks to provide location updates efficiently.

The **Fused Location Provider** is the recommended way to get location data, as it optimizes for accuracy and battery consumption. It simplifies the location acquisition process by automatically choosing the best provider (GPS, network, or Wi-Fi) for the given scenario.

**Key Components to Obtain Location Data**

1. **Google Play Services Location APIs**: The **Google Play Services Location API** provides a high-level API for accessing location services.
2. **LocationRequest**: Defines the type of location requests such as interval, priority, and accuracy (e.g., high accuracy for GPS or low power for network).
3. **FusedLocationProviderClient**: This is the main class for interacting with the location services to get the current location or receive periodic location updates.
4. **Permissions**: You need to request location-related permissions in your app, such as ACCESS_FINE_LOCATION for GPS-based tracking and ACCESS_COARSE_LOCATION for network-based location.

```
fun startLocationUpdates() {
   locationRequest = LocationRequest.create().apply {
      interval = 10000 // 10 seconds
      fastestInterval = 5000 // 5 seconds
      priority = LocationRequest.PRIORITY_HIGH_ACCURACY
   }
   locationCallback = object : LocationCallback() {
      override fun onLocationResult(locationResult: LocationResult) {
         val location = locationResult.lastLocation
         if (location != null) {
            println("Updated Location: ${location.latitude},
${location.longitude}")
         }
      }
   }
   fusedLocationClient.requestLocationUpdates(locationRequest,
locationCallback, null)
}
override fun onStop() {
   super.onStop()
   fusedLocationClient.removeLocationUpdates(locationCallback)
}
```

**Important Notes**

1. **Permissions Handling**: Always ensure that you handle permissions properly, especially with newer Android versions that require runtime permission checks.
2. **Battery Consumption**: Using high-accuracy GPS continuously may drain the battery quickly. Therefore, choose the appropriate priority for the LocationRequest based on your use case (e.g., PRIORITY_BALANCED_POWER_ACCURACY for less frequent updates).

## MONITORING A LOCATION IN ANDROID

Monitoring a location in Android involves continuously tracking the user's location and taking actions when specific conditions are met, such as when the user enters or exits a geographical area. This can be achieved through **Geofencing** or continuous **Location Updates** using the **Fused Location Provider**.

**Two Main Approaches for Monitoring Location:**

1. **Geofencing**: Allows monitoring specific geographical regions (circular areas) and triggers events when the user enters or exits those regions.
2. **Continuous Location Updates**: Provides continuous updates of the user's location, allowing you to track movement in real-time.

### What is Geofencing?

Geofencing allows apps to define geographical areas, known as geofences, and monitor when a device enters or exits these areas. It's ideal for use cases such as sending reminders when entering a store or triggering notifications when exiting a defined location.

**Creating and Add a Geofence**

1. **Create the Geofence**: A geofence is defined by a location (latitude and longitude) and a radius. It can also include expiration time and transition types (enter, exit).
2. **GeofencingRequest**: This defines how geofences should behave and handles the creation of multiple geofences.
3. **PendingIntent**: A PendingIntent is used to define what should happen when a geofence is triggered (such as starting a service or broadcasting a message).

```
class GeofenceActivity : AppCompatActivity()
private lateinit var geofencingClient: GeofencingClient
private lateinit var geofencePendingIntent: PendingIntent

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_geofence)

    geofencingClient = LocationServices.getGeofencingClient(this)
    val geofence = Geofence.Builder()
        .setRequestId("GEOFENCE_ID")
        .setCircularRegion(
            37.7749,
            -122.4194,
            100f
        ) // Coordinates for San Francisco, radius 100 meters
        .setExpirationDuration(Geofence.NEVER_EXPIRE)
        .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER or
Geofence.GEOFENCE_TRANSITION_EXIT)
        .build()

    val geofencingRequest = GeofencingRequest.Builder()
        .setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
        .addGeofence(geofence)
        .build()
    geofencePendingIntent = PendingIntent.getBroadcast(
        this,
        0,
        Intent(this, GeofenceBroadcastReceiver::class.java),
        PendingIntent.FLAG_UPDATE_CURRENT
    )
}
```

**Create a Broadcast Receiver for Geofence Transitions**

Create a BroadcastReceiver to handle events when the user enters or exits the defined geofence.

```
class GeofenceBroadcastReceiver : BroadcastReceiver() {

  override fun onReceive(context: Context, intent: Intent) {
    val geofencingEvent = GeofencingEvent.fromIntent(intent)

    if (geofencingEvent.hasError()) {
      // Handle error
      return
    }
    val geofenceTransition = geofencingEvent.geofenceTransition
    if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER
||
      geofenceTransition == Geofence.GEOFENCE_TRANSITION_EXIT) {
    }
  }
}
```

## CONSUMING WEB SERVICES USING HTTP IN ANDROID

In Android, consuming web services via HTTP is a common requirement, allowing apps to interact with remote servers, fetch data, or submit information. There are multiple ways to implement HTTP requests in Android, but two popular approaches include:

- **Retrofit**: A type-safe HTTP client for Android, which simplifies the process of making network requests and handling responses.
- **HttpURLConnection**: A built-in class in Android for managing HTTP requests, though it is more verbose compared to libraries like Retrofit.

**Retrofit for HTTP Requests**

Retrofit is highly preferred for modern Android development due to its ease of use and powerful features like automatic JSON parsing. Here's how you can use Retrofit to consume web services.

First, add the required dependencies for Retrofit and a converter (like Gson for JSON parsing) to your app's build.gradle file.

```
dependencies {

   implementation 'com.squareup.retrofit2:retrofit:2.9.0'

   implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

}
```

**Define API Endpoints**

You define API endpoints using an interface. For instance, to fetch a list of posts from a sample API.

```
import retrofit2.Call
import retrofit2.http.GET
data class Post(val userId: Int, val id: Int, val title: String, val body: String)
interface ApiService {
   @GET("posts")
   fun getPosts(): Call<List<Post>>
}
```

**Setup Retrofit Instance**

Create a singleton Retrofit instance with a base URL and a converter factory.

```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
```

```
object RetrofitInstance {
  private val retrofit by lazy {
    Retrofit.Builder()
      .baseUrl("https://jsonplaceholder.typicode.com/")
      .addConverterFactory(GsonConverterFactory.create())
      .build()
  }
  val api: ApiService by lazy {
    retrofit.create(ApiService::class.java)
  }
}
```

**Make an HTTP Request**

Now, make the HTTP request in your Activity or Fragment

```
private fun fetchPosts() {
  val call = RetrofitInstance.api.getPosts()
  call.enqueue(object : Callback<List<Post>> {
    override fun onResponse(call: Call<List<Post>>, response:
Response<List<Post>>) {
      if (response.isSuccessful) {
        response.body()?.let { posts ->
          for (post in posts) {
            println("Post: ${post.title}")
          }
        }
      }
    }
    override fun onFailure(call: Call<List<Post>>, t: Throwable) {
      println("Error: ${t.message}")
    }
  })
}
```

**HttpURLConnection for HTTP Requests**

If you prefer not to use external libraries, Android's HttpURLConnection is a lower-level solution to make HTTP requests. This is more manual but built into the framework.

```
fun fetchUsingHttpURLConnection() {
  val url = URL("https://jsonplaceholder.typicode.com/posts")
  val connection = url.openConnection() as HttpURLConnection
  try {
    connection.requestMethod = "GET"
    connection.connect()
    val responseCode = connection.responseCode
    if (responseCode == HttpURLConnection.HTTP_OK) {
      val inputStream = connection.inputStream
      val content = inputStream.bufferedReader().use { it.readText() }
      println("Response: $content")
    } else {
      println("Failed to fetch data. Response code: $responseCode")
    }
  } catch (e: Exception) {
    e.printStackTrace()
  } finally {
    connection.disconnect()
  }
}
```

**Handling Permissions**

If you're making network requests in an Android app, ensure you add the required permission in your AndroidManifest.xml file.

*<uses-permission android:name="android.permission.INTERNET"/>*

For apps targeting Android 9 (API level 28) or higher, make sure to configure **cleartext traffic** if you're using HTTP instead of HTTPS

```
<application
    android:usesCleartextTraffic="true"
    ...>
</application>
```