Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED ANDROID DEVELOPMENT (WEEKS 16-18)

ASYNCTASK VS COROUTINES IN ANDROID

CHAPTER 1: INTRODUCTION TO BACKGROUND PROCESSING IN ANDROID

1.1 Why Do We Need Background Processing?

🔹 Android applications often need to perform long-running tasks such as:

✓ Fetching data from a server (API calls).

✓ Reading/writing to a database.

✓ Processing large amounts of data.

✓ Performing file uploads or downloads.

🔹 Running such tasks on the **Main (UI) Thread** can **freeze** the UI, causing **lag and ANR (Application Not Responding) errors**.

🔹 To prevent this, **Android provides solutions** like **AsyncTask (deprecated) and Coroutines** for handling background tasks efficiently.

📌 **Example Use Cases:**

- Fetching user data from an API without blocking the UI.

- Loading images in the background while displaying placeholders.
- Processing large files without making the app unresponsive.

CHAPTER 2: UNDERSTANDING ASYNCTASK

2.1 What is AsyncTask?

🔹 **AsyncTask** was introduced in Android to perform background operations and update the UI without blocking the **Main Thread**.

🔹 It follows a **threading model** where a task runs in the background and updates the UI when completed.

🔹 However, it was **deprecated in API 30 (Android 11)** due to **memory leaks, inefficiency, and poor scalability**.

✅ **Key Features of AsyncTask:**

✓ Allows executing tasks in **background threads**.

✓ Updates the **UI thread** when the task is finished.

✓ Uses **doInBackground()** for background work and **onPostExecute()** for UI updates.

📌 **Example: Implementing AsyncTask to Fetch Data**

```
class MyAsyncTask(private val context: Context) : AsyncTask<Void, Void, String>() {

    override fun doInBackground(vararg params: Void?): String {
        Thread.sleep(3000) // Simulating network call
        return "Data Loaded"
    }
```

```
    override fun onPostExecute(result: String?) {
        Toast.makeText(context, result, Toast.LENGTH_SHORT).show()
    }
}

// Execute AsyncTask
MyAsyncTask(context).execute()
```

📌 **How it Works?**

1️⃣ **doInBackground()** – Runs on a background thread (fetching data).

2️⃣ **onPostExecute()** – Runs on the main thread (updating the UI).

2.2 Problems with AsyncTask

🔴 **Memory Leaks** – If the activity is destroyed before AsyncTask completes, it can cause a memory leak.

🔴 **Cannot Handle Lifecycle Properly** – Tasks continue running even if the user navigates away.

🔴 **Difficult to Chain Tasks** – AsyncTask does not support structured concurrency.

🔴 **Deprecated** – Google recommends using Coroutines for background tasks.

CHAPTER 3: UNDERSTANDING COROUTINES

3.1 What are Coroutines?

🔹 **Coroutines are a modern alternative to AsyncTask for background processing.**

🔹 They provide an easy way to write **asynchronous and non-blocking code**.

🔹 **Lightweight Threads** – Unlike traditional Java threads, coroutines **don't block** the main thread.

✅ **Key Features of Coroutines:**

✓ **Lightweight** – Uses fewer resources than traditional threads.

✓ **Structured Concurrency** – Automatically cancels background tasks when the lifecycle ends.

✓ **Easy to Use** – Provides simple functions (launch, async, await) for managing tasks.

✓ **Improved Performance** – Optimized for large-scale applications.

3.2 Setting Up Coroutines in Android

✅ **Step 1: Add Coroutine Dependencies in build.gradle**

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.4"
}
```

✅ **Step 2: Use Coroutine in ViewModel or Activity**

```
import kotlinx.coroutines.*

fun fetchData() {
```

```
    GlobalScope.launch(Dispatchers.IO) {
        val data = loadFromNetwork()
        withContext(Dispatchers.Main) {
            Toast.makeText(context, data,
Toast.LENGTH_SHORT).show()
        }
    }
}

suspend fun loadFromNetwork(): String {
    delay(3000) // Simulating network call
    return "Data Loaded"
}
```

📌 **How it Works?**

1. **launch(Dispatchers.IO)** – Runs in the background thread (for network calls).
2. **withContext(Dispatchers.Main)** – Switches back to the main thread to update the UI.

3.3 Types of Coroutine Dispatchers

- **Dispatchers control where the coroutine runs.**

Dispatcher	Description	Use Case
Dispatchers.Main	Runs on the main thread.	UI updates, button clicks.

Dispatchers.IO	Runs on a background thread.	Network calls, database queries.
Dispatchers.Default	Runs on a background thread optimized for CPU work.	Heavy computations, sorting data.
Dispatchers.Unconfined	Runs on the current thread.	Testing and debugging.

📌 **Example: Using Different Dispatchers**

GlobalScope.launch(Dispatchers.IO) { fetchDataFromApi() } // Runs in background

GlobalScope.launch(Dispatchers.Main) { updateUI() } // Runs on main thread

---

CHAPTER 4: ASYNCTASK VS COROUTINES – KEY DIFFERENCES

Feature	AsyncTask	Coroutines
Thread Management	Creates new threads each time	Uses lightweight coroutines
Memory Usage	Can cause memory leaks	More efficient with structured concurrency
Lifecycle Awareness	Continues running after activity is destroyed	Automatically cancels when lifecycle ends

Complexity	Requires multiple methods (doInBackground, onPostExecute)	Uses simple functions (launch, async, await)
Performance	Slower due to Java threads overhead	Faster with optimized thread handling
Recommended By Google?	❌ Deprecated	✅ Recommended

📌 **Example:**

- **Using AsyncTask** for network calls **may cause memory leaks**.
- **Using Coroutines** ensures better **thread management and automatic cancellation**.

CHAPTER 5: REPLACING ASYNCTASK WITH COROUTINES

🔹 **Convert this AsyncTask Code:**

```
class MyAsyncTask(private val context: Context) : AsyncTask<Void, Void, String>() {
    override fun doInBackground(vararg params: Void?): String {
        Thread.sleep(3000)
        return "Data Loaded"
    }
    override fun onPostExecute(result: String?) {
        Toast.makeText(context, result, Toast.LENGTH_SHORT).show()
    }
```

```
}
```

- **Replace It with Coroutines:**

```
fun fetchData() {
   GlobalScope.launch(Dispatchers.IO) {
      val data = loadFromNetwork()
      withContext(Dispatchers.Main) {
         Toast.makeText(context, data,
Toast.LENGTH_SHORT).show()
      }
   }
}
suspend fun loadFromNetwork(): String {
   delay(3000) // Simulating network call
   return "Data Loaded"
}
```

📌 **Benefits of Using Coroutines Over AsyncTask:**

✓ **No memory leaks** – Coroutines are lifecycle-aware.
✓ **Easier to read** – No need for multiple callback functions.
✓ **Faster execution** – Optimized thread management.

Exercise: Test Your Understanding

- **Why was AsyncTask deprecated?**
- **What is structured concurrency in Coroutines?**
- **How does Dispatchers.IO differ from Dispatchers.Main?**

- **Convert an existing AsyncTask implementation to Coroutines.**
- **What are the advantages of using Coroutines over traditional threading?**

Conclusion

✅ **AsyncTask was once a solution for background tasks, but it had issues with memory leaks and poor performance.**
✅ **Coroutines provide a modern, efficient, and lifecycle-aware alternative.**
✅ **Coroutines are lightweight, easy to use, and improve app performance.**
✅ **Google officially recommends using Coroutines for background tasks in Android development.**

WorkManager & JobScheduler in Android

Chapter 1: Introduction to Background Processing in Android

1.1 Why Do We Need Background Processing?

- In Android, **some tasks need to run in the background,** such as:
  - Uploading data to a server.
  - Syncing user data periodically.
  - Sending notifications at scheduled times.
  - Performing long-running operations (e.g., downloading large files).
- **Background tasks require efficient management** to ensure:

✅ **Battery efficiency** – No unnecessary power consumption.
✅ **Reliability** – Tasks continue even if the app is closed.
✅ **Doze Mode & Background Restrictions Compliance** – Tasks should run effectively despite system optimizations.

📌 **Example:**

- **A news app periodically fetches new articles using background processing.**

Chapter 2: Understanding WorkManager in Android

2.1 What is WorkManager?

🔹 **WorkManager** is **Android's recommended API for background work** that needs to:

✓ **Run even after the app is closed or device restarts**.

✓ **Be deferrable (scheduled to run at the best time)**.

✓ **Be guaranteed to execute, even under Doze Mode & App Standby restrictions**.

✅ **Key Features of WorkManager:**

✓ Supports **one-time and periodic background tasks**.

✓ Respects **battery optimizations** like Doze Mode.

✓ **Ensures guaranteed execution,** even if the app is restarted.

✓ Supports **chained tasks** (run task A, then task B).

📌 **Example Use Case:**

- **An e-commerce app uploads customer reviews in the background using WorkManager.**

2.2 Implementing WorkManager in Android

✅ **Step 1: Add WorkManager Dependency**

📌 **Add to build.gradle (Module: app)**

```
dependencies {
    implementation "androidx.work:work-runtime:2.7.1"
}
```

✅ **Step 2: Create a Worker Class**

📌 **Create UploadWorker.kt**

```
import android.content.Context
import android.util.Log
```

```
import androidx.work.Worker
import androidx.work.WorkerParameters

class UploadWorker(context: Context, workerParams:
WorkerParameters) : Worker(context, workerParams) {
   override fun doWork(): Result {
      Log.d("WorkManager", "Uploading Data in Background...")

      // Simulating work being done
      Thread.sleep(3000)

      Log.d("WorkManager", "Upload Completed!")
      return Result.success()
   }
}
```

✅ **Step 3: Enqueue WorkManager Task in MainActivity.kt**

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.work.OneTimeWorkRequest
import androidx.work.WorkManager

class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)


        val workRequest =
OneTimeWorkRequest.Builder(UploadWorker::class.java).build()

        WorkManager.getInstance(this).enqueue(workRequest)
    }
}
```

📌 **Example:**

- **When the app starts, WorkManager runs UploadWorker in the background, logging messages to Logcat.**

CHAPTER 3: SCHEDULING PERIODIC TASKS WITH WORKMANAGER

3.1 Running a Task at Regular Intervals

🔹 **Use PeriodicWorkRequest to run background tasks repeatedly**.

🔹 The minimum repeat interval is **15 minutes** due to Android's battery restrictions.

📌 **Schedule a Periodic Task (Every 15 Minutes)**

```
import androidx.work.PeriodicWorkRequest

import java.util.concurrent.TimeUnit
```

```
val periodicWorkRequest =
PeriodicWorkRequest.Builder(UploadWorker::class.java, 15,
TimeUnit.MINUTES)
   .build()

WorkManager.getInstance(this).enqueue(periodicWorkRequest)
```

📌 **Example:**

- **A weather app fetches weather updates every 30 minutes using WorkManager.**

CHAPTER 4: USING JOBSCHEDULER FOR BACKGROUND TASKS

4.1 What is JobScheduler?

🔹 **JobScheduler** is an Android API for scheduling background tasks, introduced in **Android 5.0 (API 21)**.

🔹 It is useful when tasks **don't need immediate execution** but should run under specific conditions, such as:

✓ **Only run when the device is charging**.

✓ **Run when connected to Wi-Fi**.

✓ **Execute when the device is idle**.

✅ **Key Features of JobScheduler:**

✓ Allows **scheduled execution of jobs**.

✓ Supports **batching tasks to save battery**.

✓ Ensures **jobs run even after device reboot**.

✓ Can be **used without Google Play Services** (unlike WorkManager).

📌 **Example Use Case:**

- **A backup app uploads user data only when the device is charging and connected to Wi-Fi.**

4.2 Implementing JobScheduler in Android

✅ Step 1: Create a JobService

📌 Create MyJobService.kt

```
import android.app.job.JobParameters
import android.app.job.JobService
import android.util.Log

class MyJobService : JobService() {
    override fun onStartJob(params: JobParameters?): Boolean {
        Log.d("JobScheduler", "Job Started...")

        Thread {
            Thread.sleep(5000) // Simulate work being done
            Log.d("JobScheduler", "Job Completed!")
            jobFinished(params, false)
        }.start()

        return true // Job runs on a separate thread
    }
```

```
    override fun onStopJob(params: JobParameters?): Boolean {
        Log.d("JobScheduler", "Job Cancelled!")
        return true // Retry job if it was interrupted
    }
}
```

✅ **Step 2: Schedule a Job in MainActivity.kt**

```
import android.app.job.JobInfo
import android.app.job.JobScheduler
import android.content.ComponentName
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val jobScheduler =
getSystemService(JOB_SCHEDULER_SERVICE) as JobScheduler

        val jobInfo = JobInfo.Builder(1, ComponentName(this,
MyJobService::class.java))
            .setRequiresCharging(true)  // Only run when charging
```

```
.setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED
) // Only run on Wi-Fi
        .setPersisted(true) // Runs even after device reboot
        .build()

    jobScheduler.schedule(jobInfo)
  }
}
```

📌 **Example:**

- **The job will only run when the device is charging and connected to Wi-Fi.**

CHAPTER 5: WORKMANAGER VS JOBSCHEDULER – KEY DIFFERENCES

Feature	**WorkManager**	**JobScheduler**
Minimum API Level	Works on **API 14+**	Works on **API 21+**
Task Types	Short & long-running tasks	Scheduled jobs
Battery Efficiency	Uses background restrictions	Optimized for battery
Reliability	Runs even after device reboot	Runs after reboot if setPersisted(true)

Best Use Cases	Syncing data, sending notifications	Large uploads, background downloads

📌 **Example:**

- **Use WorkManager to sync chat messages.**
- **Use JobScheduler to back up user files when on Wi-Fi.**

CHAPTER 6: BEST PRACTICES FOR BACKGROUND PROCESSING

✅ **Use WorkManager for most background tasks (recommended by Google).**
✅ **Use JobScheduler when scheduling tasks that require specific conditions (e.g., charging, Wi-Fi).**
✅ **Use AlarmManager for exact time-based tasks (e.g., calendar reminders).**
✅ **Use Coroutines for lightweight background tasks.**

📌 **Example:**

- **A stock market app updates stock prices using WorkManager every hour.**

📌 **Exercise: Test Your Understanding**

🔹 **What is the difference between WorkManager and JobScheduler?**
🔹 **When should you use PeriodicWorkRequest in WorkManager?**
🔹 **How do you ensure a job runs only when the device is charging?**
🔹 **Modify JobScheduler to run only when the device is idle.**
🔹 **Implement a WorkManager task to send daily notifications.**

📌 Conclusion

✅ **WorkManager is the recommended API for scheduling background tasks.**
✅ **JobScheduler is useful for large background jobs that depend on system conditions.**
✅ **Using the right background processing API ensures efficient battery usage and performance.**

INTRODUCTION TO DEPENDENCY INJECTION IN ANDROID DEVELOPMENT

CHAPTER 1: UNDERSTANDING DEPENDENCY INJECTION (DI)

1.1 What is Dependency Injection?

🔹 **Dependency Injection (DI)** is a **design pattern** used in software development where one object supplies the dependencies of another object.

🔹 Instead of creating dependencies inside a class, DI allows injecting them from the outside.

✅ **Why Use Dependency Injection?**

✓ Improves code reusability and modularity.

✓ Makes code easier to test (no hardcoded dependencies).

✓ Simplifies dependency management in large applications.

✓ Enhances maintainability by reducing tight coupling.

📌 **Example:**

- **A logging system where different loggers (file, console, network) can be injected into a class instead of being hardcoded.**

1.2 Key Concepts in Dependency Injection

🔹 **Dependency**: An object required by another object to function.

🔹 **Injection**: Supplying the required dependencies from an external source.

🔹 **Loose Coupling**: Reducing dependencies between components, making the system more flexible.

🔹 **IoC (Inversion of Control)**: The control of creating dependencies is moved from the class itself to an external source.

📌 **Example of Loose Coupling:**

- Instead of this tightly coupled class:

```
class Car {
  private val engine = Engine() // Hardcoded dependency
}
```

- Use dependency injection to pass an engine:

```
class Car(private val engine: Engine) // Dependency is injected
```

CHAPTER 2: TYPES OF DEPENDENCY INJECTION

2.1 Constructor Injection

🔹 The most common DI method where dependencies are passed through the constructor.

📌 **Example:**

```
class Engine

class Car(private val engine: Engine) {
  fun start() {
    println("Car started with engine: $engine")
  }
}
```

✅ **Benefits:**

✓ Easy to identify dependencies.

✓ Works well with testing and mocking frameworks.

2.2 Method Injection

🔹 The dependency is passed via a setter or method call.

📌 **Example:**

```
class Car {
    private var engine: Engine? = null

    fun setEngine(engine: Engine) {
        this.engine = engine
    }
}
```

✅ **Useful when a dependency needs to change at runtime.**

2.3 Field Injection

🔹 Dependency is injected directly into class fields (common in DI frameworks like Dagger & Hilt).

📌 **Example (Using Dagger):**

```
class Car {
    @Inject lateinit var engine: Engine
}
```

✅ **Simplifies dependency management but makes testing harder.**

Chapter 3: Dependency Injection in Android

3.1 Why Use Dependency Injection in Android?

🔹 Android apps often require multiple dependencies such as:

✓ Network clients (Retrofit, OkHttp).

✓ Database instances (Room, Firebase).

✓ Shared preferences, repositories, view models.

✅ **Without DI:**

- Each activity or fragment manually creates dependencies, leading to **tight coupling**.

✅ **With DI:**

- Dependencies are managed centrally and injected as needed, improving flexibility.

📌 **Example Without DI:**

```
class MainActivity : AppCompatActivity() {
  private val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .build()
}
```

📌 **Example With DI (Using Hilt):**

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
```

```
    @Inject lateinit var retrofit: Retrofit
}
```

✅ **This makes the class more testable and maintainable.**

Chapter 4: Dependency Injection Frameworks in Android

4.1 Popular DI Frameworks in Android

Framework	Description	Best For
Dagger 2	A fast and powerful DI framework by Google.	Large, performance-critical apps.
Hilt	A simplified DI framework built on Dagger.	Recommended for most Android apps.
Koin	A lightweight DI framework for Kotlin.	Smaller, simpler projects.

📌 **Example:**

- **Hilt is preferred in Android development due to its simplicity and integration with Jetpack.**

Chapter 5: Implementing Dependency Injection Using Hilt

5.1 What is Hilt?

🔹 **Hilt** is a **dependency injection library for Android**, built on top of Dagger.

🔹 It simplifies **injecting dependencies into Activities, Fragments, and ViewModels**.

✅ **Step 1: Add Dependencies in build.gradle**

```
dependencies {
    implementation "com.google.dagger:hilt-android:2.38.1"
    kapt "com.google.dagger:hilt-compiler:2.38.1"
}
```

✅ **Step 2: Initialize Hilt in the Application Class**

```
@HiltAndroidApp
class MyApplication : Application()
```

✅ **Step 3: Create a Module to Provide Dependencies**

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://api.example.com")
            .build()
    }
}
```

✅ **Step 4: Inject Dependencies in an Activity**

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
```

```
    @Inject lateinit var retrofit: Retrofit
}
```

📌 **Example:**

- **Retrofit is now automatically injected into MainActivity without manual initialization.**

Chapter 6: Benefits of Using Dependency Injection

✅ **Improves Code Maintainability**

✓ Makes it easier to change implementations without modifying dependent classes.

✅ **Enhances Testability**

✓ Allows for easy mocking and unit testing.

✅ **Supports Modular Architecture**

✓ DI enables breaking down applications into **smaller, independent modules**.

✅ **Reduces Boilerplate Code**

✓ DI frameworks like Hilt **remove the need for manual object creation**.

📌 **Example:**

- A **chat app** using DI can easily switch from Firebase to another backend service without changing code in multiple places.

Exercise: Test Your Understanding

- **What is dependency injection and why is it used?**
- **What are the differences between Constructor, Method, and**

Field Injection?

- **Implement a DI setup to inject a database instance in an Android app.**
- **What is the role of Hilt in dependency injection?**
- **How does DI improve unit testing in Android apps?**

Conclusion

✅ **Dependency Injection simplifies object creation and management in Android applications.**

✅ **Using DI reduces tight coupling and improves modularity.**

✅ **Hilt is the recommended DI framework for Android due to its simplicity and integration with Jetpack.**

✅ **Understanding DI is essential for building scalable and maintainable Android apps.**

IMPLEMENTING DAGGER & HILT IN AN ANDROID PROJECT

CHAPTER 1: INTRODUCTION TO DEPENDENCY INJECTION (DI)

1.1 What is Dependency Injection (DI)?

🔹 **Dependency Injection (DI)** is a design pattern that helps in **managing dependencies efficiently** by providing required objects rather than creating them manually.

🔹 DI makes the code **more maintainable, scalable, and testable**.

✅ **Why Use Dependency Injection?**

✓ **Reduces boilerplate code** by automating object creation.

✓ **Improves testability** by allowing easy dependency mocking.

✓ **Enhances modularity** by decoupling class dependencies.

📌 **Example:**

- Instead of manually creating an instance of a UserRepository in multiple places, DI injects it where needed.

CHAPTER 2: WHAT IS DAGGER AND HILT?

2.1 What is Dagger?

🔹 **Dagger** is a fully static **dependency injection framework** for Android.

🔹 It generates **optimized and efficient code** at compile time.

✅ **Features of Dagger:**

✓ **Compile-time dependency injection** (faster execution).

✓ **Provides annotations (@Inject, @Component, @Module)** for

DI.

✓ **Manages object lifecycle efficiently.**

📌 **Example:**

- A DatabaseHelper instance can be injected automatically without manual instantiation.

2.2 What is Hilt?

🔹 **Hilt** is a **simplified DI framework** built on top of Dagger for Android.

🔹 It is recommended by **Google** as the standard DI solution for Android projects.

✅ **Why Use Hilt Over Dagger?**

✓ **Easier setup** with fewer boilerplate codes.

✓ **Automatically integrates with Android components.**

✓ **Uses pre-defined component lifecycles (ApplicationComponent, ActivityComponent, etc.).**

📌 **Example:**

- Hilt allows injecting a UserRepository into a ViewModel without extra setup.

CHAPTER 3: SETTING UP HILT IN AN ANDROID PROJECT

3.1 Adding Hilt Dependencies in build.gradle

1️⃣ Open **build.gradle (Project Level)** and add:

classpath "com.google.dagger:hilt-android-gradle-plugin:2.40"

2️⃣ Open **build.gradle (Module Level)** and add:

```
plugins {
    id 'com.android.application'
    id 'kotlin-kapt'
    id 'dagger.hilt.android.plugin'
}

dependencies {
    implementation 'com.google.dagger:hilt-android:2.40'
    kapt 'com.google.dagger:hilt-compiler:2.40'
}
```

3. **Sync the project** after adding dependencies.

3.2 Enabling Hilt in the Application Class

🔹 **Annotate the Application class with @HiltAndroidApp** to enable Hilt.

📌 **Example:**

```
@HiltAndroidApp
public class MyApplication extends Application { }
```

📌 **Explanation:**

- This makes Hilt available throughout the entire application.

CHAPTER 4: IMPLEMENTING DEPENDENCY INJECTION WITH HILT

4.1 Injecting a Repository into ViewModel Using Hilt

- We will **inject UserRepository into UserViewModel** using Hilt.

✅ Step 1: Create a Repository Class (UserRepository.java)

```
import javax.inject.Inject;


public class UserRepository {
   @Inject
   public UserRepository() { }


   public String getUserData() {
      return "User Data from Repository";
   }
}
```

📌 Explanation:

- @Inject in constructor → Hilt knows how to provide UserRepository.

✅ Step 2: Create a ViewModel with Hilt (UserViewModel.java)

```
import androidx.lifecycle.ViewModel;
import javax.inject.Inject;


public class UserViewModel extends ViewModel {
```

```
    private final UserRepository userRepository;

    @Inject
    public UserViewModel(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String getUser() {
        return userRepository.getUserData();
    }
}
```

📌 **Explanation:**

- @Inject in constructor ⟶ Injects UserRepository into UserViewModel.

✅ **Step 3: Provide Dependencies Using Hilt Module (AppModule.java)**

```
import dagger.Module;
import dagger.Provides;
import dagger.hilt.InstallIn;
import dagger.hilt.components.SingletonComponent;
import javax.inject.Singleton;
```

```
@Module
@InstallIn(SingletonComponent.class)
public class AppModule {
   @Singleton
   @Provides
   public static UserRepository provideUserRepository() {
      return new UserRepository();
   }
}
```

📌 **Explanation:**

- @Module → Defines a **Hilt module** to provide dependencies.
- @Provides → Hilt knows how to create UserRepository.
- @Singleton → Ensures **only one instance** exists.

✅ **Step 4: Inject ViewModel in an Activity (MainActivity.java)**

```
import android.os.Bundle;
import android.widget.TextView;
import androidx.activity.viewModels;
import androidx.appcompat.app.AppCompatActivity;
import dagger.hilt.android.AndroidEntryPoint;

@AndroidEntryPoint
```

```
public class MainActivity extends AppCompatActivity {

  private final UserViewModel userViewModel = new
UserViewModel();


  @Override

  protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);


    TextView textView = findViewById(R.id.textView);

    textView.setText(userViewModel.getUser());

  }

}
```

📌 **Explanation:**

- @AndroidEntryPoint → Enables Hilt in MainActivity.
- userViewModel.getUser() fetches data from UserRepository.

CHAPTER 5: ADVANCED HILT FEATURES

5.1 Using Hilt with Retrofit for API Calls

✅ **Step 1: Add Retrofit Dependency**

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

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

✅ **Step 2: Create an API Interface**

```
import retrofit2.Call;
import retrofit2.http.GET;

public interface ApiService {
   @GET("users")
   Call<List<User>> getUsers();
}
```

✅ **Step 3: Provide Retrofit Instance Using Hilt Module**

```
@Module
@InstallIn(SingletonComponent.class)
public class NetworkModule {
   @Singleton
   @Provides
   public static Retrofit provideRetrofit() {
      return new Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")
            .addConverterFactory(GsonConverterFactory.create())
            .build();
   }

   @Singleton
   @Provides
```

```
    public static ApiService provideApiService(Retrofit retrofit) {
        return retrofit.create(ApiService.class);
    }
}
```

📌 **Explanation:**

- provideRetrofit() ⟶ Provides a Retrofit instance.
- provideApiService() ⟶ Creates an ApiService instance.

5.2 Injecting ViewModel in Activity Using Hilt

✅ **Modify ViewModel to Fetch API Data**

```
import androidx.lifecycle.ViewModel;
import javax.inject.Inject;

public class ApiViewModel extends ViewModel {
    private final ApiService apiService;

    @Inject
    public ApiViewModel(ApiService apiService) {
        this.apiService = apiService;
    }

    public Call<List<User>> fetchUsers() {
```

```
    return apiService.getUsers();
  }
}
```

✅ Inject ViewModel in Activity

```
@AndroidEntryPoint
public class MainActivity extends AppCompatActivity {
  private final ApiViewModel apiViewModel = new ApiViewModel();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    apiViewModel.fetchUsers().enqueue(new
Callback<List<User>>() {
      @Override
      public void onResponse(Call<List<User>> call,
Response<List<User>> response) {
        // Update UI with API data
      }

      @Override
      public void onFailure(Call<List<User>> call, Throwable t) {
```

```
            // Handle failure
        }
    });
  }
}
```

📌 **Example:**

- **A social media app injects Retrofit into ViewModel to fetch user profiles.**

Conclusion

✅ **Hilt simplifies Dependency Injection in Android.**
✅ **It reduces boilerplate code and improves testability.**
✅ **Hilt works seamlessly with ViewModel, Retrofit, and other components.**

FETCHING USER LOCATION IN ANDROID

CHAPTER 1: INTRODUCTION TO FETCHING USER LOCATION

1.1 What is User Location Tracking?

🔹 **Fetching user location** allows Android applications to access **GPS coordinates** (latitude and longitude) to provide location-based services.

🔹 **Google's Location API** offers accurate and efficient location tracking with lower battery consumption.

✅ **Why Fetch User Location?**

✓ **Navigation & Maps** – Google Maps, Uber, and delivery apps.

✓ **Personalized Services** – Weather updates, localized ads, and recommendations.

✓ **Safety & Emergency Features** – SOS alerts and location sharing.

✓ **Fitness & Tracking** – Step counters, jogging routes, and geo-tagging.

📌 **Example Use Cases:**

- A **food delivery app** gets the user's location for accurate deliveries.
- A **weather app** fetches the user's location to show local weather updates.

CHAPTER 2: SETTING UP LOCATION PERMISSIONS

🔹 **Before fetching the user's location, we need to request permissions.**

🔹 **Starting from Android 10 (API level 29), location access is restricted.**

✅ 2.1 Add Required Permissions in AndroidManifest.xml

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

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

✅ 2.2 Request Permissions in Kotlin (Android 10+)

```
if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
  != PackageManager.PERMISSION_GRANTED) {
  ActivityCompat.requestPermissions(
    this, arrayOf(Manifest.permission.ACCESS_FINE_LOCATION), 1
  )
}
```

✅ 2.3 Handle Permission Result in onRequestPermissionsResult()

```
override fun onRequestPermissionsResult(requestCode: Int,
permissions: Array<out String>, grantResults: IntArray) {
  super.onRequestPermissionsResult(requestCode, permissions,
grantResults)
  if (requestCode == 1 && grantResults.isNotEmpty() &&
grantResults[0] == PackageManager.PERMISSION_GRANTED) {
    getUserLocation()
```

```
  } else {
    Toast.makeText(this, "Permission Denied",
Toast.LENGTH_SHORT).show()
  }
}
```

Chapter 3: Fetching Location Using Fused Location Provider

3.1 What is Fused Location Provider (FLP)?

- **Fused Location Provider (FLP)** is the recommended API to fetch user location in Android.
- It combines **GPS, Wi-Fi, and cellular networks** to provide accurate location with low battery consumption.

✅ **Key Benefits of Fused Location Provider:**
✓ More accurate than traditional GPS tracking.
✓ Consumes less battery by optimizing location requests.
✓ Automatically chooses the best location provider (GPS, Network, or Wi-Fi).

3.2 Implementing Fused Location Provider in Android

✅ **Step 1: Add Google Play Services Dependency in build.gradle**

```
dependencies {
  implementation 'com.google.android.gms:play-services-
location:21.0.1'
}
```

✅ Step 2: Initialize FusedLocationProviderClient

```
private lateinit var fusedLocationClient:
FusedLocationProviderClient


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


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

✅ Step 3: Get Last Known Location

```
private fun getUserLocation() {
  if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
    == PackageManager.PERMISSION_GRANTED) {


    fusedLocationClient.lastLocation.addOnSuccessListener {
location: Location? ->
      if (location != null) {
        val latitude = location.latitude
        val longitude = location.longitude
```

```
            Toast.makeText(this, "Lat: $latitude, Long: $longitude",
Toast.LENGTH_LONG).show()
        } else {
            Toast.makeText(this, "Location not available",
Toast.LENGTH_SHORT).show()
        }
    }
  }
}
```

📌 **How it Works?**

1. FusedLocationClient initializes the **Fused Location Provider**.
2. lastLocation.addOnSuccessListener retrieves the **most recent location**.
3. Displays the **latitude and longitude** using Toast.

CHAPTER 4: GETTING CONTINUOUS LOCATION UPDATES

🔹 **To track location changes in real time, use requestLocationUpdates().**

✅ **Step 1: Define Location Request Parameters**

```
private lateinit var locationRequest: LocationRequest
private lateinit var locationCallback: LocationCallback

private fun createLocationRequest() {
  locationRequest = LocationRequest.create().apply {
```

```
    interval = 5000 // Update every 5 seconds
    fastestInterval = 2000
    priority = LocationRequest.PRIORITY_HIGH_ACCURACY
  }
}
```

✅ **Step 2: Implement Location Updates Callback**

```
private fun startLocationUpdates() {
  locationCallback = object : LocationCallback() {
    override fun onLocationResult(locationResult: LocationResult) {
      for (location in locationResult.locations) {
        Log.d("Location Update", "Lat: ${location.latitude}, Long: ${location.longitude}")
      }
    }
  }

  if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION)
    == PackageManager.PERMISSION_GRANTED) {
    fusedLocationClient.requestLocationUpdates(locationRequest, locationCallback, Looper.getMainLooper())
  }
}
```

✅ Step 3: Stop Location Updates to Save Battery

```
override fun onPause() {
    super.onPause()
    fusedLocationClient.removeLocationUpdates(locationCallback)
}
```

📌 How it Works?

1. LocationRequest specifies **update intervals and accuracy mode**.
2. LocationCallback listens for **real-time location changes**.
3. requestLocationUpdates() fetches **continuous location data**.

CHAPTER 5: REVERSE GEOCODING (CONVERT COORDINATES TO ADDRESS)

🔹 **Geocoding converts GPS coordinates into a human-readable address.**

✅ Step 1: Add Geocoder to Fetch Address from Coordinates

```
private fun getAddressFromLocation(latitude: Double, longitude: Double) {
    val geocoder = Geocoder(this, Locale.getDefault())
    val addresses: List<Address>? = geocoder.getFromLocation(latitude, longitude, 1)
    if (!addresses.isNullOrEmpty()) {
        val address = addresses[0].getAddressLine(0)
        Toast.makeText(this, "Address: $address", Toast.LENGTH_LONG).show()
```

```
    }
}
```

✅ Step 2: Fetch Address After Getting Location

```
if (location != null) {
    getAddressFromLocation(location.latitude, location.longitude)
}
```

📌 Example Output:

Lat: 40.7128, Long: -74.0060

Address: 1600 Amphitheatre Parkway, Mountain View, CA

Exercise: Test Your Understanding

- **What is the difference between GPS and Fused Location Provider?**
- **Why do we need ACCESS_FINE_LOCATION permission?**
- **How does requestLocationUpdates() differ from getLastLocation()?**
- **Implement a location-based weather app using OpenWeather API.**
- **Modify the code to track location updates only when the user is moving.**

Conclusion

✅ **Fetching user location is essential for location-based apps like navigation, food delivery, and ride-sharing.**

✅ **Fused Location Provider (FLP) is the best way to get accurate**

location with low battery usage.

✅ **Continuous location tracking should be used cautiously to avoid excessive battery drain.**

✅ **Reverse geocoding helps convert latitude/longitude into a readable address.**

INTEGRATING GOOGLE MAPS API IN ANDROID

CHAPTER 1: INTRODUCTION TO GOOGLE MAPS API

1.1 What is Google Maps API?

🔹 The **Google Maps API** allows Android developers to integrate interactive maps, location services, and geospatial data into their apps.

🔹 It provides features like **real-time navigation, place search, geolocation, markers, directions, and geofencing**.

✅ **Why Use Google Maps API?**

✓ Provides **real-time location tracking**.

✓ Enables **custom map styles, markers, and overlays**.

✓ Supports **directions, route planning, and traffic updates**.

✓ Works with **Google Places API for location-based services**.

📌 **Example Use Case:**

- **A ride-hailing app like Uber uses Google Maps API to show real-time driver locations and estimated routes.**

CHAPTER 2: SETTING UP GOOGLE MAPS API IN AN ANDROID PROJECT

2.1 Create a Google Maps API Key

✅ **Step 1: Get API Key from Google Cloud Console**

1️⃣ Visit the Google Cloud Console.

2️⃣ Create a new project or select an existing one.

3️⃣ Go to **APIs & Services → Credentials**.

4. Click **"Create Credentials"** → **"API Key"**.
5. Enable the **Google Maps SDK for Android** and **Places API**.
6. Copy the generated **API Key**.

2.2 Add Google Maps Dependencies

📌 **Add the following dependency in build.gradle (Module: app)**

```
dependencies {
    implementation 'com.google.android.gms:play-services-maps:18.0.2'
    implementation 'com.google.android.gms:play-services-location:21.0.1' // For GPS tracking
}
```

✅ **Sync the project** after adding dependencies.

CHAPTER 3: DISPLAYING A GOOGLE MAP IN AN ANDROID APP

3.1 Add a Map Fragment to the Layout

📌 **Modify activity_maps.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<fragment
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/map"
```

```
android:name="com.google.android.gms.maps.SupportMapFragment"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MapsActivity"/>
```

3.2 Initialize Google Maps in MapsActivity.java

📌 **Create MapsActivity.java or MapsActivity.kt**

```
import androidx.fragment.app.FragmentActivity;
import android.os.Bundle;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.SupportMapFragment;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class MapsActivity extends FragmentActivity implements OnMapReadyCallback {

   private GoogleMap mMap;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_maps);

    // Initialize the map fragment
    SupportMapFragment mapFragment = (SupportMapFragment)
getSupportFragmentManager()
            .findFragmentById(R.id.map);
    if (mapFragment != null) {
        mapFragment.getMapAsync(this);
    }
}

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

    // Add a marker at a location and move the camera
    LatLng location = new LatLng(37.7749, -122.4194); // Example:
San Francisco
    mMap.addMarker(new
MarkerOptions().position(location).title("Marker in San Francisco"));
```

```
mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(location, 12)); // Zoom level: 12
  }
}
```

📌 **Example:**

- **The map displays San Francisco with a marker.**

CHAPTER 4: ADDING USER LOCATION & GPS TRACKING

4.1 Request Location Permissions

📌 **Add the following permissions in AndroidManifest.xml**

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

4.2 Enable My Location Layer in Google Maps

📌 **Modify onMapReady() in MapsActivity.java**

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

```
    if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) ==
PackageManager.PERMISSION_GRANTED) {

        mMap.setMyLocationEnabled(true); // Enable user location
tracking

    } else {

        ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 1);

    }

}
```

📌 **Example:**

- **The map now displays the user's current location as a blue dot.**

CHAPTER 5: ADDING MARKERS, CUSTOM ICONS, AND INFO WINDOWS

5.1 Adding Multiple Markers on the Map

```
LatLng location1 = new LatLng(40.7128, -74.0060); // New York

LatLng location2 = new LatLng(34.0522, -118.2437); // Los Angeles


mMap.addMarker(new
MarkerOptions().position(location1).title("New York"));

mMap.addMarker(new
MarkerOptions().position(location2).title("Los Angeles"));
```

5.2 Customizing Marker Icons

mMap.addMarker(new MarkerOptions()

.position(new LatLng(48.8566, 2.3522)) // Paris

.title("Custom Marker in Paris")

.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_BLUE))); // Blue Marker

📌 **Example:**

- **Adds a blue marker at Paris.**

CHAPTER 6: GETTING DIRECTIONS BETWEEN TWO LOCATIONS

6.1 Using Google Directions API

🔹 **Google Directions API** provides **turn-by-turn navigation and route planning** between two locations.

🔹 Requires an API Key with **Directions API enabled**.

✅ **Example API Request (Replace API_KEY with your actual key):**

https://maps.googleapis.com/maps/api/directions/json?origin=New+York,NY&destination=Los+Angeles,CA&key=API_KEY

📌 **Parse the response in Android**

JSONObject jsonResponse = new JSONObject(response);

JSONArray routes = jsonResponse.getJSONArray("routes");

JSONObject route = routes.getJSONObject(0);

JSONArray legs = route.getJSONArray("legs");

```
JSONObject leg = legs.getJSONObject(0);
String distance = leg.getJSONObject("distance").getString("text");
String duration = leg.getJSONObject("duration").getString("text");
```

📌 **Example:**

- **Retrieve distance and duration between two cities using the Directions API.**

CHAPTER 7: BEST PRACTICES FOR USING GOOGLE MAPS API

✅ **Use API keys securely** – Restrict access by **app package name** in Google Cloud Console.
✅ **Cache map data for offline use** if needed.
✅ **Optimize markers and overlays** to avoid lag on large datasets.
✅ **Use geofencing to trigger location-based notifications.**
✅ **Handle API quota limits** by caching responses when possible.

📌 **Example:**

- **A food delivery app uses geofencing to notify users when their order is near.**

📌 **Exercise: Test Your Understanding**

🔹 **How does Google Maps API help in Android development?**
🔹 **Write code to add three different markers on a map.**
🔹 **How do you request the user's location in Google Maps API?**
🔹 **Modify the map to track the user's movement in real-time.**
🔹 **Implement Google Directions API to display a route between two locations.**

📌 Conclusion

✅ Google Maps API allows seamless integration of maps and location services into Android apps.
✅ Developers can add markers, enable GPS tracking, and get directions between locations.
✅ Using best practices ensures optimal performance and security.

ASSIGNMENT:

BUILD A WEATHER APP WITH GEOLOCATION AND BACKGROUND API CALLS.

📌 Step-by-Step Assignment Solution: Build a Weather App with Geolocation and Background API Calls

---

📌 Step 1: Create a New Android Project

1. **Open Android Studio** and create a new project.
2. Choose **Empty Activity** and click **Next**.
3. Name the project **WeatherApp**.
4. Select **Kotlin** as the language and click **Finish**.

📌 Step 2: Add Required Dependencies

✅ **Modify build.gradle (Module: app) and add:**

```
dependencies {
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
    implementation 'com.google.android.gms:play-services-location:21.0.1'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.0'
}
```

✅ **Sync the project** to apply the changes.

📌 Step 3: Get API Key from OpenWeatherMap

- Go to OpenWeatherMap and sign up.
- Generate a **free API key** for accessing the weather data.
- Example API endpoint:

https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid=YOUR_API_KEY&units=metric

---

📌 Step 4: Request Location Permission in Android

✅ Modify AndroidManifest.xml to request GPS permissions:

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

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

✅ Declare location service inside <application> tag:

```
<uses-feature android:name="android.hardware.location.gps"/>
```

📌 Step 5: Create a Data Model for Weather API Response

📌 Create WeatherResponse.kt to store the weather data:

```
data class WeatherResponse(
   val name: String, // City Name
   val main: Main,
   val weather: List<Weather>
)
```

```
data class Main(
    val temp: Double // Temperature in Celsius
)

data class Weather(
    val description: String // Weather condition (e.g., "clear sky")
)
```

📌 Step 6: Set Up Retrofit for API Calls

✅ Create WeatherApiService.kt to define the API endpoint:

```
import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Query

interface WeatherApiService {
    @GET("weather")
    fun getWeather(
        @Query("lat") latitude: Double,
        @Query("lon") longitude: Double,
        @Query("appid") apiKey: String,
        @Query("units") units: String = "metric"
    ): Call<WeatherResponse>
```

```
}
```

✅ Create RetrofitClient.kt to initialize Retrofit:

```
import retrofit2.Retrofit

import retrofit2.converter.gson.GsonConverterFactory


object RetrofitClient {

   private const val BASE_URL =
"https://api.openweathermap.org/data/2.5/"


   val apiService: WeatherApiService by lazy {

      Retrofit.Builder()

         .baseUrl(BASE_URL)

         .addConverterFactory(GsonConverterFactory.create()) //
Convert JSON to Kotlin objects

         .build()

         .create(WeatherApiService::class.java)

   }
}
```

📌 Step 7: Fetch User's Location Using GPS

✅ Create LocationHelper.kt to get user location:

```
import android.annotation.SuppressLint

import android.content.Context
```

```
import android.location.Location
import
com.google.android.gms.location.FusedLocationProviderClient
import com.google.android.gms.location.LocationServices
import com.google.android.gms.tasks.Task

class LocationHelper(context: Context) {
    private var fusedLocationClient: FusedLocationProviderClient =
        LocationServices.getFusedLocationProviderClient(context)

    @SuppressLint("MissingPermission")
    fun getLastLocation(onSuccess: (Location?) -> Unit) {
        fusedLocationClient.lastLocation
            .addOnSuccessListener { location: Location? ->
                onSuccess(location)
            }
    }
}
```

✅ **Modify MainActivity.kt to get location and call the weather API:**

```
import android.Manifest
import android.content.pm.PackageManager
import android.location.Location
```

```
import android.os.Bundle
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.core.app.ActivityCompat
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response

class MainActivity : AppCompatActivity() {
    private val PERMISSION_REQUEST_CODE = 100
    private lateinit var locationHelper: LocationHelper

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

        val cityTextView = findViewById<TextView>(R.id.cityTextView)
        val tempTextView =
findViewById<TextView>(R.id.tempTextView)
        val weatherDescTextView =
findViewById<TextView>(R.id.weatherDescTextView)
```

```
        locationHelper = LocationHelper(this)


        // Check location permission
        if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
PERMISSION_REQUEST_CODE)
            return
        }


        // Get location and fetch weather
        locationHelper.getLastLocation { location ->
            if (location != null) {
                fetchWeather(location, cityTextView, tempTextView,
weatherDescTextView)
            } else {
                Toast.makeText(this, "Failed to get location",
Toast.LENGTH_SHORT).show()
            }
        }
    }
```

```
private fun fetchWeather(location: Location, cityTextView: TextView, tempTextView: TextView, weatherDescTextView: TextView) {

    val apiKey = "YOUR_API_KEY" // Replace with your OpenWeatherMap API key


    RetrofitClient.apiService.getWeather(location.latitude, location.longitude, apiKey)

        .enqueue(object : Callback<WeatherResponse> {

            override fun onResponse(call: Call<WeatherResponse>, response: Response<WeatherResponse>) {

                if (response.isSuccessful) {

                    val weatherResponse = response.body()

                    weatherResponse?.let {

                        cityTextView.text = "City: ${it.name}"

                        tempTextView.text = "Temperature: ${it.main.temp}°C"

                        weatherDescTextView.text = "Weather: ${it.weather[0].description}"

                    }

                }

            }


            override fun onFailure(call: Call<WeatherResponse>, t: Throwable) {
```

```
            Toast.makeText(applicationContext, "Failed to fetch
weather", Toast.LENGTH_SHORT).show()
        }
    })
  }
}
```

📌 Step 8: Design the Weather UI

✅ Modify activity_main.xml to display weather information:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center"
  android:padding="16dp">

  <TextView
    android:id="@+id/cityTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="City: --"
    android:textSize="20sp"/>
```

```
<TextView
    android:id="@+id/tempTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Temperature: --"
    android:textSize="20sp"/>

<TextView
    android:id="@+id/weatherDescTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Weather: --"
    android:textSize="20sp"/>
</LinearLayout>
```

📌 Step 9: Run and Test the Weather App

1. Run the app on a real Android device (GPS required).
2. Accept location permissions when prompted.
3. Check if **city, temperature, and weather conditions** are displayed correctly.

📌 Conclusion

- ✅ **Fetched user's location using GPS.**
- ✅ **Made API requests to OpenWeatherMap using Retrofit.**
- ✅ **Displayed weather data dynamically in the UI.**