Android Developer Notes

Foreground VS Background Services

👉 Use **Foreground Service** when:

- The task is long-running **and** user must be aware of it.
- Eg. music apps, navigation apps, fitness apps.

👉 Use **Background Service** when:

- The task is **short-lived** and user doesn't need to know.
- On modern Android → better to use **WorkManager** or **JobScheduler**.

👉 Golden Rule:

- **User-aware ongoing work** → Foreground Service.
- **Silent, scheduled, deferrable work** → WorkManager / JobScheduler.
- **Quick async work (while app active)** → Background Service.
-

- Before Android 8.0 (Oreo), you could run long-lived background services easily.

- After Oreo → **restrictions**:

  - Apps in the background cannot freely start background services.
  - If they try, the system throws an **IllegalStateException**.

1. Foreground Service

A **Foreground Service** performs tasks that the **user is actively aware of** and usually interacts with indirectly.

- Example: Playing music, recording location during navigation, fitness tracking.

Key points:

- Must display a **persistent notification** (cannot be swiped away).
- Runs even if the app is killed (until stopped explicitly).
- Android considers it **less likely to be killed** due to memory pressure.
- Requires `startForeground()` within 5 seconds of starting the service.
- Needs `FOREGROUND_SERVICE` **permission** in manifest.

When to use:

- Tasks that user should always be aware of.
- Media playback, tracking location, uploading large files, health monitoring

🔷 2. Background Service

A **Background Service** runs in the background without notifying the user.

- Example: Syncing data, fetching updates from server.

Key points:

- Runs silently without a persistent notification.
- Since **Android 8.0 (Oreo)** → background services are restricted:
  - Can't keep running indefinitely.
  - If you need long-running background work → use **WorkManager** or **JobScheduler**.
- If the app goes to background, background service might be killed by the system.

When to use:

- Quick tasks (under a few minutes).
- Periodic jobs → best done with **WorkManager**.

Feature	Foreground Service	Background Service
User Awareness	Visible (via notification)	Hidden
Survival	Less likely to be killed	Can be killed anytime
Use Case	Music, location, file upload	Sync, short tasks
Battery impact	Higher (always running)	Lower
Restrictions (Android O+)	Allowed with notification	Severely restricted
API / Alternatives	startForegroundService()	WorkManager / JobScheduler

Instead of a raw `BackgroundService`, use **WorkManager** because:

- It is **backward compatible**.
- It automatically decides how to run:
  - Uses **JobScheduler** (API 23+)
  - Uses **AlarmManager + BroadcastReceiver** on lower APIs
- Ensures the task will run, even after app restart.
- Perfect for tasks like syncing data, uploading logs, refreshing cache.

Foreground Service

kotlin Copy code

```kotlin
class MyForegroundService : Service() {
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        val notification = NotificationCompat.Builder(this, "channel_id")
            .setContentTitle("Foreground Service")
            .setContentText("Running...")
            .setSmallIcon(R.drawable.ic_notification)
            .build()

        startForeground(1, notification)

        // Do your task here...
        return START_NOT_STICKY
    }

    override fun onBind(intent: Intent?) = null
}
```

In **Manifest:**

xml Copy code

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

<service
    android:name=".MyForegroundService"
    android:foregroundServiceType="location|mediaPlayback"
    android:exported="false"/>
```

Background Service

kotlin Copy code

```kotlin
class MyBackgroundService : Service() {
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        Thread {
            // Do background task
            stopSelf()
        }.start()
        return START_NOT_STICKY
    }

    override fun onBind(intent: Intent?) = null
}
```

Workmanager

WorkManager is an **Android Jetpack library** used for scheduling and running **deferrable, asynchronous background tasks** that:

- Need **guaranteed execution** (even if the app is killed or the device restarts).
- Are **persistent** across app restarts.
- Respect **battery optimization, Doze mode, and API level restrictions**.

It's the **recommended solution** for most background work that doesn't need immediate execution (like Foreground Services).

When to Use WorkManager

You should use WorkManager for tasks such as:

- Uploading logs or analytics.
- Syncing data periodically.
- Backing up files to cloud storage.
- Sending deferred notifications.

If you need:

- **Immediate task execution** → use **Foreground Service**.
- **Exact timing (like alarms)** → use **AlarmManager**.
- **Long-running background work** → use **WorkManager with Foreground Service**.

Key Features

- **Guaranteed execution** → task will run even if app/process is killed.
- **Chaining tasks** → run tasks sequentially or in parallel.
- **Constraints** → run tasks only under conditions (e.g., device charging, Wi-Fi connected).
- **Backoff policy** → retries with exponential or linear delays.
- **Periodic tasks** → repeat work on schedule.
-
- 🚦 **WorkManager vs Other Options**

API	Best For
WorkManager	Guaranteed background work, deferrable, persistent
Foreground Service	Immediate, user-visible, long tasks (music, navigation)
AlarmManager	Exact alarms at specific times
JobScheduler	System-managed jobs (API 21+), WorkManager actually uses this under the hood

WorkManager → Jetpack library that provides **backward-compatibility**:

  - On API 23+ → internally uses **JobScheduler**.
  - On lower APIs → falls back to **AlarmManager + BroadcastReceiver**.

```kotlin
class UploadWorker(appContext: Context, params: WorkerParameters)
    : Worker(appContext, params) {

    override fun doWork(): Result {
        // Your background task here
        uploadLogsToServer()

        // Indicate success or failure
        return Result.success()
    }
}
```

Copy code

2. Schedule Work

kotlin

Copy code

```kotlin
val uploadWorkRequest = OneTimeWorkRequestBuilder<UploadWorker>()
    .setConstraints(
        Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .build()
    )
    .build()

WorkManager.getInstance(context).enqueue(uploadWorkRequest)
```

📂 Types of Work

1. **OneTimeWorkRequest** → run once (e.g., upload file).
2. **PeriodicWorkRequest** → run repeatedly (e.g., sync every 24 hrs).

kotlin

Copy code

```kotlin
val periodicWork = PeriodicWorkRequestBuilder<UploadWorker>(24, TimeUnit.HOURS).build
WorkManager.getInstance(context).enqueue(periodicWork)
```

🔄 Chaining Works

kotlin

Copy code

```kotlin
WorkManager.getInstance(context)
    .beginWith(workA)
    .then(workB)
    .then(workC)
    .enqueue()
```

📊 Work Status

You can observe the work status:

kotlin

Copy code

```kotlin
WorkManager.getInstance(context)
    .getWorkInfoByIdLiveData(uploadWorkRequest.id)
    .observe(this) { workInfo ->
        if (workInfo != null && workInfo.state == WorkInfo.State.SUCCEEDED) {
            Log.d("WorkManager", "Upload completed!")
        }
    }
```

Foreground Service + Work Manager

You're uploading a large file.

User must see **upload progress** → Foreground Service (with notification).
Upload must **continue in background**, even if the app is closed → WorkManager.

Architecture

WorkManager (Worker) → handles the actual long-running upload.
Foreground Service → shows progress notification (binds to the worker).

🔷 **How This Solves It**

WorkManager → ensures upload runs to completion, even if the app is killed.**Foreground Service** → ensures user is aware of the ongoing upload and sees progress.

- Service = UI awareness.
- WorkManager = reliable executionn

✅ **Interview Answer**:
"In real-world apps like file uploads, we often combine a Foreground Service and WorkManager. The Foreground Service displays a persistent notification so the user knows an upload is happening, while WorkManager actually performs the upload reliably in the background. This way, if the app is killed, the task continues, and the user is always aware of progress."

Worker for Upload (Background Work)

kotlin — Copy code

```kotlin
class UploadWorker(appContext: Context, params: WorkerParameters) :
    CoroutineWorker(appContext, params) {

    override suspend fun doWork(): Result {
        val filePath = inputData.getString("filePath") ?: return Result.failure()

        // Example: Upload simulation
        for (progress in 1..100) {
            setForeground(createForegroundInfo(progress))
            delay(100) // simulate upload chunk
        }

        return Result.success()
    }

    // Create Foreground Notification for progress
    private fun createForegroundInfo(progress: Int): ForegroundInfo {
        val notification = NotificationCompat.Builder(applicationContext, "upload_channel
            .setContentTitle("Uploading File")
            .setContentText("Progress: $progress%")
            .setSmallIcon(R.drawable.ic_upload)
            .setProgress(100, progress, false)
            .build()

        return ForegroundInfo(1, notification)
    }
}
```

Enqueue the Upload (From Activity/Service)

kotlin Copy code

```kotlin
val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()
    .setInputData(workDataOf("filePath" to "/path/to/file"))
    .build()

WorkManager.getInstance(context).enqueue(uploadRequest)
```

Foreground Service + WorkManager Sync

If you still want a **Foreground Service wrapper** (e.g. showing a notification immediately when user starts upload):

kotlin Copy code

```kotlin
class UploadService : Service() {
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        // Start persistent notification
        val notification = NotificationCompat.Builder(this, "upload_channel")
            .setContentTitle("Upload Started")
            .setContentText("Preparing upload...")
            .setSmallIcon(R.drawable.ic_upload)
            .build()

        startForeground(1, notification)

        // Delegate actual upload to WorkManager
        val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()
            .setInputData(workDataOf("filePath" to intent?.getStringExtra("filePath")))
            .build()

        WorkManager.getInstance(this).enqueue(uploadRequest)

        return START_NOT_STICKY
    }

    override fun onBind(intent: Intent?) = null
}
```

Broadcast Receiver

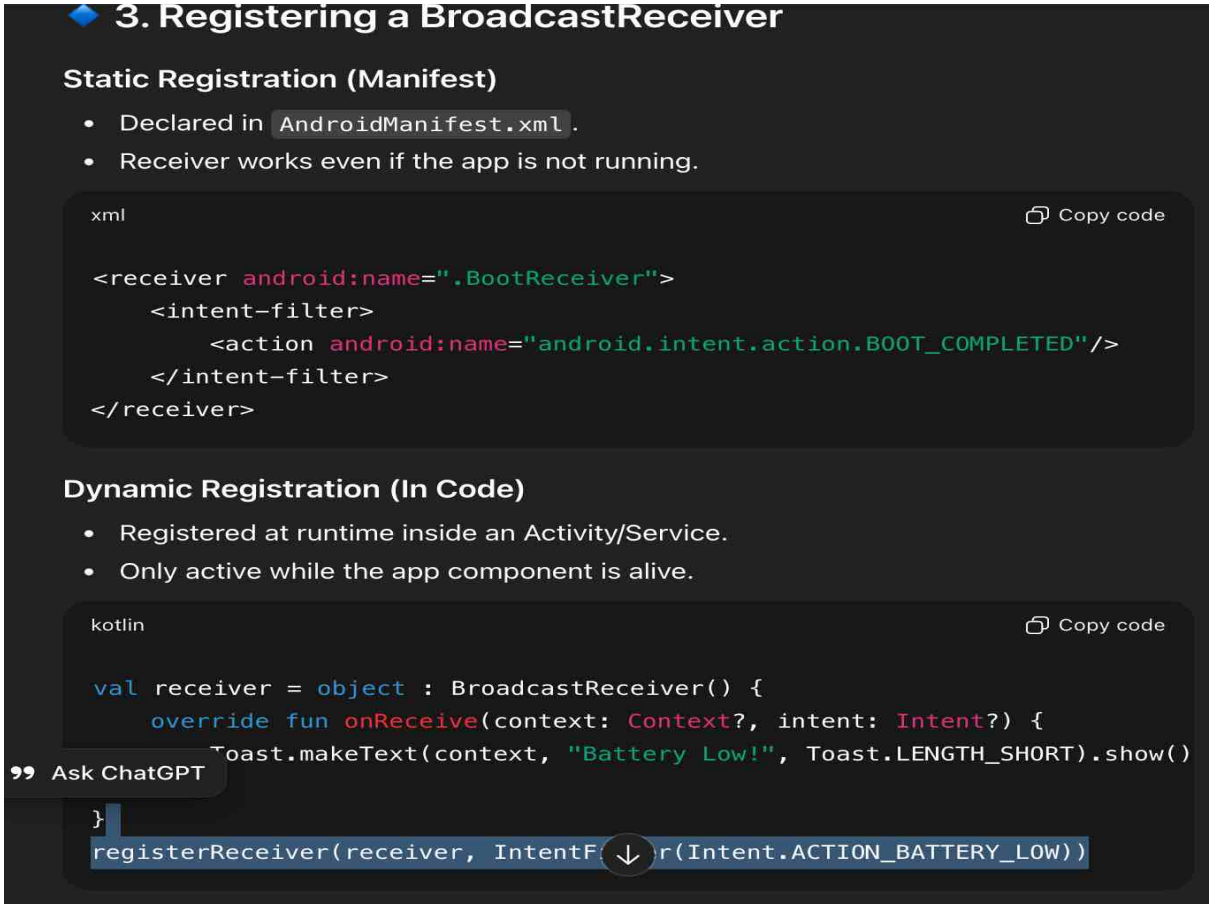
- A **BroadcastReceiver** in Android is a component that listens for **system-wide broadcast events** or **custom events** sent by apps.

- Example: Battery low, Wi-Fi connected, SMS received, or your app broadcasting "upload complete."

- It doesn't have a UI, but can **start activities, services, or show notification**

🔷 2. Types of Broadcasts

1. **System Broadcasts** → Sent by Android system.
    - Examples:
        - `android.intent.action.BOOT_COMPLETED`
        - `android.net.conn.CONNECTIVITY_CHANGE`
        - `android.intent.action.BATTERY_LOW`
2. **Custom Broadcasts** → Sent by your app.
    - Example: After file upload, broadcast an intent `com.myapp.UPLOAD_SUCCESS`.



🔷 4. When to Use BroadcastReceiver

👉 **Use BroadcastReceiver when:**

- Your app needs to react to **system-wide events**:
  - Boot completed → to reschedule jobs.
  - Network changes → to retry failed requests.
  - Battery low → to pause heavy tasks.
- Your app needs to **communicate internally**:
  - One component tells another when a task completes.
- You want a **lightweight trigger** instead of continuously running a Service.

👉 **Don't use BroadcastReceiver for:**

- Long-running tasks (use WorkManager or Service instead).
- Real-time updates (use LiveData/Flow/Callback).

🔷 7. Interview-Style Answer

"A **BroadcastReceiver** lets an app listen for system-wide or custom events. For example, receiving a notification when the device boots, when battery is low, or when Wi-Fi connects. We use it when we need to respond to these events without keeping a Service running all the time. For long-running work, we usually start a Service or schedule WorkManager from inside a BroadcastReceiver. So BroadcastReceiver is best for **lightweight, event-driven triggers**."

🔷 Common System Broadcasts in Android

1. Boot Completed

- **Action:** `Intent.ACTION_BOOT_COMPLETED`
- **Use case:** Restart scheduled jobs, alarms, or WorkManager after device reboot.

---

2. Battery Events

- **Low battery:** `Intent.ACTION_BATTERY_LOW`
  - Pause downloads or heavy tasks.
- **Battery okay:** `Intent.ACTION_BATTERY_OKAY`
  - Resume tasks when battery is normal.

---

3. Power Events

- **Power connected:** `Intent.ACTION_POWER_CONNECTED`
  - Start charging-only tasks (e.g., syncing large data).
- **Power disconnected:** `Intent.ACTION_POWER_DISCONNECTED`
  - Stop tasks that need charging.

---

4. Network Events

- **Connectivity change:** `ConnectivityManager.CONNECTIVITY_ACTION` *(deprecated in Android N, use `NetworkCallback` instead)*
  - Retry failed requests when network is back.
- **Wi-Fi state changed:** `WifiManager.WIFI_STATE_CHANGED_ACTION`.

---

5. Package Events

- **App installed:** `Intent.ACTION_PACKAGE_ADDED`
- **App removed:** `Intent.ACTION_PACKAGE_REMOVED`
- **App replaced (updated):** `Intent.ACTION_PACKAGE_REPLACED`

---

6. Media Events

- **Unmount/remount external storage:**
  - `Intent.ACTION_MEDIA_MOUNTED`
  - `Intent.ACTION_MEDIA_UNMOUNTED`

---

7. SMS & Telephony

- **SMS received:** `Telephony.Sms.Intents.SMS_RECEIVED_ACTION`
- **Phone state changed:** `TelephonyManager.ACTION_PHONE_STATE_CHANGED`

AsyncTask (deprecated) vs Coroutines

🔷 1. AsyncTask (Deprecated)

- Introduced in **API 3** to run tasks in the background and update UI in the main thread.
- Provided an easy way to avoid `Thread` + `Handler` boilerplate.

Key points:

- Has **three main methods**:
  - `doInBackground()` → background thread work.
  - `onProgressUpdate()` → update UI with progress.
  - `onPostExecute()` → final UI update after task finishes.
- **Easy but limited**:
  - One-time short tasks only.
  - Not lifecycle-aware (keeps running even if Activity/Fragment is destroyed → memory leaks).
  - Can cause crashes (e.g., `AsyncTask` referencing destroyed Activity).

Status:

- **Deprecated in API 30 (Android 11)**.
- Google recommends **Coroutines / WorkManager** instead.

🔷 2. Kotlin Coroutines

- A modern **asynchronous programming** solution in Kotlin.
- Provides **lightweight threads (coroutines)** that can be **suspended and resumed** without blocking.

Key points:

- **Lifecycle-aware** (with Jetpack's `lifecycleScope` and `viewModelScope`).
- Can easily switch between **Dispatchers**:
  - `Dispatchers.Main` → UI thread.
  - `Dispatchers.IO` → network/disk I/O.
  - `Dispatchers.Default` → CPU-intensive tasks.
- Supports **structured concurrency** → jobs are tied to a scope (cancelled automatically when scope ends).
- Handles **complex async flows** → sequential, parallel, cancellation, exception handling.

- Works with **suspend functions** like Retrofit, Room.

🔷 3. Comparison

Feature	AsyncTask	Coroutines
Introduced	API 3 (Android 1.5)	Kotlin (Jetpack supported)
Status	Deprecated (API 30)	Modern solution
Thread Handling	Fixed worker thread pool	Flexible (Main, IO, Default)
Lifecycle-aware	❌ No (memory leaks)	✅ Yes (with scopes)
Error handling	Limited	Try/catch, structured concurrency
Use case	One-off background tasks	Networking, DB, async flows

AsyncTask (Old, Deprecated)

java — Copy code

```java
private class DownloadTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... urls) {
        // Background task
        return downloadFile(urls[0]);
    }

    @Override
    protected void onPostExecute(String result) {
        // Update UI
        textView.setText(result);
    }
}
```

Ask ChatGPT

Coroutines (Modern Way)

kotlin — Copy code

```kotlin
// Inside ViewModel or Activity
lifecycleScope.launch {
    val result = withContext(Dispatchers.IO) {
        downloadFile("https://example.com/file")
    }
    textView.text = result
}
```

🔷 Interview Q&A – AsyncTask vs Coroutines

❓ Q1: What is AsyncTask and why was it deprecated?

✅ **Answer:**
"AsyncTask was an older Android API to run background tasks and update the UI thread without using Threads + Handlers. It was deprecated in Android 11 because it wasn't lifecycle-aware, often caused memory leaks, was limited to short one-time tasks, and gave poor control over cancellation and error handling."

❓ Q2: What is the modern alternative to AsyncTask?

✅ **Answer:**
"The recommended alternative is **Kotlin Coroutines** for async work. For tasks that must survive app restarts (like syncing), we use **WorkManager**."

❓ Q3: How are Coroutines better than AsyncTask?

✅ **Answer:**
"Coroutines are lightweight, lifecycle-aware, and support structured concurrency. They let us easily switch between threads (Main, IO, Default), handle errors with try/catch, and integrate with libraries like Retrofit and Room. AsyncTask, on the other hand, was rigid, tied to a thread pool, and not lifecycle-safe."

❓ Q4: Can you give a simple example?

✅ **Answer:**
"AsyncTask had methods like `doInBackground()` and `onPostExecute()`. With Coroutines, we can do the same in just a few lines using `lifecycleScope.launch {}` and `withContext(Dispatchers.IO)`. It's more concise and safer."

❓ Q5: When should I use Coroutines vs WorkManager?

✅ **Answer:**

- Use **Coroutines** for tasks while the app is active (e.g., network call, DB query).

- Use **WorkManager** when you need guaranteed execution, even if the app is killed or the device restarts.

⚡ **Quick Interview Line:**
"AsyncTask is deprecated because it wasn't lifecycle-aware and caused memory leaks. Today we use Coroutines for async work—they are safe, modern, and flexible. For guaranteed background tasks across reboots, we use WorkManager."

What Does Lifecycle-Aware Mean?

- An Android component (like **Activity** or **Fragment**) has a **lifecycle** → it can be created, started, resumed, paused, stopped, destroyed.
- A **lifecycle-aware** task/component automatically **knows when to start, pause, or stop** depending on the lifecycle state.

👉 In simple words:

- **Lifecycle-aware = it automatically stops when the Activity/Fragment is destroyed**, so no crashes or memory leaks.

Internal vs External Storage

1. Internal Storage

- **Private app storage** inside the device.
- Files saved here are by default **accessible only to your app**.

- Files are deleted when the app is uninstalled.
- Location: `/data/data/<package_name>/files/`

Key points:

- Good for **sensitive/private data** (login tokens, app configs).
- No permission needed.
- Less space compared to external storage.

Example:

```
val file = File(context.filesDir, "user.txt")
file.writeText("Hello internal storage")
val text = file.readText()
```

2. External Storage

- Shared storage area (can be internal SD card partition or external removable SD card).
- Files are **public** → accessible by user and other apps.
- Survives app uninstall (if saved in shared/public directories).
- Location: `/storage/emulated/0/`

Key points:

- Good for **media files** (images, videos, downloads) that user expects to see outside your app.
- Needs **runtime permissions**:
  - `READ_EXTERNAL_STORAGE`
  - `WRITE_EXTERNAL_STORAGE` (before Android 10, now replaced with **Scoped Storage**).
- Since Android 10 (API 29): **Scoped Storage** introduced → each app gets sandboxed external directory (`/Android/data/<package_name>/`).

Example:

```
val file = File(context.getExternalFilesDir(null), "user.txt")
file.writeText("Hello external storage")
val text = file.readText()
```

🔷 3. Comparison Table

Feature	Internal Storage 🏠	External Storage 📂
Access	Only your app (private)	Public / other apps can access
Permission	❌ Not required	✅ Required (read/write)

Feature	Internal Storage 🏠	External Storage 📂
When Deleted	On app uninstall	Remains even after uninstall
Use case	Sensitive/private data	Media, downloads, user files
Space	Limited	Larger (user visible)
Path	`/data/data/<pkg>/files/`	`/storage/emulated/0/`

🔷 4. Solution (When to Use What)

👉 **Use Internal Storage** when:

- Storing private/sensitive data (auth tokens, configs).
- You don't want other apps/users to access it.

👉 **Use External Storage** when:

- Saving files the user expects to access outside your app (photos, videos, documents).
- Sharing data between apps.

✅ Interview-Style Answer

"Internal storage is private to the app, requires no permission, and gets deleted when the app is uninstalled—best for sensitive or private data. External storage is public, requires permissions, survives uninstall, and is used for user-visible files like photos or downloads. On modern Android we use **Scoped Storage** for secure access to external files."

Scoped Storage

1. What is Scoped Storage?

- Introduced in **Android 10 (API 29)**.
- Changed how apps access **external storage** for **better privacy & security**.
- Instead of giving apps **full access to all external files**, each app gets a **sandboxed directory** inside external storage:
- `/storage/emulated/0/Android/data/<package_name>/files/`
- Users control file access more strictly.

🔷 2. Key Changes with Scoped Storage

1. **App Sandbox in External Storage**
   - Each app can freely read/write in its own folder → no permission needed.
   - Other apps cannot access it.
2. **Public Media Collections (MediaStore API)**
   - To access shared photos, videos, or audio → must use **MediaStore API**.
   - Users may see a permission dialog for each file (from Android 11 onwards).
3. **Permissions**
   - `READ_EXTERNAL_STORAGE` → needed for shared media access.
   - `WRITE_EXTERNAL_STORAGE` → mostly ignored on Android 10+ (apps should use scoped folders).
4. **File Picker (Storage Access Framework)**
   - For documents (PDF, TXT, etc.), apps should use the **system file picker** instead of raw file paths.

🔷 3. Why Scoped Storage?

- **Privacy** → Apps no longer get access to *all* user files.
- **Security** → Prevents malicious apps from scanning personal data.
- **User Control** → Users can grant access to only selected files.

🔷 4. Example Usage

Save to app's sandbox (No permission needed)

```
val file = File(context.getExternalFilesDir(null), "notes.txt")
file.writeText("Hello Scoped Storage")
```

Save image to MediaStore (Public Gallery)

```
val values = ContentValues().apply {
    put(MediaStore.Images.Media.DISPLAY_NAME, "myImage.jpg")
    put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg")
    put(MediaStore.Images.Media.RELATIVE_PATH, "Pictures/MyApp")
}

val uri =
context.contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI
, values)
uri?.let {
    context.contentResolver.openOutputStream(it)?.use { stream ->
```

```
            bitmap.compress(Bitmap.CompressFormat.JPEG, 100, stream)
        }
}
```

🔷 5. Comparison (Pre vs Post Scoped Storage)

Feature	Before Scoped Storage	After Scoped Storage (Android 10+)
Access to External Files	Full (with READ/WRITE permission)	Only app sandbox, or via MediaStore/File Picker
Privacy	Low (apps could read all files)	High (apps restricted to sandbox)
User Control	Single permission grant	Per-file / per-folder access
Best For	Legacy apps (before API 29)	Modern apps (API 29+)

🔷 6. Solution (When to Use What)

👉 **Scoped Storage (default in Android 10+)**

- Always use sandbox for app-private files.
- Use MediaStore for shared media.
- Use Storage Access Framework (file picker) for documents.

👉 **Legacy storage (opt-out, Android 10 only)**

- Temporary opt-out with `requestLegacyExternalStorage="true"`.
- Not available in Android 11+.

Interview-Style Answer

“Scoped Storage was introduced in Android 10 to improve user privacy. Earlier, apps with storage permission could access all external files. Now each app gets a sandboxed directory, and for shared media we use the MediaStore API. Sensitive data goes to app sandbox, user-visible files go to MediaStore, and documents via the Storage Access Framework. Scoped Storage is now mandatory from Android 11 onward.”

Runtime Permissions

🔷 1. What are Runtime Permissions?

- Introduced in **Android 6.0 (Marshmallow, API 23)**.
- Before that → Permissions were granted **at install time**.
- Now → "dangerous" permissions must be requested **at runtime** (when actually needed).
- User can **allow, deny, or revoke anytime** from system settings.

🔷 2. Permission Types

1. **Normal Permissions**
   - Automatically granted at install time.
   - Example: `INTERNET`, `ACCESS_NETWORK_STATE`.
2. **Dangerous Permissions** (require runtime request)
   - Grouped into categories:
     - **Camera** → `Manifest.permission.CAMERA`
     - **Storage** → `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`
     - **Location** → `ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`
     - **Contacts**, **Microphone**, **SMS**, etc.

🔷 4. CAMERA vs STORAGE Example

- **CAMERA Permission**
  - Needed to open camera directly from your app.
  - If you only use `ACTION_IMAGE_CAPTURE` via `Intent` → some devices may not require it.
  - Runtime request required on Android 6+.
- **STORAGE Permission**
  - Before Android 10 → `READ/WRITE_EXTERNAL_STORAGE` required to access external files.
  - Android 10+ → replaced by **Scoped Storage** → usually don't need `WRITE_EXTERNAL_STORAGE` anymore.
  - Still need permission to read user's media (images, audio, video).

🔷 3. Steps for Runtime Permissions

Step 1: Declare in Manifest

xml Copy code

```xml
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

Step 2: Check if Granted

kotlin Copy code

```kotlin
if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
    != PackageManager.PERMISSION_GRANTED) {
    // Request it
}
```

Step 3: Request Permission

kotlin Copy code

```kotlin
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.CAMERA),
    100
)
```

Step 4: Handle Result

kotlin Copy code

```kotlin
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>,
    if (requestCode == 100 && grantResults.isNotEmpty() && grantResults[0] == PackageMana
        // Permission granted
        openCamera()
    } else {
        // Permission denied
    }
}
```

🔷 5. Interview Pitfalls

- **If user denies twice → show rationale:**

```
if (ActivityCompat.shouldShowRequestPermissionRationale(this,
Manifest.permission.CAMERA)) {
    // Show why this permission is needed
}
```

- **Don’t ask for all permissions at once.** Request when needed.
- **Scoped Storage impact:** WRITE_EXTERNAL_STORAGE is deprecated in Android 10+.

🔷 6. Solution (When to Use)

👉 **Use Runtime Permissions** for:

- **CAMERA** → taking pictures or video directly.
- **STORAGE** → accessing shared files/media (images, documents).
- Always **check before accessing**.

👉 **Best Practices**:

- Request only when the feature is triggered (not on app launch).
- Handle denial gracefully (disable the feature, don’t crash).
- For modern Android → prefer **Scoped Storage** + **MediaStore API** over raw storage permissions.

✅ Interview-Style Answer

“Runtime permissions were introduced in Android 6 to give users more control. Dangerous permissions like CAMERA and STORAGE must be requested while the app is running, not just in the manifest. We first check if the permission is granted, then request it if not, and handle the callback. For example, to capture an image we request CAMERA, and to access files we request STORAGE (though on Android 10+ we use Scoped Storage). Best practice is to request permissions only when needed and handle denial gracefully.”

ContenProvider

1. What is a ContentProvider?

- A **ContentProvider** manages access to a **structured set of data**.
- It acts as an **interface for sharing data between apps** (or within your app).
- Data is exposed through a **URI (content://)** instead of file paths.

👉 Example:

- Contacts app provides your phone contacts through the **Contacts ContentProvider** (`content://contacts/people`).
- Other apps (like WhatsApp, Dialer) can read those contacts with proper permission.

🔷 2. Why Use ContentProvider?

- To **share data across apps** in a secure and controlled way.
- To provide a **standard API** for querying data (similar to a database).
- To allow **CRUD operations** (Create, Read, Update, Delete) using URIs.

🔷 3. ContentProvider URI Structure

```
content://authority/path/id
```

- **content://** → Always starts with this.
- **authority** → Unique name of provider (usually package name).
- **path** → Table or resource.
- **id** → Optional specific row/item.

👉 Example:

```
content://com.android.contacts/contacts/1
```

= Get contact with ID 1.

🔷 4. How to Implement

Declare in Manifest

xml Copy code

```xml
<provider
    android:name=".MyProvider"
    android:authorities="com.example.myapp.provider"
    android:exported="true"
    android:grantUriPermissions="true" />
```

Extend ContentProvider

kotlin Copy code

```kotlin
class MyProvider : ContentProvider() {
    override fun onCreate(): Boolean {
        // Initialize DB or storage
        return true
    }

    override fun query(uri: Uri, projection: Array<out String>?, selection: String?,
                       selectionArgs: Array<out String>?, sortOrder: String?): Cursor? {
        // Return data as Cursor
        return db.query("users", projection, selection, selectionArgs, null, null, sortOr
    }

    override fun insert(uri: Uri, values: ContentValues?): Uri? { ... }
    override fun update(uri: Uri, values: ContentValues?, selection: String?, selectionAr
    override fun delete(uri: Uri, selection: String?, selectionArgs: Array<out String>?):
    override fun getType(uri: Uri): String? { ... } // MIME type
}
```

🔷 5. Accessing Data from ContentProvider

Query Another App's Provider

kotlin Copy code

```kotlin
val cursor = contentResolver.query(
    ContactsContract.Contacts.CONTENT_URI,
    null, null, null, null
)

while (cursor?.moveToNext() == true) {
    val name = cursor.getString(cursor.getColumnIndexOrThrow(ContactsContract.Contacts.DI
    println(name)
}
cursor?.close()
```

👉 Here, we're accessing **Contacts ContentProvider** to read phone contacts.

🔷 6. Security & Permissions

- If you want to expose your ContentProvider, you must declare **permissions**.

xml Copy code

```xml
<provider
    android:name=".MyProvider"
    android:authorities="com.example.myapp.provider"
    android:exported="true"
    android:readPermission="com.example.READ_DATA"
    android:writePermission="com.example.WRITE_DATA"/>
```

- Apps must request these permissions in their manifest.

👉 **Use ContentProvider when**:

- You want to **share data between apps** (contacts, media, messages).
- You want to allow other apps to securely query/insert/update/delete your data.
- You want a **standard interface** to your app's database/content.

👉 **Don't use ContentProvider when**:

- Data is **private** to your app (then use Room/SharedPreferences/Files instead).

✅ Interview-Style Answer

"A ContentProvider is one of the four core Android components, used to manage and share data between apps using URIs. It provides a standard interface for CRUD operations, backed usually by a SQLite database or files. For example, the Contacts app exposes contacts via ContentProvider. We use it when data must be shared between apps securely; for app-private data, Room or SharedPreferences are sufficient."

🔷 ContentProvider – Interview Flashcards

❓ Q1: What is a ContentProvider?

✅ **Answer:**
A ContentProvider is one of the four core Android components, used to manage and share structured data between apps using URIs (`content://`).

❓ Q2: Why use ContentProvider instead of SQLite/Room directly?

✅ **Answer:**
SQLite/Room are app-private. ContentProvider provides a **standardized, permission-controlled interface** to allow other apps to query or modify your app's data.

❓ Q3: What's the URI format for ContentProvider?

✅ **Answer:**

```
content://authority/path/id
```

- **authority** = provider's unique name (usually package name).
- **path** = table/resource.
- **id** = optional row/item id.

❓ Q4: How does an app consume ContentProvider data?

✅ **Answer:**
By using the **ContentResolver API**:

```
val cursor = contentResolver.query(uri, projection, null, null, null)
```

This returns a **Cursor** for reading rows.

❓ Q5: What operations are supported by ContentProvider?

✅ **Answer:**
CRUD operations → **Insert, Query, Update, Delete** via URIs.

❓ Q6: How do we secure a ContentProvider?

✅ **Answer:**
By setting `android:readPermission` / `android:writePermission` in the manifest. Only apps with granted permissions can access it.

❓ Q7: Give a real-world example of ContentProvider.

✅ **Answer:**

- **Contacts ContentProvider** → `content://contacts/people`
- **MediaStore ContentProvider** → `content://media/external/images/media`

❓ Q8: When should you NOT use ContentProvider?

✅ **Answer:**
When the data is **private to your app only** (then use Room, SharedPreferences, or files).

File i/o APIs in Android

🔷 1. File I/O in Android – Overview

File I/O = Reading and writing files in Android.
Since Android runs in a sandboxed environment, apps can't freely read/write anywhere — they must use **specific APIs** for internal or external storage.

🔷 2. Internal Storage File APIs

- Files are **private to the app**.
- No permission required.
- Deleted automatically when app is uninstalled.

Example:

```
// Write
openFileOutput("data.txt", Context.MODE_PRIVATE).use {
    it.write("Hello Internal Storage".toByteArray())
}

// Read
val text = openFileInput("data.txt").bufferedReader().use { it.readText() }
```

🔷 3. External Storage File APIs

- Files can be **public/shared** (media, downloads).
- Requires runtime permissions (`READ_EXTERNAL_STORAGE`, deprecated `WRITE_EXTERNAL_STORAGE`).
- Android 10+ → **Scoped Storage** enforced.

Example:

```
val file = File(getExternalFilesDir(null), "notes.txt")
file.writeText("Hello External Storage")

val text = file.readText()
```

🔷 4. MediaStore API (for Shared Media)

- From Android 10+, recommended way to store **user-visible files** (photos, videos, audio).
- Works with **content URIs** instead of direct file paths.

Example:

```
val values = ContentValues().apply {
    put(MediaStore.Images.Media.DISPLAY_NAME, "myImage.jpg")
    put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg")
    put(MediaStore.Images.Media.RELATIVE_PATH, "Pictures/MyApp")
}
val uri =
contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
values)
uri?.let {
    contentResolver.openOutputStream(it)?.use { stream ->
        bitmap.compress(Bitmap.CompressFormat.JPEG, 100, stream)
    }
}
```

🔷 5. Storage Access Framework (SAF)

- For accessing **documents and files** from any storage provider (Google Drive, Downloads, USB).
- Uses **system file picker UI** → no direct path access.

Example:

```
val intent = Intent(Intent.ACTION_OPEN_DOCUMENT).apply {
    addCategory(Intent.CATEGORY_OPENABLE)
    type = "application/pdf"
}
startActivityForResult(intent, 100)
```

🔷 6. Comparison of File I/O APIs

API	Use Case	Permission Needed	Notes
Internal Storage	Private app files	❌ No	Deleted on uninstall
External Storage (sandbox)	App-specific external files	❌ No	Scoped storage enforced
MediaStore API	Shared media (images, audio, video)	✅ Yes (READ)	Modern, user-visible

API	Use Case	Permission Needed	Notes
Storage Access Framework	Documents, cloud storage, file picker	❌ No direct access	User selects files

---

🔷 7. Solution (When to Use What)

👉 **Use Internal Storage** for:

- Configs, tokens, sensitive files.

👉 **Use External Storage (sandbox)** for:

- App-specific large files (like cached images, downloads).

👉 **Use MediaStore API** for:

- User-visible media (gallery images, videos, music).

👉 **Use Storage Access Framework** for:

- Files chosen by user (PDF, DOCX, Google Drive).

✅ Interview-Style Answer

“Android provides different File I/O APIs for different use cases. Internal Storage is private to the app, no permission required. External Storage (sandboxed) is for app-specific files. For shared media like photos or audio, we use the MediaStore API. And for user-selected documents across storage providers, we use the Storage Access Framework. Scoped Storage in Android 10+ enforces these rules for privacy and security.”

🔷 Interview Flashcards

❓ Q1: Do you need permission for Internal Storage?

✅ **Answer:** No, it’s private to the app.

❓ Q2: Which API is recommended for saving images to Gallery?

✅ **Answer:** MediaStore API.

❓ **Q3: Which API lets user pick a file from Google Drive?**

✅ **Answer:** Storage Access Framework (SAF).

❓ **Q4: What replaced WRITE_EXTERNAL_STORAGE in Android 10+?**

✅ **Answer:** Scoped Storage (sandbox + MediaStore).

Retrofit basics

🔷 1. What is Retrofit?

- **Retrofit** is a type-safe HTTP client for Android (by Square).

- Makes API calls easier by converting HTTP APIs into **Java/Kotlin interfaces**.
- Uses **annotations** (`@GET`, `@POST`, `@Query`, `@Body`) to describe API endpoints.
- Works seamlessly with **Coroutines, RxJava, Gson/Moshi (JSON parsers)**.

🔷 2. Service Interface (Core of Retrofit)

You define a **Service Interface** with API endpoints.
Each method corresponds to a network call.

Example:

```
interface ApiService {
    @GET("users")
    suspend fun getUsers(): List<User>
}
```

- Here, `@GET("users")` tells Retrofit to call `BASE_URL + "users"`.
- `suspend` → coroutine support.
- Return type = `List<User>` parsed from JSON.

🔷 3. Common Retrofit Annotations

@GET

- Makes a **GET request**.

```
@GET("users")
suspend fun getUsers(): List<User>
```

@POST

- Makes a **POST request** with a body.

```
@POST("users")
suspend fun createUser(@Body user: User): Response<User>
```

@Query

- Adds query parameters (`?key=value`).

```
@GET("search")
suspend fun searchUsers(@Query("name") name: String): List<User>
```

👉 Example: `GET /search?name=manav`

@Path

- Replaces `{}` in URL.

```
@GET("users/{id}")
suspend fun getUser(@Path("id") id: Int): User
```

👉 Example: `GET /users/101`

@Body

- Sends an object in request body (usually JSON).

```
@POST("login")
suspend fun login(@Body loginRequest: LoginRequest): Response<Token>
```

@Header / @Headers

- Add headers dynamically or statically.

```
@GET("profile")
suspend fun getProfile(@Header("Authorization") token: String): Profile
```

🔷 4. Creating Retrofit Instance

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(GsonConverterFactory.create()) // for JSON
    .build()

val api = retrofit.create(ApiService::class.java)
```

🔷 5. Making a Call with Coroutines

```
lifecycleScope.launch {
    try {
        val users = api.getUsers()
        println(users)
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

🔷 6. Solution (When & Why Use Retrofit)

👉 Use Retrofit when:

- You need **type-safe, clean API calls**.
- You want to easily parse JSON → Kotlin/Java objects.
- You want easy integration with **Coroutines** or **RxJava**.

👉 Alternatives:

- **OkHttp** (lower-level, manual parsing).
- **Volley** (older, more verbose).

✅ Interview-Style Answer

"Retrofit is a type-safe HTTP client for Android. We define a service interface with annotations like `@GET`, `@POST`, `@Query`, etc., and Retrofit automatically creates the implementation. For example, `@GET("users")` fetches data, and `@Query("id")` adds query parameters. Retrofit is preferred because it integrates with Coroutines, handles JSON parsing, and reduces boilerplate compared to raw OkHttp or Volley."

🔷 Interview Flashcards

❓ Q1: What does `@GET("users")` do?

✅ **Answer:** It maps to an HTTP GET request for `BASE_URL + "users"`.

❓ Q2: Difference between @Query and @Path?

✅ **Answer:**

- `@Query` adds parameters (`?key=value`).
- `@Path` replaces `{}` in URL path.

❓ Q3: How does Retrofit parse JSON?

✅ **Answer:** Using `ConverterFactory` (like Gson, Moshi, Kotlinx Serialization).

❓ Q4: How to add headers in Retrofit?

✅ **Answer:** With `@Header` (dynamic) or `@Headers` (static).

Gson/Moshi converters

🔷 1. What is Gson / Moshi?

Both **Gson** and **Moshi** are **JSON parsing libraries** for Java/Kotlin.

- Convert JSON ↔ Java/Kotlin objects.
- Used as **Retrofit converters** to automatically map API responses into model classes.

👉 Example:
JSON response:

```
{
  "id": 1,
  "name": "Manav"
}
```

Model class:

```
data class User(val id: Int, val name: String)
```

With Gson/Moshi → JSON is **automatically parsed** into `User`.

🔷 2. Gson (by Google)

- Older, widely used.
- Easy to use, built-in support in Retrofit.
- Supports reflection-based parsing.
- **Slower** than Moshi (because of reflection).
- Flexible with custom serializers/deserializers.

Retrofit Setup with Gson:

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

🔷 3. Moshi (by Square)

- Newer, from Square (same team as Retrofit, OkHttp).
- Built for Kotlin → better performance, null-safety.
- Uses **Kotlin reflection** and supports Kotlin's `@Json` annotations.
- Works better with Kotlin's `data class` + non-null types.
- Generally **faster** and more strict than Gson.

Retrofit Setup with Moshi:

```
val moshi = Moshi.Builder().build()

val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .build()
```

🔷 4. Comparison – Gson vs Moshi

Feature	Gson	Moshi
Developer	Google	Square (Retrofit creators)
Age	Older, widely adopted	Newer, modern (Kotlin-first)
Performance	Slower (reflection-heavy)	Faster (optimized)
Kotlin Support	Limited	Strong (null-safety, Kotlin adapters)
Error Handling	Lenient (ignores unknown keys)	Strict (fails on unknown keys unless configured)
Usage	Very common, lots of docs	Growing, recommended with Kotlin

---

🔷 5. Example: Mapping JSON with Gson vs Moshi

Gson

```
val gson = Gson()
val user = gson.fromJson("""{"id":1,"name":"Manav"}""", User::class.java)
```

Moshi

```
val moshi = Moshi.Builder().build()
val adapter = moshi.adapter(User::class.java)
val user = adapter.fromJson("""{"id":1,"name":"Manav"}""")
```

🔷 6. Solution (When to Use What)

👉 **Use Gson if**:

- You're working on an older project that already uses Gson.
- You want lenient parsing and don't need strict null-safety.

👉 **Use Moshi if**:

- You're starting a new Kotlin project.
- You want better performance, strict parsing, and null-safety.
- You're already using Retrofit + Kotlin data classes.

✅ Interview-Style Answer

"Gson and Moshi are JSON parsing libraries used as Retrofit converters. Gson, from Google, is older and widely used but slower and less strict. Moshi, from Square, is newer, Kotlin-first, and faster with better null-safety. In practice, Gson is common in legacy apps, but for modern Kotlin projects Moshi is preferred."

RxJava

1. What is RxJava?

- **RxJava** = **Reactive Extensions for Java**.
- A **reactive programming library** for composing asynchronous and event-based programs using **observable streams**.
- You describe **what to do with data streams**, not *how* to manage threads.

👉 Think of it like:

- Instead of pulling data (imperative), data **flows/reacts** to subscribers when available.

🔷 2. Core Concepts

Observable

- Emits a stream of data/events.
- Example: network responses, button clicks, sensor data.

Observer (Subscriber)

- Listens to the Observable and reacts to emissions.

Operators

- Transform or combine streams (`map`, `flatMap`, `filter`, `debounce`).

Schedulers

- Decide **which thread** the work runs on:
  - `Schedulers.io()` → network, disk I/O.
  - `Schedulers.computation()` → CPU-intensive work.
  - `AndroidSchedulers.mainThread()` → update UI.

🔷 3. Basic Example

Without RxJava (traditional)

```
Thread {
    val result = api.getUsers() // network call
    runOnUiThread {
        textView.text = result.toString()
    }
}.start()
```

With RxJava

```
api.getUsers()
    .subscribeOn(Schedulers.io())               // run in background
    .observeOn(AndroidSchedulers.mainThread()) // update UI
    .subscribe(
        { result -> textView.text = result.toString() }, // onNext
        { error -> error.printStackTrace() }             // onError
    )
```

🔷 4. Real-World Use Cases

- API calls with **Retrofit + RxJava**.
- Handling **continuous streams** (search text input, sensor data).
- Chaining multiple async tasks (`flatMap`).
- Debouncing user input (e.g., typing search box).

🔷 5. RxJava vs Coroutines

Feature	RxJava	Coroutines
Style	Reactive (streams, observables)	Sequential, structured concurrency
Learning Curve	Steep (many operators)	Easier (looks like normal code)
Best for	Complex async data streams (events)	Simpler async tasks (network, DB)
Integration	Retrofit, Room, UI events	Retrofit (suspend), Room (suspend)

🔷 6. Interview-Style Answer

“RxJava is a reactive programming library for handling asynchronous and event-based data streams. It uses Observables to emit data, Observers to consume it, and Operators to transform streams. We control threading with Schedulers. For example, we can run a network call on `Schedulers.io()` and observe results on `AndroidSchedulers.mainThread()`. RxJava is powerful for complex async workflows, while today Kotlin Coroutines are preferred for simpler, sequential async programming.”

1. REST (Representational State Transfer)

- Standard **HTTP request-response model**.
- Client → sends request → Server → responds, then connection ends.
- **Stateless** → each request is independent (no persistent connection).
- Commonly used with **Retrofit** in Android.

👉 Example:

```
GET /users/1  → { "id": 1, "name": "Manav" }
```

Best for:

- CRUD operations (Create, Read, Update, Delete).
- APIs where real-time updates are not critical.
- Simple, reliable, widely supported.

🔷 2. WebSockets

- **Full-duplex communication** over a single TCP connection.
- Connection remains **open** until explicitly closed.
- Server can **push data anytime**, not just when client requests.
- Commonly used with **OkHttp WebSocket API** in Android.

👉 Example:

- Client connects: `wss://example.com/socket`
- Client → sends: `"Hello"`
- Server → pushes: `"Hi! New data available"`

Best for:

- Real-time apps (chat, notifications, stock prices, live scores).
- Situations where data must be **pushed frequently**.
- Low-latency communication.

🔷 3. Comparison Table

Feature	REST (HTTP)	WebSockets
Connection	Short-lived (per request)	Persistent (kept open)
Communication	Request → Response only	Full-duplex (both directions anytime)
State	Stateless	Stateful (connection maintained)
Performance	Higher overhead for frequent calls	Low overhead for continuous updates
Best for	CRUD, infrequent updates	Real-time updates, chat, streaming
Android API	Retrofit, OkHttp (HTTP)	OkHttp WebSocket

🔷 4. Solution (When to Use What)

👉 Use **REST** when:

- You're doing CRUD operations (user profiles, login, product data).
- Data updates are not frequent.
- API needs to be widely accessible and cacheable.

👉 Use **WebSockets** when:

- You need **real-time, continuous updates** (chat, notifications, live feeds).
- Polling with REST would be inefficient.
- Low-latency is required.

✅ Interview-Style Answer

"REST is request-response based: client requests and server responds, then the connection ends. It's stateless and best for CRUD operations. WebSockets, on the other hand, maintain a persistent full-duplex connection so both client and server can send messages anytime. REST is great for simple APIs, while WebSockets are ideal for real-time apps like chat or live scores."

Suspend Function

. What is a `suspend` function?

- A **special Kotlin function** that can be **paused (suspended)** and **resumed** later without blocking a thread.
- Declared with the `suspend` keyword.
- Runs inside a **Coroutine**.
- Looks like sequential code but is **non-blocking**.

👉 In simple words:
`suspend` = "this function might take time (like network/database), but it won't block the thread."

🔷 2. Why do we need `suspend` functions?

- Android's main thread must remain free (UI thread).
- Long tasks (network, DB) can freeze UI if not async.
- `suspend` functions allow async work while keeping code **clean & readable**.

🔷 3. Example

Without Coroutines

```
fun fetchData(): String {
    Thread.sleep(2000) // Blocks main thread ❌
    return "Hello World"
}
```

With Suspend Function

```
suspend fun fetchData(): String {
    delay(2000) // Non-blocking suspend
    return "Hello World"
```

```
}

lifecycleScope.launch {
    val result = fetchData()
    println(result) // Prints after 2 sec, but UI not blocked ✅
}
```

🔷 4. Key Points

1. **Suspend = Non-blocking**
    - Uses `delay()` instead of `Thread.sleep()`.
    - Under the hood, it suspends coroutine → frees thread → resumes later.
2. **Only callable from Coroutines**
    - You can't call a `suspend` function directly.
    - Must be called inside another `suspend` function or a `launch`/`async` coroutine builder.
3. **Looks sequential, but async**
    - Avoids "callback hell."
    - Code is easy to read.

🔷 5. Real-World Example (Retrofit + suspend)

```
interface ApiService {
    @GET("users")
    suspend fun getUsers(): List<User>
}

// Call inside coroutine
lifecycleScope.launch {
    val users = api.getUsers() // Suspended until response
    println(users)
}
```

🔷 6. Solution (When to Use suspend)

👉 Use `suspend` functions when:

- Calling APIs (Retrofit supports suspend).
- Reading/writing database (Room supports suspend).
- Doing async work (files, network, long-running tasks).

👉 Don't use `suspend` for:

- Instant calculations.
- Simple UI updates (normal functions are enough).

✅ Interview-Style Answer

“A `suspend` function in Kotlin is one that can be paused and resumed without blocking the thread. It’s used for long-running tasks like network or DB calls. For example, `suspend fun getUsers()` with Retrofit makes an async request but looks like normal code. `suspend` functions can only be called from a coroutine or another suspend function, which makes async programming safe and readable.”

Launch vs Async vs withContext

1. `launch` – Fire and Forget 🚀

- Starts a **new coroutine**.

- Returns a `Job` (not a result).
- Used when you want to **do something in the background** but don't need a return value.

```
lifecycleScope.launch {
    println("Downloading file...")
    delay(2000)
    println("Download complete ✅")
}
```

👉 Example use cases: updating UI, saving to DB, logging events.

🔷 2. `async` – Fire and Return a Result 🎯

- Starts a coroutine that **returns a result**.
- Returns a `Deferred<T>` → you call `.await()` to get the result.
- Used for **parallel/concurrent tasks** that produce values.

```
lifecycleScope.launch {
    val result1 = async { fetchData1() }
    val result2 = async { fetchData2() }
    println("Combined result: ${result1.await() + result2.await()}")
}
```

👉 Example use cases: fetching data from two APIs, combining results.

🔷 3. `withContext` – Switch Context (Thread) 🔄

- Does **not create a new coroutine**, but **switches the current coroutine's context** (like changing threads).
- Returns the result directly.
- Commonly used for running work on `Dispatchers.IO` or `Dispatchers.Default`.

```
lifecycleScope.launch {
    val data = withContext(Dispatchers.IO) {
        fetchDataFromDB() // runs on background thread
    }
    println("Got data: $data") // back on Main thread
}
```

👉 Example use cases: Database queries, network calls, CPU-heavy tasks.

🔷 4. Comparison Table

Feature	`launch`	`async`	`withContext`
Returns	`Job` (no result)	`Deferred<T>` (use `.await()`)	`T` (result directly)
Use Case	Fire-and-forget tasks	Parallel tasks, return results	Switch thread, return result
Creates Coroutine?	✅ Yes	✅ Yes	❌ No (just context switch)
Best For	Updating UI, logging	Concurrent API/database calls	One-shot background work

---

🔷 5. Real-Life Example

```
lifecycleScope.launch {
    // Fire-and-forget
    launch {
        println("Saving logs...")
    }

    // Parallel tasks
    val user = async { getUserFromApi() }
    val posts = async { getPostsFromApi() }
    println("User: ${user.await()}, Posts: ${posts.await()}")

    // Switch context
    val cached = withContext(Dispatchers.IO) {
        readFromDatabase()
    }
    println("Cache: $cached")
}
```

✅ Interview-Style Answer

“`launch` is fire-and-forget, it returns a Job and doesn’t give a result. `async` is used when you want a result, it returns a Deferred and you use `await()`. `withContext` doesn’t start a new coroutine, it just switches context (like moving to IO thread) and returns the result directly. I use `launch` for background tasks, `async` for concurrent computations, and `withContext` for switching threads safely.”

Flow vs StateFlow vs SharedFlow

1. Flow 🌊

- A **cold asynchronous stream** of data.
- Starts producing values **only when collected**.
- Each collector gets its own stream (independent).

```
fun numbersFlow(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(1000)
        emit(i) // emit values
    }
}

lifecycleScope.launch {
    numbersFlow().collect { println(it) } // prints 1,2,3
}
```

👉 **Best for:** One-shot streams like network calls, DB queries.

🔷 2. StateFlow 🌀

- A **hot stream** (always active once created).
- Always holds a **current state value** (like LiveData).
- New collectors get the **latest value immediately**.
- Requires an **initial value**.

```
val counter = MutableStateFlow(0)

// Update value
counter.value = 5

// Collect
lifecycleScope.launch {
    counter.collect { println(it) } // instantly prints "5"
}
```

👉 **Best for:** UI state (MVVM ViewModel → UI).

🔷 3. SharedFlow 🔊

- A **hot stream** like StateFlow, but does **not hold a single value**.
- Can **replay** a certain number of past values to new subscribers.
- No initial value required.

```
val events = MutableSharedFlow<String>()

// Emit event
lifecycleScope.launch { events.emit("User clicked!") }

// Collect
```

```
lifecycleScope.launch {
    events.collect { println(it) } // receives "User clicked!"
}
```

👉 **Best for:** Events (navigation, one-time messages, analytics).

🔷 4. Comparison Table

Feature	Flow (Cold)	StateFlow (Hot)	SharedFlow (Hot)
Type	Cold stream (starts on collect)	Hot stream (always active)	Hot stream (always active)
Initial Value	❌ Not required	✅ Required	❌ Not required
Stores Value	❌ No	✅ Keeps last value (state)	❌ No (but can replay N past values)
Collectors	Independent	All collectors get latest state	All collectors share the same events
Best for	One-shot data (network, DB)	UI state (ViewModel → Compose/Activity)	UI events (navigation, messages, logs)

🔷 5. Solution (When to Use What)

- **Flow** → use for streams that should start fresh each time (DB query, network request).
- **StateFlow** → use for **UI state** (compose, LiveData replacement).
- **SharedFlow** → use for **events** (snackbar, navigation, analytics).

✅ Interview-Style Answer

“Flow is a cold stream: it starts producing values only when collected, and each collector is independent. StateFlow is hot, it always has a current value and new collectors immediately get the latest value—like LiveData for state. SharedFlow is also hot but doesn’t hold a single state; instead, it emits events to multiple collectors and can replay past values. So I’d use Flow for one-shot data, StateFlow for UI state, and SharedFlow for one-time events.”

Cold vs Hot Flows

. Cold Flow 🌊 (default Flow)

- **Doesn't emit anything until collected.**
- Each collector gets its **own fresh stream** of data.
- Values are produced **per collector**.

👉 Example:

```
fun numbersFlow(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(500)
        emit(i)
    }
}

// Each collector gets a fresh sequence 1,2,3
lifecycleScope.launch {
    numbersFlow().collect { println("Collector1: $it") }
}
lifecycleScope.launch {
    numbersFlow().collect { println("Collector2: $it") }
}
```

Output:

```
Collector1: 1,2,3
Collector2: 1,2,3
```

✅ Best for: one-shot streams (API call, DB query).

🔷 2. Hot Flow 🔥 (StateFlow, SharedFlow, Channels)

- **Emits values whether or not there are collectors.**
- Collectors **share the same stream**.
- New collectors may miss past values (unless replayed).

👉 Example with `StateFlow`:

```
val state = MutableStateFlow(0)

// Update value
state.value = 42
```

```
// New collector instantly gets 42
lifecycleScope.launch {
    state.collect { println("Collector: $it") }
}
```

👉 Example with `SharedFlow`:

```
val events = MutableSharedFlow<String>()

// Emit event before collector joins
lifecycleScope.launch { events.emit("Clicked!") }

// Collector may miss it if not replayed
lifecycleScope.launch {
    events.collect { println(it) }
}
```

✅ Best for: **UI state, events, real-time streams** (like LiveData).

🔷 3. Comparison Table

Feature	Cold Flow 🌊	Hot Flow 🔥
Start Emitting	When collected	Always emitting (with/without collectors)
Collector Behavior	Each collector gets its own values	Collectors share the same values
Missed Values	No (always replayed for each collector)	Yes (if collector not active at the time)
Examples	`Flow`, `sequence`	`StateFlow`, `SharedFlow`, `Channel`
Best for	One-shot data (API, DB)	UI state, events, live updates

---

✅ Interview-Style Answer

“A **cold flow** produces values only when collected—every collector gets its own fresh stream, like a database query. A **hot flow** is always active and emits values regardless of collectors—like a radio broadcast. Collectors share the same values, and new ones may miss past emissions. StateFlow and SharedFlow are hot, whereas Flow is cold.”

Room Database

1. What is Room?

- **Room** is Google's **ORM (Object Relational Mapping) library** on top of SQLite.
- Provides a **type-safe, compile-time checked** way to access SQLite.
- Eliminates most boilerplate of raw SQLite.
- Works seamlessly with **Coroutines, Flow, LiveData**.

🔷 2. Why Use Room?

- Writing raw SQLite requires:
  - SQL queries as strings.
  - Manual parsing of `Cursor` into objects.
  - No compile-time query validation.

👉 Room solves this:

- Queries checked at **compile time** (no runtime crashes for bad SQL).
- Automatic **mapping between tables ↔ data classes**.
- Supports **reactive updates** via Flow/LiveData.

🔷 3. Core Components of Room

1. Entity (Table model)

Represents a table in the DB.

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey val id: Int,
    val name: String,
    val age: Int
)
```

---

2. DAO (Data Access Object)

Defines queries (insert, update, delete, read).

```
@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)

    @Query("SELECT * FROM users WHERE id = :id")
    suspend fun getUserById(id: Int): User

    @Query("SELECT * FROM users")
    fun getAllUsers(): Flow<List<User>>
}
```

---

3. Database

Holds the database and serves DAOs.

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

---

4. Create DB Instance

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java,
    "my_database"
).build()

val userDao = db.userDao()
```

🔷 4. Using Room in ViewModel

```
class UserViewModel(private val userDao: UserDao) : ViewModel() {
    val allUsers: Flow<List<User>> = userDao.getAllUsers()

    fun insertUser(user: User) = viewModelScope.launch {
        userDao.insertUser(user)
    }
}
```

🔷 5. Room with Coroutines & Flow

- Queries can return:
  - **suspend fun** → one-shot result.
  - **Flow<T>** → continuous updates (like LiveData).

👉 Example:

```
userDao.getAllUsers().collect { users ->
    println(users) // updates automatically when DB changes
}
```

🔷 6. Advantages of Room

✅ Compile-time SQL validation
✅ Less boilerplate than raw SQLite
✅ Works with Coroutines/Flow/LiveData
✅ Easy migration system
✅ Strongly typed models

🔷 7. Solution (When to Use Room)

👉 Use **Room** when:

- You need a **local database** with structured data.
- You want reactive updates (UI auto-updates when DB changes).
- You need compile-time safety for SQL.

👉 Don't use Room if:

- You just need **simple key-value storage** (then use SharedPreferences / DataStore).
- You're dealing with large binary data (better use file storage).

✅ Interview-Style Answer

"Room is a persistence library on top of SQLite. It provides an abstraction layer that validates SQL at compile time and automatically maps results to Kotlin/Java objects. It has three main components: `@Entity` for tables, `@Dao` for queries, and `@Database` for the DB holder. Room integrates with Coroutines and Flow, so queries can return reactive streams. It's preferred over raw SQLite because it reduces boilerplate and prevents runtime errors."

Entity, DAO, Database class

1. Entity → Table 🏛️

- Represents a **table** in the Room database.
- Annotated with `@Entity`.
- Each field = column.
- `@PrimaryKey` → unique identifier.

👉 Example:

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val name: String,
    val age: Int
)
```

✔️ Creates a `users` table with columns: `id`, `name`, `age`.

🔷 2. DAO (Data Access Object) → Queries 📜

- Defines **methods for DB operations**.
- Annotated with `@Dao`.
- Methods → annotated with `@Insert`, `@Update`, `@Delete`, `@Query`.
- Supports **suspend functions** and **Flow** for reactive updates.

👉 Example:

```
@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)

    @Update
    suspend fun updateUser(user: User)

    @Delete
    suspend fun deleteUser(user: User)

    @Query("SELECT * FROM users WHERE id = :id")
    suspend fun getUserById(id: Int): User?
```

```
    @Query("SELECT * FROM users")
    fun getAllUsers(): Flow<List<User>>
}
```

✔ Provides CRUD operations for the `users` table.

🔷 3. Database Class → DB Holder 🗄️

- Abstract class annotated with `@Database`.
- Lists all entities and DAOs.
- Must extend `RoomDatabase`.

👉 Example:

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

🔷 4. Putting It All Together

```
// Create DB instance
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java,
    "my_database"
).build()

// Use DAO
val userDao = db.userDao()

// Insert user
lifecycleScope.launch {
    userDao.insertUser(User(name = "Manav", age = 25))
}
```

✅ Interview-Style Answer

“In Room, an **Entity** represents a table (with fields as columns). A **DAO** defines methods for database operations like insert, query, update, and delete. The **Database class** is the main holder, annotated with `@Database`, where we list all entities and DAOs, and it extends `RoomDatabase`. These three together form the backbone of Room: Entity → DAO → Database.”

Coroutines & flow support in Room

. Room + Coroutines

Room integrates directly with Kotlin **Coroutines**:

- You can mark DAO methods as **`suspend`** → Room will run them off the main thread automatically.
- No need to create your own `Dispatchers.IO` block.

👉 Example:

```
@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)

    @Query("SELECT * FROM users WHERE id = :id")
    suspend fun getUserById(id: Int): User?
}
```

Usage:

```
viewModelScope.launch {
    val user = userDao.getUserById(1) // Suspends, no blocking
    println(user?.name)
}
```

✅ **Why useful?**

- Keeps DB operations **off the UI thread**.
- Cleaner than callbacks.

🔷 2. Room + Flow

Room DAOs can return **Flow** → a reactive stream of database results.

- Whenever the underlying table changes → Flow **emits new values** automatically.
- Perfect for **UI state** (auto-updates when DB changes).

👉 Example:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getAllUsers(): Flow<List<User>>
}
```

Usage:

```
viewModelScope.launch {
    userDao.getAllUsers().collect { users ->
        println("User count: ${users.size}")
    }
}
```

✅ **Why useful?**

- No need to re-query manually.
- Automatically updates UI → great for MVVM + Jetpack Compose.

🔷 3. Flow vs LiveData in Room

- **Old way:**
- `@Query("SELECT * FROM users")`
- `fun getAllUsers(): LiveData<List<User>>`

  LiveData observes changes.

- **Modern way:**
- `@Query("SELECT * FROM users")`
- `fun getAllUsers(): Flow<List<User>>`

  Flow is preferred → works with Coroutines, more flexible.

🔷 4. Real-Life Example

```
class UserRepository(private val userDao: UserDao) {
    val usersFlow: Flow<List<User>> = userDao.getAllUsers()

    suspend fun insert(user: User) = userDao.insertUser(user)
}

class UserViewModel(private val repo: UserRepository) : ViewModel() {
    val users = repo.usersFlow.stateIn(
        viewModelScope,
        SharingStarted.Lazily,
        emptyList()
    )
}
```

🔷 5. Solution (When to Use What)

- **Use `suspend` functions** in DAO → for **one-shot queries/inserts**.
- **Use Flow** in DAO → for **observing continuous updates**.
- **Combine both** → Insert new data (`suspend`), observe all (`Flow`).

✅ Interview-Style Answer

“Room has first-class support for Coroutines and Flow. We can mark DAO methods as `suspend`, so Room runs them off the main thread automatically. For continuous updates, DAO methods can return `Flow`, which automatically emits new values when the table changes. This makes it easy to build reactive UIs with MVVM and Compose. Compared to LiveData, Flow is more flexible and coroutine-friendly.”

TypeConverters

1. What are TypeConverters?

- By default, Room only supports **primitive types** (Int, String, Boolean, etc.).
- If you use a **custom type** (e.g., `Date`, `List<String>`, `enum class`) in your `@Entity`, Room won't know how to store it in SQLite.
- **TypeConverters** tell Room **how to convert custom types → supported database types (and back)**.

👉 Example: Store a `Date` object as a `Long` (timestamp).

🔷 2. Creating a TypeConverter

Example: Converting `Date ↔ Long`

```
class Converters {
    @TypeConverter
    fun fromTimestamp(value: Long?): Date? {
        return value?.let { Date(it) }
    }

    @TypeConverter
    fun dateToTimestamp(date: Date?): Long? {
        return date?.time
    }
}
```

🔷 3. Registering TypeConverters

Option 1: At Database Level

```
@Database(entities = [User::class], version = 1)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Option 2: At Entity Level

```
@Entity
@TypeConverters(Converters::class)
data class User(
    @PrimaryKey val id: Int,
    val name: String,
    val birthDate: Date
```

```
)
```

🔷 4. Example with `List<String>`

Entity:

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val name: String,
    val hobbies: List<String>
)
```

Converter:

```
class Converters {
    @TypeConverter
    fun fromList(value: List<String>?): String? {
        return value?.joinToString(",")
    }

    @TypeConverter
    fun toList(value: String?): List<String>? {
        return value?.split(",")
    }
}
```

🔷 5. Rules

- Always **annotate functions with `@TypeConverter`**.
- Converters should be **pure functions** (no side effects).
- Store data in a SQLite-supported type (Int, Long, Float, String, ByteArray).

🔷 6. Solution (When to Use TypeConverters)

👉 Use TypeConverters when:

- You have a **custom data type** not supported by SQLite.
- Examples: `Date`, `LocalDateTime`, `List<String>`, `enum class`, custom objects.

👉 Don’t use TypeConverters when:

- The type is already supported by SQLite.

✅ Interview-Style Answer

“Room only supports primitive data types. If we want to store custom types like Date, enums, or lists, we use **TypeConverters**. A TypeConverter defines how to convert a custom type into a supported type (like Date ↔ Long, or List ↔ String). We annotate the converter functions with `@TypeConverter` and register them at the Database or Entity level. This allows Room to automatically convert data when reading or writing.”

Migration

1. What is Migration in Room?

- A **Migration** lets you update the **database schema** (tables, columns, indexes) when you release a new version of your app.
- Without migration, if schema changes → **Room throws an exception** (`IllegalStateException`).
- Migrations preserve **existing user data** while updating schema.

👉 Example: Adding a new column `age` to `users` table.

🔷 2. Basic Setup

Room requires a `version` in the `@Database` annotation.

```
@Database(entities = [User::class], version = 2) // incremented from 1 → 2
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

🔷 3. Writing a Migration

You define a `Migration` object from oldVersion → newVersion.

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE users ADD COLUMN age INTEGER NOT NULL
DEFAULT 0")
    }
}
```

🔷 4. Adding Migration to Database Builder

```
val db = Room.databaseBuilder(
    context,
    AppDatabase::class.java,
    "my_database"
).addMigrations(MIGRATION_1_2)
 .build()
```

🔷 5. Multiple Migrations

If schema evolves across versions (e.g., v1 → v2 → v3), you can chain multiple migrations:

```
.addMigrations(MIGRATION_1_2, MIGRATION_2_3)
```

🔷 6. AutoMigrations (AndroidX Room 2.4+) 🚀

- Room can **automatically handle simple migrations** (adding/removing columns, renaming).
- Annotate with `@AutoMigration`.

```
@Database(
    entities = [User::class],
    version = 3,
    autoMigrations = [
        AutoMigration (from = 2, to = 3)
    ]
)
abstract class AppDatabase : RoomDatabase()
```

👉 Use AutoMigration for simple schema changes.
👉 Use manual Migration for complex cases (data transformations).

🔷 7. Fallback Option

For debugging / non-critical data apps, you can skip migration with:

```
.fallbackToDestructiveMigration()
```

⚠️ This will **wipe existing data** instead of migrating → ❌ avoid in production apps.

🔷 8. Solution (When to Use What)

👉 **Migration (manual)** → when schema changes must preserve user data.
👉 **AutoMigration** → for simple schema changes (Room 2.4+).
👉 **Fallback** → only in test/dev apps where data can be discarded.

✅ Interview-Style Answer

“In Room, a Migration updates the database schema when the version number changes. We define a Migration object with oldVersion → newVersion and use `execSQL` to modify tables, preserving existing user data. From Room 2.4+, simple changes can be handled with `@AutoMigration`. If we don’t provide a migration, Room throws an exception, unless we use `fallbackToDestructiveMigration`, which wipes the data but isn’t recommended for production.”

Android Studio and ADB

1. Android Studio

- Official **IDE for Android** (by Google, built on JetBrains IntelliJ).
- Provides everything needed to **develop, test, debug, and build Android apps**.

Key Features

- **Code Editor** → Java, Kotlin, XML with IntelliSense.
- **Layout Editor** → Drag-and-drop UI builder.
- **Gradle Build System** → Dependency management, build variants.
- **Profiler** → Monitor CPU, memory, network usage.
- **Logcat** → System logs, debugging output.
- **Emulator** → Run apps without a physical device.
- **Lint & Inspections** → Catch errors and performance issues.

👉 **Interview Tip:** Be ready to explain where you check logs (Logcat), how you run builds (Gradle), and how to debug (breakpoints, profiler).

🔷 2. ADB (Android Debug Bridge)

- A **command-line tool** that lets you communicate with Android devices (emulator or physical).
- Comes with the Android SDK (bundled in Android Studio).
- Used for **installing apps, debugging, running shell commands, transferring files**.

Common ADB Commands

```
adb devices              # List connected devices/emulators
adb install app.apk      # Install APK on device
adb uninstall package    # Uninstall app
adb logcat               # View real-time logs
adb shell                # Open device shell
adb push local remote    # Copy file to device
adb pull remote local    # Copy file from device
adb reboot               # Restart device
```

👉 Example:

```
adb install myapp-debug.apk
adb logcat | grep MyApp
```

🔷 3. Android Studio vs ADB

Feature	Android Studio (IDE)	ADB (Command Line)
Usage	Development, coding, design, debugging	Device management, low-level debug
UI	GUI (rich interface)	CLI (terminal-based)
Best For	Writing & testing apps	Fast device control, automation
Example	Set breakpoint, debug app	`adb logcat` to check crash logs

🔷 4. Solution (When to Use What)

👉 Use **Android Studio** for:

- Writing code, designing layouts, debugging with breakpoints, profiling.

👉 Use **ADB** for:

- Installing/uninstalling APKs quickly.
- Checking logs (`adb logcat`).
- Automating CI/CD build pipelines.
- Debugging device-only issues (even without Android Studio).

✅ Interview-Style Answer

“Android Studio is the official IDE for Android development, providing tools like code editor, layout builder, Gradle build system, and profiler. ADB, on the other hand, is a command-line tool that allows developers to communicate with devices directly—for example, installing APKs, checking logs with `adb logcat`, or running shell commands. While Android Studio is GUI-based and best for development, ADB is lightweight, fast, and often used for debugging and automation.”

Debugging crashes with stacktrace

🔷 1. What is a Stacktrace?

- A **stacktrace** shows the sequence of method calls that led to a crash (exception).
- It's printed in **Logcat** or via `adb logcat`.
- Helps identify **what exception occurred** and **where in the code it happened**.

🔷 2. Example Crash Log

```
E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.example.myapp, PID: 12345
    java.lang.NullPointerException: Attempt to invoke virtual method
        'int java.lang.String.length()' on a null object reference
        at com.example.myapp.ui.MainActivity.onCreate(MainActivity.kt:25)
        at android.app.Activity.performCreate(Activity.java:8000)
        at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3449)
```

🔷 3. How to Read This

1. **Exception type** → `java.lang.NullPointerException`
   👉 Meaning: You tried to use an object that is `null`.
2. **Message** → `Attempt to invoke virtual method 'int String.length()' on a null object`
   👉 Code tried to call `.length()` on a `null` string.
3. **Important line** →
4. `at com.example.myapp.ui.MainActivity.onCreate(MainActivity.kt:25)`

   👉 Crash occurred in **MainActivity.kt, line 25**.

🔷 4. Debugging Steps

1. Open file **MainActivity.kt** at **line 25**.
2. Check which variable might be `null`.
3. `val name: String? = null`
4. `println(name.length) //` ❌ `Crash here`
5. Fix by handling null safely:

```
6. println(name?.length ?: 0) // ✅ Safe call with default
```

🔷 5. Using ADB to Debug

If you don't have Android Studio open, you can grab stacktrace via:

```
adb logcat *:E
```

Or filter for your app:

```
adb logcat | grep com.example.myapp
```

🔷 6. Types of Common Crashes

- **NullPointerException** → Accessing `null` variable.
- **IndexOutOfBoundsException** → Invalid list/array index.
- **IllegalStateException** → Wrong lifecycle state.
- **NetworkOnMainThreadException** → Doing network on main thread.
- **ClassCastException** → Wrong type cast.

🔷 7. Best Practices for Debugging Crashes

✅ Always start from **topmost app package line** (ignore framework lines like `ActivityThread`).
✅ Use **breakpoints** in Android Studio to reproduce the issue.
✅ Add **try-catch + logging** for critical operations.
✅ Use **Crashlytics / Firebase** for real-world crash reports.

✅ Interview-Style Answer

"When debugging crashes, I first check the **exception type** and the **line number** in the stacktrace. For example, a NullPointerException in `MainActivity.kt:25` means I tried to call a method on a null object there. I use `adb logcat` or Android Studio Logcat to capture the stacktrace, then trace back to the code and fix it, often by adding null checks or correcting lifecycle issues. In production, I rely on tools like Firebase Crashlytics to capture stacktraces remotely."

Profiler Tools

1. What are Profiler Tools?

- Android Studio provides **Profiler tools** to monitor app performance in real-time while running on an emulator or device.
- They help debug issues like:
  - High CPU usage ⚡
  - Memory leaks 🧠
  - Network bottlenecks 🌐
  - Battery drain 🔋

🔷 2. Types of Profilers

🟢 1. CPU Profiler (Performance issues)

- Shows **method traces, thread activity, CPU usage**.
- Helps identify heavy methods (long loops, recursion, unoptimized code).
- Modes: **Sampled**, **Instrumented**, **System Trace**.

👉 Example: Detecting an infinite loop or heavy bitmap decoding on UI thread.

🟢 2. Memory Profiler (Leaks & GC)

- Monitors **heap memory usage** in real-time.
- Shows **Garbage Collection (GC) events**.
- Can capture **heap dumps** → find objects that are not released (memory leaks).

👉 Example: Activity not garbage collected → memory leak.

🟢 3. Network Profiler (API calls)

- Shows **network requests (REST, WebSockets)**.
- Displays request/response size, time taken.
- Helps detect **slow APIs, unoptimized payloads, unnecessary calls**.

👉 Example: Repeated network calls caused by LiveData/Flow re-collection.

🟢 4. Energy Profiler (Battery drain)

- Shows **CPU, GPS, network, and sensor usage**.
- Detects **wakelocks** or **frequent background tasks**.

👉 Example: Location updates running every second instead of 10 minutes.

🔷 3. How to Use Profiler

1. Run app on **emulator or device**.
2. In Android Studio → **View > Tool Windows > Profiler**.
3. Select device + process.
4. Choose CPU / Memory / Network / Energy tab.
5. Record traces or take heap dumps as needed.

🔷 4. Real-Life Debugging Examples

- **CPU Profiler** → Found a heavy `for` loop running on the main thread → moved to `Dispatchers.Default`.
- **Memory Profiler** → Activity leaked because of a static reference → fixed with weak references.
- **Network Profiler** → Large JSON payload → optimized with pagination.
- **Energy Profiler** → Location updates drained battery → changed interval to `FusedLocationProvider` with 10 mins.

🔷 5. Solution (When to Use Which)

- **CPU Profiler** → app feels slow, UI janky.
- **Memory Profiler** → OutOfMemoryError, app keeps growing in memory.
- **Network Profiler** → APIs feel slow, app hangs on API calls.
- **Energy Profiler** → App drains battery fast.

✅ Interview-Style Answer

“Profiler tools in Android Studio help analyze performance. The CPU Profiler shows thread/method execution to catch heavy work on the main thread. The Memory Profiler tracks heap usage, garbage collection, and memory leaks. The Network Profiler monitors API calls, payload sizes, and latency. The Energy Profiler shows power-hungry components like GPS and wakelocks. I typically use these during debugging to ensure the app runs smoothly, doesn’t leak memory, and doesn’t drain the battery.”

Security

1. Common Android Security Risks

- **Storing sensitive data insecurely** (API keys, passwords in plain text).
- **Unencrypted network calls** (HTTP instead of HTTPS).
- **Weak authentication** (no biometrics, no strong session handling).
- **Exported components** (`Activity`, `Service`, `BroadcastReceiver` accessible to other apps unintentionally).
- **SQL Injection / WebView injection** (unsafe input handling).
- **Reverse engineering** (APKs can be decompiled with tools like Apktool, JADX).

🔷 2. Best Practices in Android Security

🔑 Secure Data Storage

- ❌ Don't store passwords/API keys in `BuildConfig` or SharedPreferences.
- ✅ Use **EncryptedSharedPreferences** or **Jetpack Security Crypto library**.
- ✅ For tokens → use **Android Keystore System**.

```
val masterKey = MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build()

val prefs = EncryptedSharedPreferences.create(
    context,
    "secure_prefs",
    masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
```

🌐 Network Security

- Always use **HTTPS** (TLS).
- Enable **Network Security Config** for domain pinning.

```
<network-security-config>
    <domain-config cleartextTrafficPermitted="false">
        <domain includeSubdomains="true">example.com</domain>
    </domain-config>
</network-security-config>
```

- Validate SSL certificates.

🔐 Authentication

- Use **OAuth2 / JWT tokens** (never store raw passwords).
- Use **BiometricPrompt API** for fingerprint/face unlock.

📦 App Components Security

- By default, set components **non-exported** unless needed.

```
<activity android:name=".SecretActivity"
    android:exported="false" />
```

- Protect `BroadcastReceiver` with permissions.

🗄️ Database Security

- Use **SQLCipher** or **Room with Encrypted DB** for sensitive data.
- Sanitize inputs → prevent SQL injection.

🌐 WebView Security

- Disable JavaScript if not needed.
- Prevent file access:

```
webView.settings.allowFileAccess = false
```

- Use `https` content only.

🛡️ Obfuscation & Reverse Engineering Protection

- Enable **ProGuard/R8** → obfuscates and shrinks code.

```
minifyEnabled true
proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
rules.pro'
```

- Use **NDK (native code)** for sensitive logic.
- Avoid hardcoding secrets in APK.

🛠️ Tools for Security

- **ProGuard/R8** → obfuscation.
- **SafetyNet / Play Integrity API** → detect rooted devices, tampering.
- **Firebase App Check** → verify requests come from your app.

🔷 3. Interview-Style Answer

“Android apps face risks like insecure data storage, exposed components, and reverse engineering. To mitigate this, I use EncryptedSharedPreferences or Keystore for sensitive data, HTTPS with Network Security Config for network calls, and ensure activities/services are not exported unless required. I also enable ProGuard/R8 for code obfuscation, sanitize SQL inputs, and secure WebViews. For authentication, I rely on OAuth2/JWT and use biometrics when possible. Together, these practices make the app much harder to exploit.”

Api key management

Why not hardcode API keys?

❌ If you put API keys directly in code (`const val API_KEY = "12345"`), they get compiled into the APK.

- Anyone can **decompile the APK** (via Apktool, JADX) and steal the keys.
- This leads to **API abuse**, quota exhaustion, or security breaches.

🔷 2. Recommended Approaches

✅ Option 1: BuildConfig Fields (Compile-time, safer than inline)

- Define in `gradle.properties` (outside version control):
- `API_KEY="your_real_api_key_here"`
- Pass it to `BuildConfig` in `app/build.gradle`:
- `android {`
- `    defaultConfig {`
- `        buildConfigField "String", "API_KEY", "\"${API_KEY}\""`
- `    }`
- `}`
- Access in code:
- `val apiKey = BuildConfig.API_KEY`

⚠️ Still retrievable from APK (not 100% secure), but **better than hardcoding in code**.

✅ Option 2: GitHub Actions / CI Secrets

- Store API keys in **GitHub Actions secrets** (or Jenkins credentials).
- Inject them into `local.properties` or `BuildConfig` at build time.

```
- name: Set API Key
  run: echo "API_KEY=${{ secrets.API_KEY }}" >> local.properties
```

✅ Option 3: NDK / Native Layer

- Store key in **C/C++ code** (NDK).
- Harder (but not impossible) to extract.

```
extern "C"
JNIEXPORT jstring JNICALL
Java_com_example_App_getApiKey(JNIEnv* env, jobject) {
    return env->NewStringUTF("your_api_key_here");
}
```

Called in Kotlin:

```
System.loadLibrary("native-lib")
external fun getApiKey(): String
```

✅ Option 4: Remote Config (Best for Security)

- Don't ship keys at all → **fetch from a secure server** at runtime.
- Use Firebase Remote Config or your own backend.
- This way, even if APK is decompiled, key is not exposed.

🔷 3. Best Practices Summary

✔️ Never commit API keys to GitHub.
✔️ Use `.gitignore` for `local.properties`.
✔️ Use **BuildConfig** for environment-based API keys.
✔️ For sensitive keys, keep them on **server-side** (not in APK).
✔️ Obfuscate with **ProGuard/R8** if absolutely needed.

✅ Interview-Style Answer

"I never hardcode API keys into my Android code. Instead, I use `gradle.properties` and `BuildConfig` to inject keys at build time, so they don't live in version control. For CI/CD, I store keys in GitHub Actions or Jenkins secrets and pass them into the build. For highly sensitive keys, I either move them to a secure backend or load them via Firebase Remote Config. This ensures that even if the APK is decompiled, the keys are not easily exposed."

Obfuscation with Progaurd

What is ProGuard / R8?

- **ProGuard** = Java bytecode shrinker & obfuscator.
- **R8** = Newer tool (default since Android Gradle Plugin 3.4+). Combines:
  - Shrinking (removes unused code).
  - Obfuscation (renames classes/methods/fields to meaningless names).
  - Optimization (improves bytecode).
  - Preverification (for Java ME).

👉 Purpose:

- Makes APK smaller.
- Harder to **reverse engineer** (e.g., from `MySecretClass → a`).

🔷 2. Enabling ProGuard / R8

In `app/build.gradle`:

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile(
            'proguard-android-optimize.txt'
        ), 'proguard-rules.pro'
    }
}
```

- `minifyEnabled true` → enables ProGuard/R8.
- `shrinkResources true` → removes unused resources (images, XML, etc.).
- `proguard-rules.pro` → custom rules file.

🔷 3. Example ProGuard Rules

Keep a class (prevent obfuscation)

```
-keep class com.example.api.** { *; }
```

Keep class members with annotations

```
-keepclassmembers class * {
    @com.google.gson.annotations.SerializedName <fields>;
}
```

Keep native methods

```
-keepclasseswithmembernames class * {
    native <methods>;
}
```

🔷 4. Example – Before & After Obfuscation

✅ Original:

```
public class UserManager {
    public String getUserName() {
        return "Manav";
    }
}
```

❌ After R8 obfuscation:

```
class a {
    String a() {
        return "Manav";
    }
}
```

👉 Harder for attackers to understand.

🔷 5. Best Practices

- Always test release builds (ProGuard may strip required classes).
- Keep model classes used by **Gson/Moshi/Room** (they rely on reflection).
- Add `-keep` rules for libraries like Retrofit, Glide, Firebase.
- Use **mapping.txt** (generated in `app/build/outputs/mapping/release/`) to de-obfuscate crash reports.

🔷 6. Interview-Style Answer

“ProGuard/R8 is a tool that shrinks, optimizes, and obfuscates Android apps. I enable it in release builds with `minifyEnabled true`. It renames classes and methods to make reverse engineering harder and removes unused code/resources to reduce APK size. For example,

`UserManager.getUserName()` might become `a.a()`. Since reflection-based libraries like Gson need original names, I add `-keep` rules in `proguard-rules.pro`. I also use `mapping.txt` to de-obfuscate stack traces from crashes."

Flow:
👨‍💻 Original Code → ⚙️ ProGuard/R8 (minify + shrink + optimize) → 📦 Obfuscated APK (methods renamed) → 📝 mapping.txt (for debugging).

👉 Key takeaways:

- Makes APK **smaller + harder to reverse engineer**.
- Always keep **mapping.txt** to de-obfuscate crash logs.
- Add `-keep` rules for reflection-based libs (e.g., Gson, Room).

ProGuard / R8 Basics

Q1. What is ProGuard / R8?
👉 A tool for **shrinking, obfuscating, and optimizing** Android apps.

- ProGuard = older tool.
- R8 = newer, default since Android Gradle Plugin 3.4 (faster, integrated).

Q2. What does obfuscation mean?
👉 Renaming classes, methods, and variables into meaningless short names.
Example:
`UserManager.getUserName()` → `a.a()`.
This makes reverse engineering harder.

Q3. What is shrinking?
👉 Removes unused classes, methods, and resources → smaller APK size.

Q4. Where do you enable ProGuard/R8?
👉 In `app/build.gradle`:

```
release {
   minifyEnabled true
   shrinkResources true
   proguardFiles getDefaultProguardFile(
```

```
        'proguard-android-optimize.txt'
    ), 'proguard-rules.pro'
}
```

🔷 Rules & Configuration

Q5. Why do we need `proguard-rules.pro`?
👉 To **keep** classes or methods from being removed/renamed (especially for reflection-based libraries like Gson, Room, Retrofit).

Example Rule:

```
-keep class com.example.models.** { *; }
```

🔷 Debugging

Q6. What is `mapping.txt`?
👉 A file generated after obfuscation that maps **obfuscated names → original names**.

- Needed to **read crash logs**.
- Found in: `app/build/outputs/mapping/release/mapping.txt`.

Q7. What happens if you lose `mapping.txt`?
👉 You can't de-obfuscate crash stack traces → debugging becomes very hard.

🔷 Best Practices

Q8. When should you enable ProGuard/R8?
👉 Always in **release builds** (not debug).

Q9. Does ProGuard secure API keys?
👉 No — obfuscation only makes code harder to read, but API keys can still be extracted.

Q10. What's the main benefit for users?
👉 Smaller APKs and slightly better performance.

🔷 ProGuard vs R8

Feature ⚡	ProGuard 🟠 (Older)	R8 🟢 (Newer, Default)
Introduction	Older obfuscation tool (pre-AGP 3.4)	Default since Android Gradle Plugin 3.4
Function	Shrinking + Obfuscation only	Shrinking + Obfuscation + Optimization (in one step)
Integration	Separate tool (runs after build)	Integrated into D8 dexer (build step)
Performance	Slower (two-step process)	Faster (single-step process)
Rules File	`proguard-rules.pro`	Same rules syntax (`proguard-rules.pro`)
Output Size	Good APK size reduction	Better, smaller APK size
Future Support	Legacy, not actively improved	Actively maintained & recommended
Debugging	Needs `mapping.txt`	Also generates `mapping.txt` (same usage)

Secure Shared Preference

1. Problem with Normal SharedPreferences

- `SharedPreferences` stores key–value pairs in plain text XML (`/data/data/<package>/shared_prefs/`).
- If the device is rooted or backup is exposed, data can be **read easily**.
- ❌ Not suitable for sensitive data (tokens, passwords, API keys).

🔷 2. Solution → EncryptedSharedPreferences

- Part of **Jetpack Security (crypto library)**.
- Automatically encrypts:
  - Keys (AES256-SIV).
  - Values (AES256-GCM).
- Uses **Android Keystore** to manage master keys.
- Safe for **tokens, session IDs, user data**.

🔷 3. How to Use EncryptedSharedPreferences

Add Dependency

```
dependencies {
    implementation "androidx.security:security-crypto:1.1.0-alpha06"
}
```

Create Encrypted Preferences

```
import androidx.security.crypto.EncryptedSharedPreferences
import androidx.security.crypto.MasterKey

// Create or get a Master Key stored in Android Keystore
val masterKey = MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build()
```

```
// Create EncryptedSharedPreferences
val encryptedPrefs = EncryptedSharedPreferences.create(
    context,
    "secure_prefs",  // file name
    masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)

// Store data securely
encryptedPrefs.edit().putString("token", "abc123").apply()

// Retrieve data securely
val token = encryptedPrefs.getString("token", null)
```

✅ Keys & values are encrypted before being written to disk.

🔷 4. Under the Hood

- Uses **MasterKey** stored in **Android Keystore**.
- Every read/write is automatically encrypted/decrypted.
- Even if `/shared_prefs/secure_prefs.xml` is extracted, data looks like:

```
<map>
    <string name="encrypted_key">base64_random</string>
</map>
```

🔷 5. Best Practices

- Use for **tokens, session data, user credentials**.
- Avoid storing **long-term secrets** (like API keys → keep server-side).
- Use `.apply()` instead of `.commit()` for async saves.
- Combine with **BiometricPrompt** if extra user protection is needed.

✅ Interview-Style Answer

“Normal SharedPreferences store values in plain XML and can be read on rooted devices. For sensitive data, I use Jetpack Security’s EncryptedSharedPreferences. It uses the Android Keystore to create a MasterKey and encrypts both keys and values with AES256. For example, when I save a token with `encryptedPrefs.putString("token", "abc123")`, it gets stored encrypted on disk. This way, even if someone extracts the preferences file, they cannot read the data.”

Network security config

1. Problem

- By default, Android **allowed cleartext traffic (HTTP)**.
- This is **insecure** because data can be intercepted (Man-in-the-Middle attacks).
- Since **Android 9 (API 28)** → cleartext HTTP traffic is blocked by default.

🔷 2. Solution → Network Security Config

- Lets you define **security rules in XML** for network connections.
- Can enforce:
  ✅ Only HTTPS allowed
  ✅ Certificate pinning
  ✅ Custom trusted CAs

🔷 3. Enforcing HTTPS Only

Step 1: Create `network_security_config.xml`

📂 `res/xml/network_security_config.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <!-- Block all cleartext (HTTP) traffic -->
    <base-config cleartextTrafficPermitted="false">
        <trust-anchors>
            <!-- Use system default CAs -->
            <certificates src="system" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

Step 2: Reference in `AndroidManifest.xml`

```
<application
    android:name=".MyApp"
    android:networkSecurityConfig="@xml/network_security_config"
    android:usesCleartextTraffic="false"
    ... >
</application>
```

🔷 4. Allow Exceptions (if needed)

👉 Example: Allow HTTP for dev server, but enforce HTTPS for production.

```
<network-security-config>
    <base-config cleartextTrafficPermitted="false" />

    <!-- Exception: allow HTTP for dev server -->
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">dev.example.com</domain>
    </domain-config>
</network-security-config>
```

🔷 5. Certificate Pinning (Extra Security)

You can pin certificates to prevent MITM even if system CA is compromised.

```
<network-security-config>
    <domain-config>
        <domain>secure.example.com</domain>
        <pin-set expiration="2026-01-01">
            <pin digest="SHA-256">base64-hash-of-cert</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

🔷 6. Best Practices

- Always use **HTTPS** in production.
- Use **certificate pinning** for critical apps (banking, payments).
- For dev environments, explicitly allow HTTP only if needed.

✅ Interview-Style Answer

“Network Security Config is an XML file introduced in Android 7+ to configure network security without changing app code. I use it to block all cleartext HTTP traffic by setting `cleartextTrafficPermitted=false`, which enforces HTTPS-only communication. If needed, I can allow exceptions for dev servers and even use certificate pinning for critical domains. I reference the config in the AndroidManifest under the `<application>` tag. This ensures secure networking and prevents accidental insecure requests.”

Authentication and Server Interactions

1. Types of Authentication in Android

1️⃣ API Key Authentication (Basic)

- Simple, often used for public APIs.
- ❌ Weak — key can be leaked if stored in app.
- ✅ Better: Use API key + restrictions (domain, IP).

2️⃣ Token-Based Authentication (Modern, Secure)

- **JWT (JSON Web Token)** or OAuth2 tokens.
- Flow:
  - User logs in → server returns **access token** + (optional) **refresh token**.
  - App uses **access token** in every request (`Authorization: Bearer <token>`).
  - When access token expires, use **refresh token** to get a new one.

👉 Example API call with Retrofit:

```
@GET("user/profile")
suspend fun getProfile(
    @Header("Authorization") token: String
): UserProfile
```

3️⃣ Session-Based Authentication

- Older model → server keeps a session ID.
- Less used in modern mobile apps (tokens preferred).

4️⃣ Social Login (OAuth2)

- Google, Facebook, Apple → handled via SDK or OAuth flow.
- Returns token → exchange for server-side session or JWT.

🔷 2. Secure Token Storage

- ❌ Don’t store in plain SharedPreferences.
- ✅ Use **EncryptedSharedPreferences** or **Android Keystore**.
- ✅ For short-lived → keep in memory (until app closes).

🔷 3. Retrofit + Token Interceptor

👉 Best practice: use an **OkHttp Interceptor** to add tokens automatically.

```
class AuthInterceptor(private val tokenProvider: TokenProvider) :
Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val token = tokenProvider.getAccessToken()
        val request = chain.request().newBuilder()
            .addHeader("Authorization", "Bearer $token")
            .build()
        return chain.proceed(request)
    }
}

// Add to OkHttp
val client = OkHttpClient.Builder()
    .addInterceptor(AuthInterceptor(tokenProvider))
    .build()
```

🔷 4. Secure Server Interactions

✅ Always use **HTTPS (TLS)** → block HTTP with **Network Security Config**.
✅ Validate SSL certificates, consider **certificate pinning** for critical apps.
✅ Never hardcode secrets (API keys) → use BuildConfig or remote config.
✅ Rate limit requests to prevent abuse.
✅ Handle **401 Unauthorized** → refresh token flow.

🔷 5. Authentication Flow Example

1. User logs in with username/password.
2. Server validates → returns JWT Access Token + Refresh Token.
3. App stores tokens securely (EncryptedSharedPreferences).
4. App uses Access Token in every API call (`Authorization: Bearer ...`).
5. If token expired → refresh with Refresh Token → get new Access Token.
6. If refresh fails → force user login again.

✅ Interview-Style Answer

“In Android apps, I usually work with token-based authentication (JWT or OAuth2). When a user logs in, the server returns an access token and optionally a refresh token. I store these securely using EncryptedSharedPreferences or the Keystore. I use an OkHttp Interceptor in Retrofit to attach the token automatically to all requests. I ensure all communication happens over HTTPS, enforce it with Network Security Config, and use refresh tokens to maintain sessions. This ensures secure and seamless authentication and server interaction.”

Dependency Injection (Hilt)

Dependency Injection (DI) in Simple Words

👉 Imagine you're making **coffee** ☕:

- You need **milk, sugar, coffee powder, water**.
- If you **make them yourself every time**, it's extra work.
- If someone **hands them to you already prepared**, you just make coffee easily.

That's **Dependency Injection**:

- Instead of **creating things yourself**, they're **given ("injected") to you** when needed.

🔷 Without DI (Normal Way)

```
class Engine {
    fun start() = "Engine started"
}

class Car {
    private val engine = Engine() // Car creates Engine itself ❌ tightly
coupled
    fun drive() = engine.start()
}
```

Problem:

- Car **depends** directly on Engine.
- If you want to switch to `ElectricEngine`, you must change Car's code.

🔷 With DI

```
class Car(private val engine: Engine) {
    fun drive() = engine.start()
}
```

Now:

- Engine is **injected** into Car.
- Car doesn't care if it's `Engine` or `ElectricEngine`.
- More flexible, testable, and maintainable. ✅

🔷 What is Hilt?

- **Hilt** is Android's official DI framework (built on top of Dagger).
- It automatically **provides and manages dependencies** for you.
- Handles **scoping** (singleton, activity-level, fragment-level).
- Integrates with **ViewModel**, **WorkManager**, **Navigation**, etc.

👉 In plain English:
Hilt is like a manager in a factory that delivers the right parts (dependencies) to the right worker (class) at the right time.

🔷 Example with Hilt

Step 1: Add Hilt dependency (Gradle)

```
implementation "com.google.dagger:hilt-android:2.51"
kapt "com.google.dagger:hilt-compiler:2.51"
```

Step 2: Enable Hilt

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

Step 3: Provide dependency

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

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

Step 4: Inject into class

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

    @Inject lateinit var retrofit: Retrofit  // 🚀 Hilt gives it
automatically

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

```
        println(retrofit.baseUrl())
    }
}
```

🔷 Why Use Hilt?

✅ Avoids manual object creation everywhere.
✅ Makes **testing easier** (you can swap dependencies).
✅ Manages **lifecycles** (singleton, activity, fragment, etc.).
✅ Reduces boilerplate compared to raw Dagger.

📌 Layman Recap

- **Normal way**: You go to market every time to buy sugar, milk, coffee → lots of effort.
- **DI (Hilt)**: A delivery guy (Hilt) always brings the right ingredients to your kitchen when you need them.

Pending Intent

What is a PendingIntent?

- A **PendingIntent** is like giving someone else (another app or Android system) **a permission slip** ✉️ that says:
  "You are allowed to run *this intent* on my app's behalf, later."

🔷 Everyday Analogy

Imagine you're at a **bank** 🏦:

- You write a cheque and sign it ✍️.
- Even if you're not present, the bank can cash it later, because it has your **permission (signature)**.
- That's exactly what a PendingIntent is → a signed cheque for an Intent.

🔷 Why PendingIntent is Needed?

Normally:

- An `Intent` runs immediately (start activity, service, etc.).

But sometimes you want:

- The **system** (not you) to run the Intent **later** (e.g., when alarm goes off, or when user taps a notification).
- The **system** doesn't have permission to act on your app directly.
- So you give it a **PendingIntent** (permission slip).

🔷 Common Use Cases

1. **Notifications**
   - When user taps a notification, system launches your activity using PendingIntent.

```
val intent = Intent(this, MainActivity::class.java)
val pendingIntent = PendingIntent.getActivity(
    this, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT
)
notificationBuilder.setContentIntent(pendingIntent)
```

7. **Alarms (AlarmManager)**
    - You schedule something in future. AlarmManager uses PendingIntent to start your broadcast/service.
8. **Widgets / App shortcuts**
    - Home screen widget button → triggers a PendingIntent.

🔷 Types of PendingIntent

- `getActivity()` → Launch an Activity later.
- `getService()` → Start a Service later.
- `getBroadcast()` → Send a broadcast later.

🔑 Key Point

- A normal Intent = "Do this action **now**."
- A PendingIntent = "Here's a signed permission to do this action **later, on my behalf**."

⚡ In one line:
PendingIntent is like giving Android a pre-signed cheque (intent) it can cash in later, even if your app is not running.

MVVM vs MVC vs MVP

1. MVC (Model – View – Controller)

- **Model** → Data & business logic (e.g., database, API).
- **View** → UI (Activity/Fragment layouts).
- **Controller** → Middleman, handles input and updates Model/View.

👉 Analogy:

- **Model** = Kitchen (cooks the food 🥗).
- **View** = Dining table (shows food).
- **Controller** = Waiter (takes orders, serves food).

⚠️ Problem in Android:

- `Activity` often becomes **both View & Controller**, so it gets **too fat** (God Activity).

🔷 2. MVP (Model – View – Presenter)

- **Model** → Same as before (data, API, DB).
- **View** → UI (Activity/Fragment, but only shows data).
- **Presenter** → Does all logic, tells View what to show.

👉 Analogy:

- **Model** = Kitchen 👨‍🍳
- **View** = TV Screen (just shows what Presenter says).
- **Presenter** = News Anchor 🎙️ (takes info from Model, presents it clearly).

✅ View is passive.
✅ Easy to test Presenter (logic separated).
⚠️ But still needs manual View ↔ Presenter contracts (a lot of boilerplate).

🔷 3. MVVM (Model – View – ViewModel) ✅ (modern Android standard)

- **Model** → Same (data, repository, DB, API).
- **View** → Activity/Fragment/Compose UI → observes data.
- **ViewModel** → Exposes **data streams (LiveData / StateFlow)**. Handles UI logic.

👉 Analogy:

- **Model** = Kitchen 🍲
- **View** = Window 👀 (just watches)
- **ViewModel** = Delivery Guy 🚴 (keeps the latest food ready, View just observes it).

✅ View just observes ViewModel → no manual calls needed.
✅ Lifecycle-aware (ViewModel survives config changes).
✅ Official recommended pattern by Google.

📌 Quick Comparison Table

Feature	MVC	MVP	MVVM
Who handles logic?	Controller (often Activity)	Presenter	ViewModel
View's role	Both UI & logic (fat)	Passive, Presenter updates	Observes ViewModel
Communication	Controller ↔ Model ↔ View	View ↔ Presenter ↔ Model	View observes ViewModel (LiveData/Flow)
Boilerplate	Low but messy	More boilerplate	Clean (with LiveData/Flow)
Android Fit	Poor (God Activity issue)	Better	Best (recommended)

⚡ In one line

- **MVC** → Waiter does everything, kitchen + dining table. (God Activity problem).
- **MVP** → Presenter is the smart guy, View is dumb (just shows).
- **MVVM** → ViewModel streams data, View observes (clean & modern).

IPC

What is IPC?

👉 **IPC = Inter-Process Communication**

- In Android, each **app runs in its own sandboxed process** (separate memory space).
- Apps normally **cannot directly access each other's memory** (for security).
- If one app/process needs to talk to another, it uses **IPC**.

🔷 Everyday Analogy

Imagine:

- Each **house** 🏠 = an app (separate process).
- You **cannot enter your neighbor's house** directly.
- To communicate, you either:
  - **Send a letter** ✉️ (intent).
  - **Call a hotline** ☎️ (binder).
  - **Use a post office service** 📦 (ContentProvider).

That's IPC → **safe communication between separate houses (processes)**.

🔷 IPC in Android

Android provides several **IPC mechanisms**:

1. **Intents (most common)**
   - Send messages between components or apps.
   - Example: Sharing a photo from Gallery → WhatsApp.
2. **Binder (low-level, core IPC in Android)**
   - Lightweight, high-performance IPC mechanism used internally by Android.
   - Example: When your app talks to the **System Service** (Location Manager, Telephony, etc.), it uses Binder.
3. **AIDL (Android Interface Definition Language)**
   - Special tool to define interfaces for IPC.
   - Used when you need **two-way communication** across processes (e.g., media player service and UI app).

4. **Messenger (Handler-based IPC)**
    - Uses Binder under the hood, but message-passing style.
    - Easier than AIDL for simple requests/responses.
5. **ContentProvider**
    - IPC for **data sharing** (structured).
    - Example: Contacts app provides phone numbers to WhatsApp.
6. **Broadcasts**
    - One-to-many communication.
    - Example: Battery low event is broadcast to all apps.

🔷 Visual

- **Intent** = Sending a letter to another house.
- **Binder/AIDL** = Direct phone line between houses.
- **ContentProvider** = Public library where multiple apps can read/write.
- **Broadcast** = Loudspeaker in the city → everyone hears it.

📌 Summary

- **IPC = how apps/processes talk in Android.**
- By default, apps are **isolated for security**.
- Android gives tools: Intents, Binder, AIDL, Messenger, ContentProvider, Broadcast.

⚡ In one line:
IPC in Android is the set of mechanisms that let apps/services (different processes) safely talk to each other.

Android IPC Mechanisms (Cheat Sheet)

IPC Mechanism	Analogy	Who talks to whom?	When to Use
Intent ✉️	Sending a letter	One app/activity → another	Start Activity, Service, or send Broadcast
PendingIntent 📝	Signed cheque (system can cash later)	App → System → back to App	Notifications, AlarmManager, Widgets
BroadcastReceiver 📢	City loudspeaker	System/App → many apps	System events (battery low, WiFi change), custom app events
ContentProvider 📚	Public library	App ↔ Other apps	Share structured data (Contacts, Media, Calendar)
Binder ☎️	Direct phone line	App ↔ System Service	Core Android IPC (Location, Telephony, etc.)
AIDL 🔗	Contract between 2 companies	App ↔ Another App (different process)	Complex two-way communication (e.g., Media player service)
Messenger 📬	Postman with letters	App ↔ Service	Simple request/response across processes, easier than AIDL

What is Social Login?

- **Social login** = Letting users **sign in with their existing social media account** (Google, Facebook, Twitter, Apple, etc.) instead of creating a new account.
- It uses **OAuth 2.0 protocol** behind the scenes.

👉 Layman analogy:
Instead of showing your **passport everywhere**, you just show a **trusted ID card** (Google/Facebook). The app trusts that ID instead of making you create a new one.

🔷 Why Use Social Login?

✅ User convenience (no need to remember new passwords).
✅ Faster onboarding (1 click → logged in).
✅ Trusted identity (Google/Facebook already verified them).
⚠️ App depends on third-party (if their login service fails, users can't log in).

🔷 How It Works (Step by Step with Google Example)

1. **User clicks "Sign in with Google"**.
2. Your app launches Google's **OAuth login screen**.
3. User authenticates with Google → Google sends back an **Access Token / ID Token**.
4. Your app sends this token to your **backend server**.
5. Server verifies the token with Google servers (to ensure it's valid & not fake).
6. Once verified → server creates a **session/JWT token** for your app.

👉 User is now logged in ✅

🔷 Android Example (Google Sign-In)

1. Add dependency

```
implementation 'com.google.android.gms:play-services-auth:20.7.0'
```

2. Configure Google Sign-In

```
val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
    .requestIdToken(getString(R.string.default_web_client_id))
    .requestEmail()
    .build()

val googleSignInClient = GoogleSignIn.getClient(this, gso)
```

3. Launch Sign-In Intent

```
val signInIntent = googleSignInClient.signInIntent
startActivityForResult(signInIntent, 1001)
```

4. Handle Result

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == 1001) {
        val task = GoogleSignIn.getSignedInAccountFromIntent(data)
        val account = task.result
        val idToken = account.idToken  // Send to backend for verification
    }
}
```

🔷 Other Social Logins

- **Facebook Login** → Facebook SDK (returns access token).
- **Twitter Login** → OAuth-based, requires developer keys.
- **Apple Sign In** → Required for iOS apps with social login.

All follow **OAuth 2.0** principle:

- Redirect user to provider → Provider authenticates → Returns token → App verifies token.

📌 Summary

- **Social Login** = Users log in with **Google/Facebook/Apple** instead of new credentials.
- Uses **OAuth 2.0** protocol.
- Benefits: Fast, secure, familiar.
- Risk: Depends on third-party availability.

⚡ In one line:
Social login is like using your trusted Google/Facebook ID card to enter a new app instead of making a new password.

Intent

What is an Intent?

👉 In Android, an **Intent** is simply a **message object** you use to ask Android to do something.

- It can be used to **start an Activity** (new screen),
- **start a Service** (background work),
- or **deliver a message** (Broadcast).

🔷 Layman Analogy

Think of an **Intent like a letter** ✉️:

- You write **what you want to do** (open Gmail, share photo, call number).
- You put the **address** (which app/activity should handle it).
- You send it → Android's **Post Office (System)** delivers it to the right app/component.

🔷 Types of Intents

1. Explicit Intent

- You specify **exactly which component** to start.
- Example: Open your own Activity.

```
val intent = Intent(this, SecondActivity::class.java)
startActivity(intent)
```

👉 Like writing a letter with the **receiver's exact name & address**.

---

2. Implicit Intent

- You say **what action you want**, Android finds the right app to handle it.
- Example: Share text, open a web page, take a photo.

```
val intent = Intent(Intent.ACTION_VIEW)
```

```
intent.data = Uri.parse("https://www.google.com")
startActivity(intent)
```

👉 Like writing “To: Any Restaurant serving pizza” instead of a specific name.

🔷 Common Uses of Intents

✅ Start a new Activity → navigate between screens.
✅ Start a Service → background tasks (e.g., download file).
✅ Send a Broadcast → notify other apps/components (e.g., battery low).
✅ Share data → open other apps (e.g., share image with WhatsApp).

🔷 Key Parts of an Intent

- **Action** → What to do (e.g., `ACTION_VIEW`, `ACTION_SEND`).
- **Data/URI** → What it applies to (e.g., web URL, contact).
- **Extras (Bundle)** → Extra data (key-value pairs).
- **Component name** → Which exact class (for explicit intent).

Example:

```
val intent = Intent(Intent.ACTION_SEND)
intent.type = "text/plain"
intent.putExtra(Intent.EXTRA_TEXT, "Hello from my app!")
startActivity(Intent.createChooser(intent, "Share via"))
```

📌 Summary

- **Intent = Message object** asking Android to do something.
- **Explicit** → start a specific component.
- **Implicit** → let Android/system find who can handle it.

⚡ In one line:
Intent is like a letter you send in Android → telling the system what you want done, and Android delivers it to the right place.

Context

🔷 What is Context?

👉 In Android, **Context is like a "handle" to the system**.
It gives your app access to:

- **System resources** (strings, images, layouts).
- **System services** (location, wifi, sensors).
- **Launching activities, services, intents**.

Without Context → your app doesn't know where it is or how to interact with Android OS.

🔷 Layman Analogy

Think of **Context like a "passport" or "identity card"** for your component.

- When you go to the bank 🏦, they ask for your ID before giving services.
- Similarly, when your app asks Android: *"Give me the LocationManager"* → Android says: *"Show me your Context (who are you) first."*

⚡ In short: **Context = the environment information your app needs to talk to the Android system.**

🔷 Types of Context

1. Application Context

- Lives as long as your app lives.
- Good for things tied to the whole app (e.g., database, shared preferences).
- Example:
- `val appContext = applicationContext`

👉 Like your **passport** – works everywhere, valid for entire journey.

---

2. Activity Context

- Tied to an Activity's lifecycle.
- Good for UI-related things (e.g., showing dialogs, inflating layouts).

- Example:
- `val intent = Intent(this, SecondActivity::class.java) // here "this" = Activity context`
- `startActivity(intent)`

👉 Like your **office ID card** – valid only in the office (Activity).

---

3. Other contexts

- **Service Context** (inside services).
- **BroadcastReceiver Context** (inside receivers).

Each component has its own "ID card" (context) to interact with Android.

🔷 Where You Use Context

✅ Start activities/services → `startActivity(intent)`
✅ Access system services → `getSystemService(Context.LOCATION_SERVICE)`
✅ Access resources → `getString(R.string.app_name)`
✅ Show Toast → `Toast.makeText(context, "Hello", Toast.LENGTH_SHORT).show()`

🔷 Key Difference

- **Application Context** → app-level, longer-lived, avoid memory leaks.
- **Activity Context** → UI-level, tied to Activity lifecycle, can leak memory if misused.

📌 Summary

- **Context = Passport/ID** that proves "who you are" to the Android system.
- Needed to access system services, resources, launch activities.
- Two main types: **Application Context** (global) & **Activity Context** (local).

⚡ In one line:
Context is the "identity" of your app/component that Android needs before giving you system services or resources.

1. What is Context in Android?

👉 **Answer:**
Context is an interface to global information about the application environment. It provides access to resources, services, and helps start activities/services.

🔷 2. How many types of Context exist?

👉 **Answer:**
Mainly **two types**:

- **Application Context** → tied to the application lifecycle (global).
- **Activity Context** → tied to the activity lifecycle (UI-related).
  (Also Service & BroadcastReceiver contexts, but they behave like Application Context mostly).

🔷 3. When should you use Application Context vs Activity Context?

👉 **Answer:**

- Use **Application Context** for long-lived, app-wide operations (DB, Retrofit, WorkManager).
- Use **Activity Context** for UI (dialogs, inflating layouts, theming).

🔷 4. What happens if you pass an Activity Context to a Singleton?

👉 **Answer:**
⚠️ **Memory leak risk!**
Because the Singleton may hold onto the Activity reference even after it's destroyed, preventing it from being garbage-collected.
👉 Solution: Use **Application Context** in Singletons.

🔷 5. Can you use Application Context to show a Dialog?

👉 **Answer:**
❌ No, because Dialog requires a **UI-related Context** (Activity Context).
If you try with Application Context → `WindowManager.BadTokenException`.

🔷 6. What happens if you start an Activity using Application Context?

👉 **Answer:**

- You must add **FLAG_ACTIVITY_NEW_TASK** flag, otherwise it will crash.

```
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
applicationContext.startActivity(intent)
```

Because only an Activity Context is tied to an existing task stack.

🔷 7. Which context does a BroadcastReceiver's `onReceive()` method provide?

👉 **Answer:**
It gives you a **Context parameter** (but not always an Activity).
It's usually a **restricted Application-like Context** → safe for background operations, but not for UI things like dialogs.

🔷 8. Is Application Context always safe to use?

👉 **Answer:**
✅ Yes for long-lived objects, services, singletons.
❌ Not for UI components (dialogs, theming, inflating XML with styles).

📌 Quick Flashcard Rules

- ✅ **Use Activity Context** for UI.
- ✅ **Use Application Context** for singletons, services, global tasks.
- ❌ Don’t keep Activity Context in long-lived objects → memory leak.
- ❌ Don’t use Application Context for UI → runtime crash.