Android Developer Interview Handbook

Your Ultimate Guide to Crack Android Interviews with Confidence

500+ Real Interview Questions with Clear Explanations

Written By

Anand Gaur

Table of Contents

1. Android Core Concepts

Q:1) What are the core building blocks of an Android application?

Android apps are built using several essential components provided by the Android framework. These components work together to handle **UI**, **background tasks**, **user interactions**, and **data sharing**.

Core Building Blocks of an Android Application:

1. Activities

- Represents a single screen with a user interface.
- Acts as the entry point for user interaction.
- Every screen in an app is usually an Activity.

2. Fragments

- A modular piece of UI that lives inside an Activity.
- Can be reused in multiple activities.
- Useful for responsive layouts (e.g., tablets vs phones).

3. Services

- Runs background operations without user interaction.
- Useful for long-running tasks like downloading files, playing music, etc.

4. Broadcast Receivers

- Listens to system-wide or app-wide broadcast messages.
- Useful for responding to events like battery low, Wi-Fi connected, etc.

5. Content Providers

- Used to share data between apps.
- Acts as a data access interface, like a database that can be queried using URI.

6. View:

- Every UI component like Button, TextView, etc.
- Created in XML or code using Jetpack Compose.

7. Layouts:

- Organize UI views. Examples: LinearLayout, ConstraintLayout, Compose Column/Row.

8. Manifest File:

- Declares all components, permissions, and metadata.
- It's like the blueprint of your app.

Q:2) What is the intent?

Intent is a messaging object used to request an action from another app component (activity, service, or broadcast receiver). It allows communication between components and even across different applications.

There are two types of intents in Android:

1) Explicit Intent

- Used to launch a **specific component** (e.g., another activity **within the same app or in another app**, if the component name is known).
- You define the **class name** or **component name** directly.

Example:

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

2) Implicit Intent

- Used to perform an action **without specifying the exact component**.
- The Android system will match the intent with the appropriate component using **intent filters**.

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

Q:3) What is the Android Application Lifecycle?

The **Android Application Lifecycle** is the process your **entire app** goes through — from the moment it starts running to the moment it's closed or killed by the system.

This lifecycle is managed by the **`Application` class**.

Key Steps in the Application Lifecycle:

`onCreate()`

- Called **once** when the app is first launched
- Best place to **initialize global things** like Firebase, logging, or any SDKs

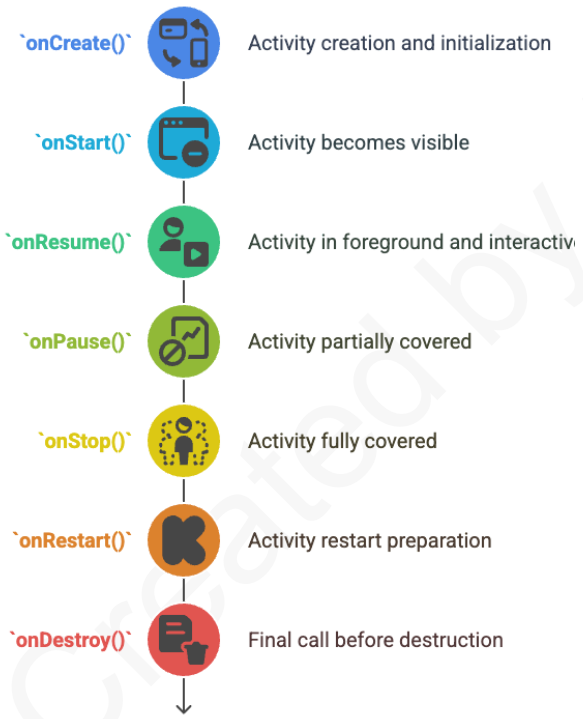
`onTerminate()`

- Called when the app is about to close (only in emulators or rare cases)
- Not reliable on real devices

`onLowMemory()` / `onTrimMemory()`

- Called when the system is running low on memory
- Use this to **free up memory** (like clearing image cache)

Activity Lifecycle Methods:



Q:4) What is the Scenario in which only onDestroy is called for an activity without onPause() and onStop()?

If **finish()** is called in the OnCreate method of an activity, the system will invoke onDestroy() method directly.

Commonly used in:

- **Notifications** – to open an activity when user taps it
- **AlarmManager** – to run something at a scheduled time
- **Broadcasts** – to send data in the future

Q:26) What are Intent Filters?

- Intent Filters are used to tell Android which intents an activity, service, or broadcast receiver can handle.
- They are defined in the **AndroidManifest.xml** file.
- The system uses intent filters to decide which component should respond to a specific intent.
- **For example**, if a user clicks on a web link, the system checks the intent filters to find an app that can handle it.
- An intent filter includes:
    - **action** – what kind of action (like view, send, etc.)
    - **category** – extra information (like default)
    - **data** – type of data (like http, tel, etc.)

Q:27) What is a BroadcastReceiver in Android?

- A **BroadcastReceiver** is a component in Android that **listens for system-wide or app-specific broadcast messages** (called Intents).
- It helps your app **respond to events**, even if your app is not currently open.
- You can use it to listen to **system events** like:
    - Phone is charging
    - Battery is low
    - Internet connectivity changes
    - Device boot completed
- You can also create and send **custom broadcasts** within your own app.
- It does **not show any UI**, but you can use it to start a service, show a notification, or launch an activity.

Q:28) What are Loaders in Android?

- **Loaders** are used to **load data in the background** from a data source (like a database or content provider).
- They were introduced in **API level 11 (Android 3.0)**.
- Loaders help to **avoid running long tasks on the main thread** (which can freeze the UI).
- They are commonly used with **CursorAdapters** to load data into list-based views.
- Loaders can **automatically reconnect to the last loaded data** after a configuration change (like screen rotation), so they prevent **duplicate queries**.

- Loaders are managed by **LoaderManager**, which handles the lifecycle.

Q:29) What are Launch Modes in Android ?

In Android, **launch modes** decide **how activities are created and managed** in the back stack when you open or reopen them.

● Standard (Default)

- A **new activity instance is always created**, even if it's already in the stack.
- You can have **multiple copies** of the same activity.

Example:
If current stack is: A → B → C
You launch **B** again with **standard**, new stack becomes:
A → B → C → B

Use Case: Chat screen in a messaging app (like WhatsApp)

- Suppose you're chatting with the same person and open their chat screen multiple times using different paths.
- Each time, a **new instance** of the chat screen (activity) is created.

● SingleTop

- Works like **standard**, but if the **same activity is already on top**, it won't create a new one.
- It will reuse the existing one by calling onNewIntent().

Example:
Stack: A → B → C
Launch **C** again with **singleTop** → still: A → B → C (no duplicate)
But if stack is A → B → C, and you launch **B** again with **singleTop**, new stack becomes:
A → B → C → B (because B was not on top)

Use Case: Notifications in Gmail

- You tap on a Gmail notification, and it opens the **Email Detail screen**.
- If the user is already on that same screen (top of stack), **no new instance is created**.
- Instead, the existing screen is updated with new data using `onNewIntent()`.

● SingleTask

- Only **one instance** of the activity exists at a time in the task.
- If it's already in the stack, all activities **above it are removed**, and that activity is reused.

Example:
Stack: A → B → C → D
Launch **B** with **singleTask** → stack becomes:
A → B (C and D are removed)

Use Case: Splash screen or Login screen

- You log into an app, and the login activity is launched with `singleTask`.
- Now after login, you go to the main dashboard.
- If you accidentally launch the login activity again (like by deep link), **all screens above it are removed**, and you're brought back to login.

● SingleInstance

- Same as **singleTask**, but the activity is launched in its **own separate task**.
- No other activities will be in this task.

Example:
Stack: A → B → C → D
Launch **B** with **singleInstance** →

- Task 1: A
- Task 2: B (in its own task)

Use Case: Video player or call screen (like Zoom, Google Meet, or YouTube PiP)

- You open a video or a call screen that should run in a **separate task**.
- No other activity should be part of this task.
- If you press Home and come back to the app, the video/call continues in the separate task.

Q:30) What is ConstraintLayout?

- ConstraintLayout is a **layout in Android** that lets you design **complex UIs without nesting** multiple layouts.
- It helps to create **flat and efficient layouts**, which means better performance.
- Similar to RelativeLayout, but **more powerful and flexible**.
- You position views by creating **constraints** between:
    - one view and another view
    - or a view and the parent layout
- It supports advanced features like:
    - Chains (for evenly spaced items)
    - Guidelines (for alignment)
    - Barriers (for dynamic layout)

Q:66) What is PeriodicWorkRequest and when to use it?

- A **PeriodicWorkRequest** is used in **WorkManager** to run background tasks **repeatedly** at a fixed time interval.
- It's ideal for work that needs to happen **regularly**, even if the app is closed or the device restarts.

Example Use Cases

You should use **PeriodicWorkRequest** for tasks like:

- Syncing app data with the server every few hours
- Uploading logs or analytics daily
- Checking for app or content updates
- Cleaning cache or temporary files at intervals

Important Rules

- The **minimum repeat interval** allowed is **15 minutes** — you cannot set it lower than that.
- WorkManager will try to run the task **as close to the interval as possible**, but **exact timing is not guaranteed** (for battery optimization).
- You can add **constraints** (like run only on Wi-Fi or when charging).

Q:67) What are the different states of Work in WorkManager?

- In **WorkManager**, every task (WorkRequest) can be in one of several **states**.
- These states help you **track progress**, **handle retries**, or **debug issues** in background work.

Work States

(A) ENQUEUED

- The work has been **added to the queue**, but has **not started yet**.
- WorkManager is waiting for **constraints** (like Wi-Fi, charging) to be satisfied.

(B) RUNNING

- WorkManager has started executing the work on a **background thread**.
- The work is currently being processed.

(C) SUCCEEDED

- Work completed **successfully**.
- WorkManager marks it as **done**.

(D) FAILED

- Work **failed permanently**.
- WorkManager will **not retry** unless you specify `setBackoffCriteria()` for retries.

(E) BLOCKED

- Work is **blocked because it depends on other work** that has not finished yet.
- Example: Work B depends on Work A — Work B will be BLOCKED until Work A finishes.

(F) CANCELLED

- Work was **manually cancelled** using `WorkManager.cancelWorkById()` or related methods.

Key Points

- Use **LiveData or Flow** to observe **real-time state changes** of work.
- WorkManager handles **automatic retries** for failed work if configured.
- These states help in **debugging** and **updating UI** about background task progress.

Q:68) What are Constraints in WorkManager and how to use them?

Constraints control when the work should run.

Example: Only run when the device is charging and connected to the network.

2. OOPS Concepts

Q:1) What is a Class and Object in Android?

Class

- A **class** is like a **blueprint** or **template** for creating objects.
- It defines **properties (variables)** and **behaviors (functions/methods)**.
- In Android (Kotlin/Java), you use classes to **structure your app**.

Key Points:

- Defines **what an object will have and do**.
- Doesn't occupy memory by itself until an **object** is created.

Object

- An **object** is a **real instance** of a class.
- It occupies **memory** and can **use the properties and functions** defined in the class.

Key Points:

- Object is the **actual entity** created from the class blueprint.
- You can create **multiple objects** from the same class, each with **different data**.

Q:2) What are Primary and Secondary Constructors in Kotlin?

Primary Constructor

- The **main constructor** of a class.
- Defined **in the class header**.
- Can directly initialize **properties**.

Usage in Android:

- Pass **data directly when creating an object**.

Key Points:

- There can be **only one primary constructor**.
- Can include **`init` block** for additional initialization:

Secondary Constructor

- **Optional additional constructors** for different ways to create an object.
- Defined **inside the class body** with a `constructor` keyword.
- Must **delegate to the primary constructor** (if primary exists) using `: this(...)`.

Usage in Android:

- Useful when you want **flexible object creation** in different scenarios.

Key Points:

- You can have **multiple secondary constructors**.
- Helps when **default values or alternative initialization** is needed.

Q:3) Explain Inheritance in Android with an Example

- **Inheritance** is an **OOP concept** where one class (child/subclass) **inherits properties and behaviors** of another class (parent/superclass).
- Helps **reuse code**, **reduce duplication**, and **create hierarchical relationships**.

How It Works in Android

- Android apps are built using **classes**, so inheritance is common:
  - **Activities and Fragments** extend `AppCompatActivity` or `Fragment`.
  - **Custom Views** extend `View` or `TextView`.
  - **Adapters** can extend `RecyclerView.Adapter`.

Q4: What is Polymorphism in Android?

- **Polymorphism** is an **OOP concept** that allows an object to take **many forms**.

3. Kotlin Concepts

Q:1) What are the main features of Kotlin?

- **Concise**: Less boilerplate than Java
- **Null Safety**: Built-in null checks
- **Extension Functions**: Add functions to existing classes
- **Coroutines**: Lightweight concurrency
- **Smart Casts**: No need for explicit casting after type check
- **Data Classes**: Auto-generate equals(), hashCode(), toString(), etc.
- **Default & Named Arguments**
- **Higher-order functions & Lambdas**

Q:2) What is the difference between `val`, `var`, and `const` in Kotlin?

In Kotlin, `val` and `var` are used to declare variables, but they behave differently:

1. `var` (Variable)

- A **mutable** variable.
- You **can change** its value after it's assigned.
- Stored in **memory at runtime**.

2. `val` (Value)

- An **immutable** variable (like `final` in Java).
- You **can assign only once**.
- Value is also stored **at runtime**, but can't be reassigned.

3. `const val` (Constant)

- A **compile-time constant**.
- Can only be used **with top-level properties** or **inside objects or companion objects**.
- **Must be of a primitive type or String**, and value must be known **at compile time**.

```
val name = "Anand" // Cannot be changed later
var age = 30       // Can be updated
age = 31
```

Q:3) What are null safety features in Kotlin?

Kotlin eliminates `NullPointerException` (NPE) by making all types **non-nullable** by default.

Types:

- **Non-nullable:** `var name: String = "Anand"` → cannot hold null
- **Nullable:** `var name: String? = null` → can hold null

Safe Operations:

- **Safe call `?.`**: Skips execution if the object is null.
- **Elvis `?:`**: Provide default value if null.
- **Not-null Assertion `!!`**: Throws Null Pointer Exception if value is null.
- **Safe Cast `as?`**: Returns null instead of throwing ClassCastException.

Q:4) What is a `data class` in Kotlin?

A `data class` is a special class made specifically for storing data. It automatically gives you useful methods like:

- `toString()` – so you can print the object easily
- `equals()` and `hashCode()` – to compare objects or use in HashMap/Set
- `copy()` – to create a new object with some properties changed
- `componentN()` – to access values using destructuring (like `val (a, b) = obj`)

Syntax:

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

4. Android Architecture

Q:1) What is Android Architecture?

- It defines a way to structure code into layers.
- Helps separate UI, data, and business logic.
- Makes the code easy to maintain, test, and scale.

Q:2) What is MVVM Architecture?

- MVVM stands for **Model-View-ViewModel**.
- **Model**: Manages data (e.g., from API or database).
- **View**: UI layer (Activity, Fragment, or Compose).
- **ViewModel**: Holds UI data and business logic, survives screen rotation.
- Helps reduce code in Activity/Fragment.

Q:3) What is ViewModel?

- Part of Android Architecture Components.
- Stores UI-related data across configuration changes.
- Provides data to the View using LiveData or StateFlow.
- Doesn’t contain references to View (Activity/Fragment).

Q:4) What is LiveData?

- Lifecycle-aware observable data holder.
- UI observes LiveData to get automatic updates.
- Prevents memory leaks as it only updates when the UI is active.

Q:5) What is the difference between LiveData and StateFlow?

- **LiveData** is lifecycle-aware, works well with XML-based UIs.
- **StateFlow** is not lifecycle-aware, works better with Kotlin Coroutines and Jetpack Compose.
- StateFlow is a part of Kotlin Flow and used for modern reactive UIs.

Q:6) What is Repository in MVVM?

The Repository is responsible for fetching data. It abstracts the data sources (API, Room database, Firebase, etc.) from the ViewModel. This separation makes it easy to manage and test data logic.

Q:7) What are UseCases in Clean Architecture?

- A UseCase contains **a single specific business logic** (e.g., GetUserDetails).
- Keeps the ViewModel clean by handling complex logic inside it.
- Lies in the **domain layer** in Clean Architecture.
- Reusable and testable units of code.

Q:8) What is Clean Architecture?

- Divides app into **three layers**:
    - **Presentation**: ViewModel, UI
    - **Domain**: Business logic (UseCases)
    - **Data**: API, Room, Repositories
- Makes code **modular, testable, and maintainable**.
- Helps in scaling large applications.

Q:9) What is Room in Android Architecture?

- Room is a library that provides an easy way to use SQLite.

```
data class User(
  @PrimaryKey val id: Int,
  @Embedded val address: Address
)
```

Q:78) How to update only specific fields in Room?

You can write a custom `@Query` to update only one or two fields:

```
@Query("UPDATE user SET name = :name WHERE id = :id")
suspend fun updateName(id: Int, name: String)
```

Avoid using `@Update` if partial update is needed.

Q:79) Explain SOLID Principles in Android with examples

- SOLID is a set of **five design principles** that help in writing **clean, scalable, and easy-to-maintain code**.
- Each letter in SOLID stands for one principle:
    1. **S** – Single Responsibility
    2. **O** – Open/Closed
    3. **L** – Liskov Substitution
    4. **I** – Interface Segregation
    5. **D** – Dependency Inversion

Let's understand them one by one

S – Single Responsibility Principle (SRP)

- A class should have **only one reason to change**, meaning it should do **only one job**.
- Example in Android:
    - Don't mix UI logic and data logic inside an Activity.
    - Use **Activity** for UI and **ViewModel** for business logic.
    - Use **Repository** for data handling (API/Database).
- This makes code cleaner and easier to test or modify.

O – Open/Closed Principle (OCP)

- A class should be **open for extension** but **closed for modification**.
- You should add new features **without changing existing code**.
- Example in Android:
  - Suppose you have a `PaymentProcessor` class.
  - Instead of editing it for every new payment method (UPI, Card, Wallet), Create new classes like `CardPayment`, `UPIPayment` that implement a `PaymentInterface`.
- This keeps the original class safe from future changes.

L – Liskov Substitution Principle (LSP)

- Subclasses should be usable **in place of their parent class** without breaking the app.
- Example:
  - If you have a `Bird` class with `fly()` method, then any subclass like `Sparrow` or `Eagle` should also support flying.
  - But if the Penguin can't fly, it shouldn't extend `the bird`.
- In Android, this means your subclasses should **behave consistently** with their base classes.

I – Interface Segregation Principle (ISP)

- Don't create **large, all-in-one interfaces**.
- Instead, create **smaller, specific interfaces** that serve one purpose.
- Example in Android:
  - One big `UserActions` interface with methods `login()`, `logout()`, `uploadPhoto()`.
  - Split into smaller interfaces like `AuthActions`, `ProfileActions`, `MediaActions`.
- This keeps code flexible — classes only implement what they need.

D – Dependency Inversion Principle (DIP)

- High-level modules (like `ViewModel`) shouldn't depend on low-level modules (like `RepositoryImpl`).
- Both should depend on **an abstraction (interface)**.
- Example in Android:
    - Create a `UserRepository` interface.
    - Then have multiple implementations like `RemoteUserRepo` and `LocalUserRepo`.
    - Inject it using **Dagger/Hilt** or **manual dependency injection**.
- This makes it easy to **swap implementations** (for example, in testing).

Real Example

Let's say you have a **User Profile Screen**:

- **SRP:** Separate classes for UI (Activity), business logic (ViewModel), and data (Repository).
- **OCP:** Add a new API provider without changing existing data layer code.
- **LSP:** Replace `LocalUserRepository` with `RemoteUserRepository` safely.
- **ISP:** Create small interfaces for login, logout, and profile operations separately.
- **DIP:** ViewModel depends on `UserRepository` interface, not a concrete class.

Q:80) How does Dagger Hilt facilitate the application of the Dependency Inversion Principle in Android?

- Dagger Hilt automatically **injects dependencies** instead of manually creating them.
- It allows your classes (like ViewModels) to depend on **interfaces** instead of concrete classes.
- Example:
    - Define an interface `UserRepository`.
    - Bind `UserRepositoryImpl` using `@Binds` in a module.
    - Hilt provides the implementation automatically wherever needed.

5. Jetpack Compose

Q:1) What is Jetpack Compose?

Jetpack Compose is Android's modern UI toolkit that lets you build UI using Kotlin code instead of XML.

- It's **declarative**, meaning you describe what the UI should look like, and the system updates it automatically when the data changes.
- It replaces traditional XML + View-based UI system.
- Offers **less boilerplate**, better state handling, and **Kotlin-first approach**.

Q:2) What is a Composable function?

A Composable is a special Kotlin function marked with `@Composable` that describes part of the UI.

Example:

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name")
}
```

You can call one composable inside another to build complex UIs.

Q:3) What is recomposition in Jetpack Compose?

Recomposition is when Compose **redraws** parts of the UI because data/state has changed.

- Only the part of the UI where data changed is recomposed.
- Compose optimizes this to avoid redrawing everything.

Example: If you update a `count` value shown in a `Text`, only that `Text` composable will recompose.

Q:4) What is State in Compose?

State holds data that changes over time and triggers recomposition.

You can use `remember` and `mutableStateOf`:

```
val count = remember { mutableStateOf(0) }
```

When `count.value` changes, any UI that depends on it will update automatically.

Q:5) What is remember and rememberSaveable?

- `remember` stores state during recomposition but **resets on configuration changes** (like rotation).
- `rememberSaveable` stores state across recomposition **and configuration changes** using `Bundle`.

Use `rememberSaveable` for things like text input or selection state that should survive screen rotation.

Q:6) What is Modifier in Jetpack Compose?

Modifier is used to **modify or decorate** a composable — like setting padding, background, size, click behavior, etc.

Example:

```
Text(
  text = "Hello",
  modifier = Modifier
    .padding(16.dp)
    .background(Color.Yellow)
)
```

Modifiers are chained and read from **left to right**.

Q:7) What is a Scaffold in Jetpack Compose?

Scaffold is a layout component that provides basic structure like:

- TopBar
- BottomBar
- FloatingActionButton
- Drawer
- SnackbarHost

Example:

```
Scaffold(
    topBar = { TopAppBar(title = { Text("Home") }) },
    floatingActionButton = { FloatingActionButton(onClick = {}) { Text("+") } }
) {
    // Content
}
```

Useful for material design layouts.

Q:8) What is SideEffect in Jetpack Compose?

- In Jetpack Compose, a **SideEffect** is **any operation that affects something outside of the Compose UI tree**.
- Compose functions are **pure by default**, meaning they should not change anything outside themselves.
- **SideEffect** lets you perform actions that **interact with external systems** safely during recomposition.

Why It's Needed

- Compose functions can **recompose multiple times**, so directly performing side-effects (like updating a variable, logging, or showing a toast) can cause **bugs or repeated actions**.
- **SideEffect APIs** provide a **safe way** to run external operations **exactly when Compose recomposes**.

Common Examples of SideEffects

- Updating a **state in ViewModel**
- Showing a **Toast message**
- Logging events
- Triggering **analytics events**

6. Unit Testing

Q:1) What is Unit Testing in Android?

Unit testing is the practice of testing **individual components or functions** in isolation to ensure they behave correctly.

- In Android, we typically use **JUnit** for unit testing.
- Unit tests run on the **JVM** and are **fast** because they don't require a device/emulator.

Q:2) What is the difference between Unit Tests and Instrumentation Tests in Android?

Unit Test	Instrumentation Test
Runs on JVM	Runs on a real device/emulator
Fast	Slower due to UI/device interaction
Tests logic in isolation	Tests integration, UI, and end-to-end
Uses JUnit/Mockito	Uses Espresso, UI Automator, etc.

Q:3) Which tools/libraries are used for Unit Testing in Android?

- **JUnit** – Base library for writing tests.
- **Mockito / MockK** – For mocking dependencies.
- **Truth / AssertJ / Hamcrest** – Assertion libraries.
- **Robolectric** – Allows you to run Android SDK code in JVM unit tests.
- **Turbine** – For testing Kotlin Flow.
- **Kotlin Test DSL** – For idiomatic Kotlin test writing.

Q:4) How do you test ViewModel in Android?

- ViewModels are easy to test because they don't depend on Android Framework.
- You can write plain JUnit tests and verify outputs by observing `LiveData` or `StateFlow`.

7. Android Security

Q:1) How can you securely store sensitive data in an Android app?

You should never store sensitive data (like passwords or tokens) in plain text. Instead:

- Use **EncryptedSharedPreferences** for small data like tokens.
- Use **Android Keystore** to store cryptographic keys securely.
- Avoid storing sensitive info in internal or external storage.

Q:2) What is Android Keystore and why is it used?

Android Keystore is a secure container that helps store cryptographic keys. These keys can be used for encryption, decryption, or signing without exposing them directly to the app.

It ensures that:

- Keys cannot be extracted.
- Operations happen in secure hardware (if available).
- Your app remains safe even if rooted.

Q:3) What are common security risks in Android apps?

Some common risks:

- Storing data in plain text.
- Using HTTP instead of HTTPS.
- Hardcoding API keys in code.
- Not validating inputs (leading to injection attacks).
- Using outdated libraries with vulnerabilities.

Q:4) How can you protect your API keys in Android?

- Don’t hardcode keys in code or strings.xml.
- Use **BuildConfig** with Gradle to store API keys.

- Store keys on the server and use **token-based auth**.
- Use **NDK (native C++)** for critical keys (not fully secure but harder to reverse).

Q:5) How can you prevent reverse engineering of your APK?

- Use **ProGuard** or **R8** to obfuscate the code.
- Remove unused code and classes.
- Avoid storing logic or secrets in the app.
- Sign APKs with release keystore.
- Monitor unauthorized APKs using Play Store Console.

Q:6) What is the use of ProGuard/R8 in Android?

ProGuard (now replaced by R8) is a tool that:

- **Minifies** code (removes unused code).
- **Obfuscates** names (changes class/method names to random characters).
- Makes it harder for attackers to reverse engineer the app.

Q:7) How can you secure communication between app and server?

- Always use **HTTPS (SSL/TLS)** to encrypt data in transit.
- Use **certificate pinning** to verify the server.
- Avoid logging sensitive data (e.g., tokens or passwords).
- Use secure authentication methods like **OAuth2** or JWT.

Q:8) What is certificate pinning?

Certificate pinning is a technique where you **hardcode your server's public certificate or key** in the app. It ensures:

- The app only trusts your server.

8. Scenario Based Questions

Q:1) How do you handle configuration changes (like screen rotation) in Android without losing data?

ViewModel stores UI-related data across configuration changes.

When screen rotates:

- Activity/Fragment is destroyed and recreated.
- ViewModel is not destroyed.
- ViewModel retains the data and passes it again to the UI.

Example:
In a profile screen, if user scrolls halfway and rotates the screen, without ViewModel the screen will reload from start.
But with ViewModel, the profile data and scroll position can be restored smoothly.

Q:2) You have two API calls that must run in parallel and update UI when both complete. How do you implement this?

Use Kotlin Coroutines with `async` and `await`.

```
viewModelScope.launch {
    val userDeferred = async { api.getUser() }
    val postsDeferred = async { api.getPosts() }
    val user = userDeferred.await()
    val posts = postsDeferred.await()
    _uiState.value = Success(user, posts)
}
```

This way, both calls run in parallel and UI updates after both are done.

Q:3) You need to fetch data from both the local Room database and network. How do you design this?

Use **Repository** with a fallback logic:

1. First try Room DB (cached data).

2. If data is old/missing, fetch from API.
3. Save new data in the Room.

This ensures:

- Fast response (local DB)
- Always fresh data (network)

Q:4) A user opens an app with no internet. How do you show offline data?

Use Room as the local cache.

- Repository checks connectivity.
- If offline, fetch from Room.
- If online, fetch from API and update Room.

Show “You’re offline” toast/snackbar while loading cached data.

Q:5) In MVVM, who should handle click events and why?

The **ViewModel** should handle logic, not the Activity/Fragment.

- UI calls `viewModel.onLoginClicked()`
- ViewModel checks input, performs API call
- Emits success/error state via LiveData or StateFlow

Keeps code testable and follows separation of concerns.

Q:6) In Jetpack Compose, how do you preserve scroll position when the user navigates back?

Use `rememberLazyListState()` in Composable:

```
val listState = rememberLazyListState()
LazyColumn(state = listState) { ... }
```

9. DevOps in Android

Q:1) What is CI/CD in Android?

CI/CD in Android development refers to **Continuous Integration and Continuous Delivery/Deployment**, a set of practices that **automate the building, testing, and delivery of Android applications**.

- **Continuous Integration (CI)** means that developers regularly push code to a shared repository (like GitHub), and every push automatically **triggers a build and test**. This helps catch errors early.

- **Continuous Delivery (CD)** means that once code is tested and validated, it can be automatically packaged (APK or AAB) and delivered to testing environments (like Firebase App Distribution).

- **Continuous Deployment** goes one step further and automatically **publishes the app to production** like Google Play once it passes all quality checks.

CI/CD improves team collaboration, reduces manual errors, and speeds up release cycles.

Q:2) Why is CI/CD important in Android development?

CI/CD helps in:

- **Faster development cycles** by automating build and testing.

- **Early bug detection** due to frequent code integration and automated tests.

- **Better team collaboration**, as code is constantly merged and verified.

- **Reduced manual work** — no need to manually run tests, generate APKs, or upload to Play Store.

- **Consistent builds** because the process is scripted and version-controlled.

Q:3) Which tools are commonly used for CI/CD in Android?

Some commonly used CI/CD tools are:

- **GitHub Actions** – Integrated with GitHub, good for open-source and personal projects.

- **Bitrise** – Android and iOS friendly, no setup needed, GUI-based.

10. Gradle Concepts & Issues

Q:1) What is Gradle in Android?

- Gradle is the **build system** used in Android.
- It automates compiling code, packaging APKs, and managing dependencies.
- Think of it as a **recipe** that tells Android Studio how to build your app.
- It's fast, flexible, and supports custom build configurations.

Q:2) What is the difference between Project-level and Module-level `build.gradle`?

Project-level `build.gradle`

- Applies to the entire project.
- Defines global configurations such as:
  - Gradle version
  - Repositories
  - Classpath for plugins

Module-level `build.gradle`

- Specific to each app/module.
- Defines module-specific settings such as:
  - Dependencies (`implementation`, `api`, etc.)
  - Build types (debug/release)
  - Product flavors
  - Android SDK version

Key Point: Project-level is for general setup affecting all modules, while Module-level is for app/module-specific configurations.

Q:3) What are Build Variants and Product Flavors?

- **Build Variants:** Combination of build type (debug/release) + flavor. Example: `freeDebug`, `paidRelease`.

Testimonials

"The Android Developer Interview Handbook was an invaluable resource during my interview preparation. Clear, concise, and comprehensive—highly recommended!"

— Eva Das

"This handbook is just wow! It reveals hidden details behind concepts that many overlook. A must-read for every native Android developer."

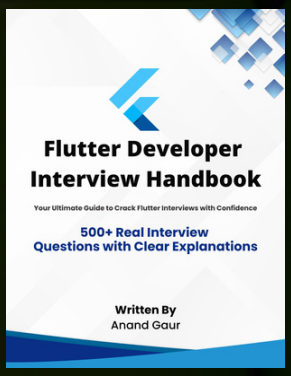
— Dharmender Kumar

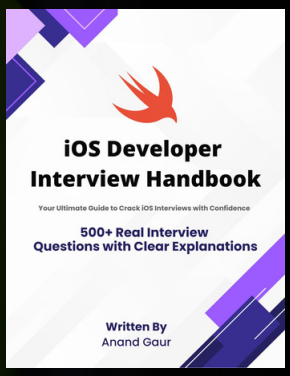
"Extremely helpful! Whether you're a beginner or experienced Android developer, this book offers great value. Perfect for job switchers or anyone wanting to strengthen their Android and architectural concepts."

— Sushmita Biswas

"Your Complete Guide to Cracking Android Interviews"

If you enjoyed this, check out my other books





About Author

Anand Gaur is a Tech Lead with rich experience in mobile app development, specializing in Android, Kotlin, and Kotlin Multiplatform. He has mentored many developers and guided them to crack interviews at top IT companies.

You can find Anand at https://linktr.ee/anandgaur