# iOS Developer Interview Handbook

Your Ultimate Guide to Crack iOS Interviews with Confidence

## 500+ Real Interview Questions with Clear Explanations

**Written By**

Anand Gaur

## Table of Contents

# 1. Introduction

## Why This Handbook?

Preparing for iOS interviews can feel overwhelming because the questions vary widely depending on the company and the role. One round might test your Swift language fundamentals, another might focus on UIKit or SwiftUI implementation, and at senior levels you'll often be asked to discuss architecture, scalability, and system design.

That's exactly why this handbook was created — to be your **one-stop guide**. It brings together all the key concepts, common questions, and scenario-based discussions in a structured way, so you don't waste hours searching through scattered resources.

As a developer, I know this struggle firsthand. I always found it frustrating to prepare for interviews by juggling multiple sources for questions and answers. There was no single place that covered real, relevant, and repeatedly asked questions with clear explanations. That's when I decided to create this handbook — a structured and reliable guide that helps every iOS developer prepare smarter and faster.

This handbook has been created by collecting real interview questions from the past 7–8 years, gathered from multiple developers who have gone through real interview processes across different companies. Over time, I documented every challenging question I encountered, not only to improve myself but also to share with others so they can benefit too.

Here's what you'll find inside:

- **Topic-wise categorized questions** so you can prepare step by step (Swift, UIKit, SwiftUI, Architecture, System Design, etc.).
- **Coverage for all experience levels** — from freshers who are just starting out to senior developers and architects aiming for leadership roles.
- **Practical examples and scenario-based discussions** that reflect real-world interview patterns.

No matter how much experience you have, this handbook will give you a clear roadmap of what to expect and how to prepare, helping you approach your interviews with confidence.

## How to Use This Handbook Effectively

Think of this handbook as a **step-by-step preparation path**:

1. **Begin with the fundamentals** – If you're a fresher, start with Swift basics and iOS application lifecycle. Build a strong foundation before moving on.

2. **Progress gradually** – Move into UI frameworks (UIKit/SwiftUI), then cover data persistence, networking, and concurrency.

3. **Focus on real-world problems** – Don't just read theory. Practice scenario-based questions like "How would you design offline support in an app?" since interviews increasingly emphasize practical problem solving.

4. **Advance to higher-level concepts** – For mid-level to senior developers, dive into design patterns, architecture (MVC, MVVM, VIPER, Clean), and dependency management.

5. **Don't ignore DevOps and Security** – Continuous integration, app distribution, SSL pinning, and data protection are becoming standard expectations, even in interviews.

6. **Revise smartly** – Use the scenario and system design sections as your final revision before the interview.

By following this order, you'll avoid the common trap of studying random topics without direction.

## Levels of iOS Interviews

- **Fresher / Junior (0–2 years)**
  Expect questions on Swift syntax, optionals, closures, basic UIKit/SwiftUI components, simple REST API calls, and basic table/collection views.

- **Mid-level (2–5 years)**
  Focus shifts to architectural understanding, MVVM or MVC, Core Data or Realm, multithreading with GCD/OperationQueue, error handling, and performance optimization.

- **Senior (5+ years)**
  You'll be evaluated on how you design applications end-to-end, apply design patterns, use dependency injection, optimize for performance, and mentor teams. Expect tough discussions around trade-offs in architecture and scalability.

- **Architect / Lead**
  Interviews go beyond code. You'll be asked to design large-scale systems, integrate DevOps pipelines, handle security at scale, and make decisions for modularization and team productivity. Soft skills—like leadership, communication, and decision-making—become as important as technical depth.

## Common Interview Patterns in Product & Service Companies

- **Product Companies (FAANG, unicorn startups, product-focused firms):**
  - Algorithm and data structures round (usually easy to medium level)
  - Deep technical rounds on Swift, memory management, UIKit/SwiftUI
  - System design and scalability discussions
  - Culture fit or behavioral interviews
- **Service Companies (Infosys, TCS, Accenture, Wipro, etc.):**
  - More emphasis on practical, implementation-based questions
  - Testing knowledge of frameworks like UIKit, SwiftUI, Core Data
  - Scenarios around API integration, error handling, offline caching
  - Usually fewer rounds and faster decision-making process

# 2. iOS Basics

## Q1: Can you explain the history of iOS and how it evolved over time?

iOS is Apple's mobile operating system that powers the iPhone, iPad, and iPod Touch. Its journey started back in **2007** when Apple released the first iPhone. At that time, it was called **iPhone OS**, and it mainly supported basic apps like Phone, Mail, Safari, and iPod.

Over the years, Apple has released major updates that completely transformed iOS:

- **iPhone OS 1 (2007):** The very first version with basic apps. No App Store yet.
- **iPhone OS 2 (2008):** Launch of the **App Store**, which revolutionized mobile apps.
- **iPhone OS 3 (2009):** Added **copy-paste, MMS, push notifications**.
- **iOS 4 (2010):** Renamed from iPhone OS to **iOS**. Introduced **multitasking** and FaceTime.
- **iOS 5 (2011):** Brought **iMessage, Notification Center, and iCloud**.
- **iOS 6 (2012):** Introduced **Apple Maps** and **Passbook** (later Wallet).
- **iOS 7 (2013):** Major **UI redesign** with flat design, added **Control Center** and AirDrop.
- **iOS 8 (2014):** Allowed **third-party keyboards**, introduced **HealthKit** and HomeKit.
- **iOS 9 (2015):** Performance and battery optimizations, **Proactive Assistant**.
- **iOS 10 (2016):** Revamped iMessage, redesigned notifications, better Siri integration.
- **iOS 11 (2017):** Big iPad improvements—**multitasking, drag-and-drop, Files app**.
- **iOS 12 (2018):** Added **Screen Time** and grouped notifications.
- **iOS 13 (2019): Dark Mode, Sign in with Apple,** stronger privacy.
- **iOS 14 (2020): Home screen widgets, App Library, picture-in-picture.**
- **iOS 15 (2021): SharePlay, Focus Mode,** redesigned notifications.

- **iOS 16 (2022): Lock Screen customization, iMessage edit & undo send.**
- **iOS 17 (2023): Contact Posters, NameDrop, interactive widgets.**
- **iOS 18 (2024):** Focused on **AI features (Apple Intelligence), more customization, tighter macOS + visionOS integration.**
- **iOS 26 (2025, upcoming):**
    - Apple skipped iOS 19–25 and renamed it **iOS 26** to match the **2025–2026 cycle**.
    - Announced at **WWDC 2025**, releasing September 2025 with **iPhone 17**.
    - Key features:
        - New **"Liquid Glass" UI** design with smooth, translucent effects.
        - AI-powered upgrades like **Live Translations, smarter Siri, on-device intelligence.**
        - **Call Screening** to block scam/spam calls.
        - **Wireless charging improvements** and Live Activities for CarPlay.

## Q2: What is the difference between iOS and other mobile platforms?

The two biggest mobile platforms today are **iOS (Apple)** and **Android (Google)**. Both have similarities, but there are some very important differences that interviewers usually expect you to highlight:

### 1. Ecosystem & Hardware Control

- **iOS:** Apple controls both **hardware and software**. iPhones, iPads, and iOS are designed together, which gives very smooth performance, optimization, and long-term updates.
- **Android:** Google develops the OS, but many manufacturers (Samsung, OnePlus, Xiaomi, etc.) make hardware. This leads to **fragmentation** and inconsistent updates across devices.

### 2. App Store vs Play Store

- **iOS:** Apps are distributed only through the **Apple App Store** (unless jailbroken). Apple has strict **review guidelines**, which ensures quality and security but also means higher approval time for developers.

- **Android:** Apps are available on **Google Play Store** and even third-party stores (like Samsung Galaxy Store). Developers have more flexibility, but this also increases the risk of **malware apps**.

### 3. Programming Languages & Frameworks

- **iOS:** Apps are mainly built using **Swift** and **Objective-C**, with UI frameworks like **UIKit** and **SwiftUI**.

- **Android:** Apps are built using **Kotlin** and **Java**, with UI done in XML or Jetpack Compose.

### 4. Design Philosophy & UI Consistency

- **iOS:** Apple enforces very strict **Human Interface Guidelines (HIG)**. This means apps on iOS generally look and feel consistent.

- **Android:** Material Design is the guideline, but manufacturers often customize UI heavily (e.g., Samsung One UI, Xiaomi MIUI). So, Android apps sometimes feel less consistent across devices.

### 5. Updates & Support

- **iOS:** Apple devices receive **5–6 years of software updates** regularly. Even older iPhones keep getting the latest iOS version.

- **Android:** Updates depend on the manufacturer. Google Pixel gets fast updates, but most other phones stop receiving updates after **2–3 years**.

### 6. Security & Privacy

- **iOS:** Strong focus on **privacy and security**. Features like App Tracking Transparency, on-device AI, strict sandboxing, and controlled app store review process.

- **Android:** Google has improved security (Play Protect, sandboxing), but because of openness and third-party app stores, malware risk is higher.

### 7. Customization

- **iOS:** More **restricted customization**. Users can change wallpapers, add widgets, and now customize lock screen, but system-wide modifications are limited.

- **Android:** Very **flexible**. Users can change launchers, themes, icons, and even root the device to deeply customize it.

# 3. Swift Programming Language

## Q1: What is the difference between `var` and `let` in Swift?

In Swift, we use `var` and `let` to **store values in memory**. But the difference comes in how we can **change** those values later.

**`var` = Variable (Changeable value)**

- If you use `var`, the value can be changed later.

**Example:**

```
var name = "Anand"
name = "Rahul"    // Allowed
```

Here, we first stored `"Anand"`, then changed it to `"Rahul"`.

**`let` = Constant (Fixed value)**

- If you use `let`, the value cannot be changed once it's assigned.

**Example:**

```
let pi = 3.14159
pi = 3.14    //  Error: Cannot change a constant
```

This makes your code **safe** because you know the value won't change accidentally.

**When to use `var`:**

- Use it when the value is expected to change in the future.
- Example:
  - User's current location
  - A counter in a loop
  - A score in a game

**When to use `let`:**

- Use it when the value should stay the same.
- Example:
  - Birthdate of a person
  - API keys

# 4. Object-Oriented & Protocol-Oriented Programming

## Q1: Explain Protocol-Oriented Programming (POP). How is it different from OOP?

### Protocol-Oriented Programming (POP)?

- POP is a programming paradigm introduced in Swift where **protocols** are the primary tool for defining **interfaces and behavior**.
- Focuses on **what types can do**, not **what they are**.
- Encourages **composition over inheritance**, making code more flexible and reusable.

```swift
protocol Drivable {
    func drive()
}

extension Drivable {
    func drive() {
        print("Driving...")
    }
}

struct Car: Drivable {}
struct Bike: Drivable {}

let car = Car()
car.drive()   // Prints "Driving..."
```

Here, `Car` and `Bike` **both conform to `Drivable`** without sharing a common superclass.

Default behavior is provided via protocol extension.

### How is POP Different from OOP?

| Feature | OOP (Object-Oriented) | POP (Protocol-Oriented) |
|---|---|---|
| **Main Focus** | Objects and their hierarchy | Protocols and behavior |

| | | |
|---|---|---|
| **Code Reuse** | Inheritance (classes) | Protocol extensions & composition |
| **Flexibility** | Less flexible with deep inheritance | Highly flexible, can mix behaviors easily |
| **Type System** | Class-based reference types | Works with structs, enums, and classes |
| **Default Implementation** | Must override or subclass | Provided via protocol extensions |

### Advantages of POP

1. Promotes **composition over inheritance** → less tight coupling.
2. Works with **value types** (structs & enums), enabling safer and faster code.
3. Allows **default implementations** in protocol extensions.
4. Improves **testability** by decoupling behavior from concrete types.
5. Reduces problems like the **diamond inheritance problem** in OOP.

### When to Use POP

- When you want **shared behavior without creating a deep class hierarchy**.
- For **value types** (structs/enums).
- When you want to **compose multiple behaviors** in a type safely.

## Q2: Why Swift is called a Protocol-Oriented Programming (POP) language

### 1. Protocols Are First-Class Citizens

- In Swift, **protocols can define behavior, properties, and requirements**.
- Types (structs, classes, enums) can **conform to multiple protocols**, enabling **flexible composition**.
- This makes **protocols the central way to define interfaces and shared behavior**.

## 2. Default Implementations via Protocol Extensions

- Swift allows **protocol extensions** to provide default implementations.
- This means types get **shared behavior automatically** without needing inheritance.

```
protocol Greetable {
    func greet()
}

extension Greetable {
    func greet() {
        print("Hello!")
    }
}

struct Person: Greetable {}
Person().greet()  // Prints "Hello!"
```

Even `structs` and `enums` can adopt these behaviors — something **class-based OOP cannot do as flexibly**.

## 3. Encourages Composition Over Inheritance

- Traditional OOP relies heavily on **class hierarchies**, which can get rigid and complex.
- Swift's POP encourages **combining multiple protocols** to compose functionality without deep inheritance trees.

```
protocol Drivable { func drive() }
protocol Flyable { func fly() }

struct FlyingCar: Drivable, Flyable {
    func drive() { print("Driving") }
    func fly() { print("Flying") }
}
```

## 4. Works Seamlessly with Value Types

- Swift is optimized for **structs and enums**, not just classes.
- POP allows **structs and enums** to adopt protocols and get shared behavior, making code **safer and more efficient**.

# 5. iOS UI Development (UIKit)

## Q1: What are the advantages and disadvantages of using Storyboards?

### Advantages of Storyboards

#### 1. Visual Representation

- You can see the **entire app flow visually**, including screens (ViewControllers) and segues.
- Makes understanding navigation and UI relationships easier.

#### 2. Quick Prototyping

- Drag-and-drop UI elements and connect actions/outlets quickly.
- Great for **rapid prototyping** or small projects.

#### 3. Built-in Auto Layout Support

- You can configure constraints visually.
- Interface Builder gives immediate feedback on layout issues.

#### 4. Easy Segues & Navigation

- Connect screens using segues without writing code.
- Handles navigation transitions automatically.

#### 5. Less Boilerplate Code

- No need to manually create views in code for simple UIs.

### Disadvantages of Storyboards

#### 1. Merge Conflicts

- Storyboard files are XML-based.
- Multiple developers editing the same storyboard can easily cause **git merge conflicts**.

**2. Scalability Issues**

- Large storyboards with many screens can be **slow to open** and difficult to manage.
- Hard to find specific views or segues.

**3. Lack of Reusability**

- Reusing a screen or component across multiple storyboards requires **copying and pasting**, not ideal for modular apps.

**4. Tightly Coupled Code**

- Storyboard-based UIs can make it **harder to separate UI and business logic**.
- Testing and programmatic modifications become more complex.

**5. Performance Overhead**

- Loading a very large storyboard at runtime may take slightly longer than loading programmatic views.

## Q2: What's the difference between Storyboards, XIB files, and Programmatic UI?

### 1. Storyboards

- **Definition:** A single visual file that can contain multiple screens (ViewControllers) and segues.
- **Usage:** Shows the entire app flow visually.
- **Pros:**
    - Visual representation of navigation and UI.
    - Quick to prototype and connect screens.
    - Auto Layout support with Interface Builder.
- **Cons:**
    - Hard to manage for large apps (merge conflicts, slow to load).
    - Less reusable for components.

# 6. SwiftUI

## Q1: What is SwiftUI and how does it differ from UIKit?

- **SwiftUI** is Apple's **declarative UI framework** introduced in 2019.
- It allows you to **build UIs using Swift code in a declarative way**: you describe **what the UI should look like** rather than how to construct it step by step.
- Works across **iOS, iPadOS, macOS, watchOS, and tvOS**.
- Integrates with **data binding**, animations, and accessibility automatically.

### Key Features of SwiftUI

#### 1. Declarative Syntax:

- You declare the UI using structs and modifiers.
- Example:

```
Text("Hello, SwiftUI!")
    .font(.title)
    .foregroundColor(.blue)
```

#### 2. State-driven:

- UI automatically updates when your data changes using `@State`, `@Binding`, or `@ObservedObject`.

#### 3. Cross-platform:

- Same SwiftUI code can run on iPhone, iPad, Mac, Apple Watch, and Apple TV with minimal changes.

#### 4. Live Preview:

- Xcode provides **instant previews** of your UI while coding.

**Differences Between SwiftUI and UIKit**

| Feature | UIKit | SwiftUI |
|---|---|---|
| **Programming Style** | Imperative (you tell the system *how* to do things) | Declarative (you tell the system *what* you want) |
| **UI Updates** | Manual updates required (set properties, call `setNeedsLayout`) | Automatic updates using state bindings |
| **Cross-platform** | iOS only | iOS, iPadOS, macOS, watchOS, tvOS |
| **View Hierarchy** | Views are classes (`UIView`), need manual management | Views are structs (`View`), lightweight and value types |
| **Animations** | Manual (`UIView.animate`) | Built-in with simple modifiers (`withAnimation`) |
| **Learning Curve** | Mature, extensive documentation, more boilerplate | Newer, simpler syntax, but limited legacy support |

## Q2: Explain the SwiftUI view lifecycle.

Unlike UIKit, **SwiftUI doesn't have explicit lifecycle methods** like `viewDidLoad` or `viewWillAppear`. Instead, it is **state-driven and declarative**, and the system decides when to create, update, or destroy views.

### 1. View Creation

- SwiftUI views are **structs**, not classes.
- Whenever the state changes, SwiftUI **recreates the view struct**.
- The system **diffs the new view with the previous one** to update only what changed (efficient rendering).

### 2. Body Evaluation

- The `body` property is **re-evaluated whenever state changes**.

**Example:**

```
struct ContentView: View {
    @State private var counter = 0

    var body: some View {
        VStack {
            Text("Count: \(counter)")
            Button("Increment") {
                counter += 1
            }
        }
    }
}
```

- Pressing the button **updates `counter`**, which triggers **body re-evaluation**.

### 3. View Updates

- SwiftUI **only updates what has changed** in the UI.
- This is called **diffing**—the framework compares old and new views and applies minimal changes.

### 4. onAppear & onDisappear

- For actions that need to run when a view appears/disappears, use:

```
Text("Hello")
    .onAppear { print("View appeared") }
    .onDisappear { print("View disappeared") }
```

### 5. State-driven Lifecycle

- The view lifecycle is **completely driven by state changes**.
- Views **don't persist in memory** unless referenced; they're recreated as needed.

### 6. Environmental & Observable Objects

- Views can subscribe to `@ObservedObject`, `@StateObject`, `@EnvironmentObject`.
- When data in these objects changes, **SwiftUI triggers a view refresh** automatically.

## Q3: What are ViewBuilder and @ViewBuilder used for?

### `ViewBuilder`:

- `ViewBuilder` is a **special type of function builder** in SwiftUI.
- It allows you to **combine multiple views** inside a closure and return them as a single view.
- Without `ViewBuilder`, you'd have to wrap everything manually, which would be messy.

Example:

```
VStack {
    Text("Hello")
    Text("World")
}
```

Here, the closure after `VStack { ... }` uses `@ViewBuilder` under the hood, so you can write multiple views without wrapping them in an array.

### `@ViewBuilder:`

- `@ViewBuilder` is an **attribute** you put on a function parameter or property.
- It tells SwiftUI: "This closure will return multiple views, and I'll combine them into one using ViewBuilder rules."

**Example:**

```
struct MyCustomView<Content: View>: View {
    let content: () -> Content   // normal closure

    var body: some View {
        VStack {
            content()
        }
    }
}
```

## Q4: What does the `@main` attribute do in a SwiftUI app?

The `@main` attribute in SwiftUI marks the **entry point of your app**.

# 8. Networking in iOS

## Q1: What is URLSession? How is it different from the deprecated NSURLConnection?

- `URLSession` is Apple's **high-level networking API** used to make HTTP requests (GET, POST, etc.), download/upload files, and manage background transfers.
- It's built on top of lower-level sockets but provides an **easy, modern, async-friendly API**.
- It supports:
    - Data tasks → simple requests (e.g., fetch JSON).
    - Download tasks → large files (auto-saves to disk).
    - Upload tasks → send files.
    - Background tasks → run even if the app is suspended.

### Difference between URLSession and NSURLConnection

Apple deprecated `NSURLConnection` in favor of `URLSession`. Here's why:

| **Feature** | `NSURLConnection` | `URLSession` |
|---|---|---|
| **API Style** | Older, delegate-heavy | Modern, supports completion handlers + async/await |
| **Background Transfers** | ❌ Not supported | ✅ Built-in with `URLSessionConfiguration.background` |
| **Task Types** | Limited (basic requests only) | Multiple: `dataTask`, `downloadTask`, `uploadTask`, `streamTask` |
| **Efficiency** | Less efficient, no fine-grained config | More efficient, configurable caching, cookies, timeouts |
| **Reusability** | One connection per request | Sessions can be reused across multiple tasks |
| **Current Status** | Deprecated since iOS 9 | Standard networking API in iOS, macOS, watchOS, tvOS |

# 11. Dependency Management

## Q1: What is Swift Package Manager? What advantages does it have over other dependency managers?

- It's Apple's **official tool** for managing dependencies and sharing code.
- Built directly into **Xcode** and the Swift toolchain.
- Lets you add, update, and use libraries without needing extra tools like CocoaPods or Carthage.

You describe your package in a `Package.swift` file, and SPM handles downloading, compiling, and linking everything for you.

### Advantages of SPM over other dependency managers

#### 1. Official and Native

- Built by Apple, fully supported in Xcode.
- No need to install extra tools (unlike CocoaPods or Carthage).

#### 2. Lightweight & Fast

- Direct integration with the Swift compiler.
- Faster dependency resolution and build times compared to CocoaPods.

#### 3. Cross-Platform

- Works not only for iOS/macOS but also for **Linux and server-side Swift** projects.
- CocoaPods and Carthage are mostly iOS/macOS focused.

#### 4. Simple Setup

- Just add dependencies via Xcode's "Swift Packages" menu or edit `Package.swift`.
- No messing around with Xcode project files (CocoaPods modifies them).

**5. No Extra Files**

- Doesn't generate bulky `Pods` directories or `.xcworkspace`.
- Keeps your repo cleaner.

**6. Better Git Integration**

- Dependencies are fetched directly from Git repositories.
- Supports versioning using **semantic versioning** (`1.2.3`).

**7. Future Proof**

- Apple continues to improve SPM (e.g., binary dependencies, resources support).
- More and more third-party libraries are shifting to SPM-first.

## Q2: What's the difference between `.library()` and `.executable()` products?

When you define a Swift package in **Package.swift**, you specify **products**.
Products are what your package actually exposes to the outside world (like apps or other packages).

There are mainly two kinds: **library** and **executable**.

### `.library()` Product

- **What it is:**
  A collection of Swift code that can be reused by other packages or apps.
- **Purpose:**
  To provide **reusable functionality** (e.g., helper functions, UI components, networking layer).
- **Output:**
  Doesn't run by itself, it just gives you Swift modules/frameworks to import.

### `.executable()` Product

- **What it is:**
  A Swift program that can be run directly (like a command-line tool or even the main

# 17. DevOps for iOS Developers

## Q1: What is Mobile DevOps? How is it different from traditional web DevOps?

- **Mobile DevOps** is the practice of applying **DevOps principles** specifically to **mobile app development**.
- It focuses on **continuous integration, delivery, and monitoring** for mobile apps across iOS and Android platforms.
- The goal is to **release high-quality apps faster**, with automated testing, deployment, and feedback loops.

### Key Components of Mobile DevOps

| Component | Description |
|---|---|
| **Continuous Integration (CI)** | Automatically build and test apps whenever code changes are committed. |
| **Continuous Delivery / Deployment (CD)** | Automate deployment of apps to testers (TestFlight, Play Store beta) or production. |
| **Automated Testing** | Unit, UI, and integration tests run automatically to catch bugs early. |
| **Monitoring & Analytics** | Track app crashes, performance, and user behavior in real-time. |
| **Release Management** | Plan releases, manage versions, and handle app store submissions efficiently. |

### How Mobile DevOps Differs from Traditional Web DevOps

| Aspect | Mobile DevOps | Web DevOps |
|---|---|---|
| **Deployment Target** | App Store (iOS) / Play Store (Android) | Web servers / cloud platforms |
| **Release Cycle** | Longer due to app store review and approvals | Faster, can push live immediately |

| | | |
|---|---|---|
| **Testing Environment** | Multiple devices, OS versions, screen sizes | Browsers & server environments |
| **Installation** | User must download/update app | Users access via browser, no install needed |
| **Monitoring** | Crash reporting, app performance metrics | Server logs, uptime, response times |

### Key Challenges in Mobile DevOps

**1. Multiple Platforms**

- iOS and Android need separate pipelines and tools.

**2. App Store Constraints**

- Deployment is slower due to **review processes**.

**3. Device Fragmentation**

- Need to test on **different devices, OS versions, screen sizes**.

**4. Version Management**

- Manage multiple **app versions simultaneously** for testing and production.

## Q2: What is the difference between DevOps, CI, and CD in the context of mobile development?

### 1. DevOps in Mobile Development

- **DevOps** is a **culture and set of practices** that combines **development and operations** to deliver apps faster and more reliably.
- In mobile, DevOps covers **everything from code commits to deployment and monitoring** on iOS and Android.
- Goals: **automation, collaboration, feedback loops, and faster releases**.

# 18. Scenario-Based Interview Questions

## Q1: Users report that your app crashes randomly on older devices while scrolling a `UITableView`. How would you investigate and resolve this crash?

**Reproduce the crash**

- Test on older devices or low-memory simulators to see the problem firsthand.

**Check crash logs**

- Use Xcode Organizer, Crashlytics, or Sentry to get stack traces and error details.

**Identify memory issues**

- Use **Instruments** → **Allocations & Leaks** to detect high memory usage, leaks, or retain cycles.

**Optimize table view performance**

- Reuse cells properly with `dequeueReusableCell`.
- Avoid heavy computations or image processing in `cellForRowAt`.
- Load images asynchronously or lazily.

**Fix retain cycles**

- Check closures capturing `self` strongly and use `[weak self]` or `[unowned self]` where needed.

**Test thoroughly**

- Verify on multiple devices, iOS versions, and simulate memory warnings to ensure the fix works.

## Q2: Your iOS app takes ~8 seconds to launch, affecting user experience. How would you optimize app launch time?

**Measure launch time**

- Use **Instruments** → **Time Profiler** or `os_signpost` to identify slow parts.

**Move heavy work off main thread**

- Run network calls, database queries, or JSON parsing asynchronously.

**Lazy load resources**

- Load images, views, or data only when needed instead of during launch.

**Optimize storyboard/XIB**

- Reduce deep view hierarchies and unnecessary UI elements.
- Consider lightweight views or SwiftUI for faster layout rendering.

**Cache frequently used data**

- Use `UserDefaults`, `Core Data`, or local files to avoid fetching from network repeatedly.

**Preload only essential resources**

- Delay non-critical initializations to after app launch.

## Q3: A view controller is not deallocated after navigation, causing memory leaks. How would you handle this?

**Detect retain cycles**

- Use **Xcode Memory Graph Debugger** or **Instruments** → **Leaks**.

**Identify strong reference cycles**

- Look for closures or delegates holding a strong reference to `self`.