iOS Developer Interview Questions (2025 Edition)

General iOS & Swift Questions

1. Can you walk us through your experience with Swift and iOS development?

Answer:
I have been working with Swift and iOS development for **[X] years**. During this time, I have developed and maintained multiple iOS applications and SDKs, focusing on delivering high-performance and scalable solutions.

Key areas of expertise:

- **Swift Development:** Proficient in Swift, leveraging modern language features like generics, protocols, extensions, and property wrappers to write clean and reusable code.
- **UIKit & SwiftUI:** Experience with UIKit for building traditional iOS apps, along with SwiftUI for declarative UI development.
- **Architectures & Design Patterns:** Hands-on experience with MVC, MVVM, and VIPER for structuring code and improving maintainability.
- **Networking & APIs:** Worked extensively with `URLSession` and `Alamofire` for network calls, implementing best practices for handling API requests and responses.
- **Memory Management & Performance Optimization:** Proficient in identifying and fixing memory leaks, optimizing performance through Instruments and profiling tools.
- **Testing & CI/CD:** Experience with writing unit tests using `XCTest`, implementing test-driven development (TDD), and integrating CI/CD pipelines for automated builds and releases.

I have also worked on developing iOS SDKs, ensuring smooth integration for third-party developers, maintaining backward compatibility, and optimizing for performance.

---

2. What are the key differences between Swift and Objective-C?

Answer:
Swift and Objective-C are both used for iOS development, but Swift is a modern, more efficient language with safety and performance improvements. Below are the key differences:

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Feature	Swift	Objective-C
Syntax	Clean, concise, and expressive	Verbose and complex
Type Safety	Strongly typed, preventing many errors	Weakly typed, leading to potential runtime issues
Memory Management	Automatic Reference Counting (ARC) for both value and reference types	ARC only for Objective-C objects, requiring manual memory handling for C-based code
Performance	Faster due to static typing and optimizations	Slower due to dynamic typing and method dispatch
Optionals	Introduces optionals (`?` and `!`) to handle `nil`safely	Uses `nil`, which can cause unexpected crashes
Interoperability	Works seamlessly with Objective-C and C code	Cannot directly use Swift code
String Handling	Uses `String` type with built-in Unicode support	Uses `NSString`, requiring manual conversion
Error Handling	Uses `do-catch`, `throws`, and `try` for structured error handling	Uses `NSError`, requiring manual error handling

Why Swift is Preferred?

- It is **faster**, **safer**, and **easier to read and write**.
- Reduces the chances of **null pointer exceptions** with optionals.
- Modern **error handling** and **memory management** features enhance stability.

However, Objective-C is still used in legacy applications, and knowledge of it is useful when working with older codebases or integrating with C libraries.

---

3. How do you handle memory management in Swift?

Answer:
Memory management in Swift is handled using **Automatic Reference Counting (ARC)**, which tracks and manages memory allocation and deallocation for class instances.

Key Aspects of Memory Management in Swift:

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

1. **Reference Counting:**
    - Each object in Swift has a reference count. When it reaches zero, the memory is deallocated.

Example:

```
class Person {
    var name: String
    init(name: String) {
        self.name = name
    }
}
var person1: Person? = Person(name: "John")
person1 = nil // Reference count drops to 0, deallocating memory.
```

2. **Strong, Weak, and Unowned References:**
    - **Strong references**: Keep an object in memory. Can cause **retain cycles** if two objects hold strong references to each other.
    - **Weak references**: Used to avoid retain cycles. Can become `nil`, so always declared as **optional** (`weak var`).
    - **Unowned references**: Similar to `weak`, but never `nil`. Used when one object will never outlive the other.

Example of a Retain Cycle and Fixing it:

```
class Parent {
    var child: Child?
}
class Child {
    weak var parent: Parent? // Using weak to break the retain cycle
}
```

3. **Using Capture Lists in Closures:**
    - Closures **retain references** by default, which can cause memory leaks.
    - Use `[weak self]` or `[unowned self]` in capture lists.

Example:

```
class ViewController {
    var completion: (() -> Void)?

    func setupClosure() {
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
        completion = { [weak self] in
            print("Closure executed")
        }
    }
}
```

4. **Profiling Memory Usage:**
    - Use **Instruments (Leaks, Allocations, Zombies)** in Xcode to detect memory leaks.
    - Check **retain cycles** and **memory spikes** in long-running applications.
    - **Best Practices for Memory Management:**
      ✅ Use `weak` or `unowned` where necessary to prevent retain cycles.
      ✅ Avoid strong references to `self` inside closures.
      ✅ Release unused objects by setting them to `nil`.
      ✅ Use `lazy var` for objects that are not needed immediately.

---

4. What are value types and reference types in Swift? When would you use each?

Answer:
Swift has **value types** and **reference types**, which determine how data is stored and passed in memory.

Type	Definition	Example
Value Type	A copy of the value is created when assigned or passed to a function. Changes do not affect the original instance.	`struct`, `enum`, `tuple`
Reference Type	A reference (pointer) is passed, meaning multiple variables can refer to the same instance. Changes affect all references.	`class`, `closure`

Example:

```
struct ValueType {
    var number: Int
}
class ReferenceType {
    var number: Int
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
    init(number: Int) { self.number = number }
}

var a = ValueType(number: 10)
var b = a  // A copy is created
b.number = 20
print(a.number) // 10 (original remains unchanged)

var x = ReferenceType(number: 10)
var y = x  // Both x and y point to the same instance
y.number = 20
print(x.number) // 20 (both reflect changes)
```

When to Use Each?

> ✅ **Use value types (`struct`)** for data that should remain immutable and independent (e.g., `CGPoint`, `CGRect`).
> ✅ **Use reference types (`class`)** when objects need to share state (e.g., ViewControllers, network sessions).

---

5. What are optionals in Swift? How do you handle them safely?

Answer:
Optionals are a Swift feature that allows variables to either hold a value or be `nil`, preventing crashes due to uninitialized variables.

Declaring Optionals:

```
var name: String? = "John" // Can be nil
var age: Int?              // Defaults to nil
```

Safely Handling Optionals:
Forced Unwrapping (Not Recommended)

```
print(name!) // Unsafe if name is nil
```

1. **Optional Binding (Recommended)**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
    if let unwrappedName = name {

    print(unwrappedName) // Safely unwraps
}
```

2. **Nil-Coalescing Operator (??)**

```
    let username = name ?? "Guest" // Uses default if nil
```

3. **Guard Statement (Early Exit)**

```
    func printName() {

    guard let safeName = name else { return }
    print(safeName)
}
```

✅ **Best Practice:** Always use optional binding or `guard` to safely unwrap values. Avoid force unwrapping to prevent crashes.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

SDK Development & Architecture

1. What challenges have you faced while developing an iOS SDK?

Developing an iOS SDK comes with several challenges, ranging from API design and backward compatibility to performance optimization and security concerns. Some of the key challenges I have faced include:

1. API Design & Usability

- Designing an **intuitive and developer-friendly API** is critical.
- Ensuring **clean, well-documented**, and **easy-to-use** APIs to minimize integration effort.
- Providing clear error messages and debugging logs.

✅ **Solution:** Follow Apple's API design guidelines, provide sample implementations, and write thorough documentation.

2. Backward Compatibility

- Apps integrating the SDK might be using different iOS versions.
- Breaking changes in newer SDK versions can cause integration issues.

✅ **Solution:**

- Use **feature flags** to maintain compatibility with older versions.
- Deprecate APIs gradually instead of removing them outright.
- Provide **versioning support** and changelogs for developers.

3. Performance Optimization

- Large SDKs can slow down the host app.
- Network-heavy SDKs can degrade app performance.

✅ **Solution:**

- Optimize memory usage and reduce unnecessary background processes.
- Use **lazy loading** for resource-intensive features.
- Profile SDK performance using **Instruments**.

4. Security & Data Privacy

- If the SDK interacts with APIs or stores user data, security becomes a concern.
- GDPR and App Store policies require data handling best practices.

✅ **Solution:**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

- Use **secure encryption techniques** for sensitive data.
- Avoid hardcoded secrets or API keys.
- Implement **SSL pinning** to prevent Man-in-the-Middle (MitM) attacks.

5. Dependency Management

- Developers might not want unnecessary dependencies.
- Compatibility issues may arise due to conflicting versions.

✅ **Solution:**

- Keep dependencies **minimal** and avoid unnecessary third-party libraries.
- Provide **multiple integration options** (CocoaPods, Swift Package Manager, and Carthage).

---

2. How would you design an SDK to be easily integrated by other developers?

A well-designed SDK should be **lightweight, easy to integrate, and developer-friendly**. Here's how I would design it:

1. Keep the API Simple and Intuitive

- Expose only essential methods and minimize complexity.
- Use **fluent APIs** and **builder patterns** where applicable.
- Provide clear method names, return types, and parameters.

✅ **Example:**

```
SDKManager.shared.setup(apiKey: "12345", userID: "56789",
enableLogging: true)
```

Use a fluent API:

```
SDKManager.shared.configure()
    .setAPIKey("12345")
    .setUserID("56789")
    .enableLogging(true)
    .initialize()
```

2. Provide Multiple Integration Options

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

- Support **CocoaPods, Swift Package Manager (SPM), and Carthage**.
- Offer a **manual installation option** for flexibility.

3. Minimize External Dependencies

- Avoid unnecessary third-party libraries to prevent version conflicts.
- If a library is needed, ensure it's well-maintained and lightweight.

4. Write Comprehensive Documentation

- Include a **quick-start guide**, API reference, and FAQs.
- Provide **code samples** and demo projects.

✅ **Tools for Documentation:**

- Use **Jazzy** to generate API docs from Swift comments.
- Host documentation on **GitHub Pages** or a dedicated website.

5. Provide Proper Error Handling & Debugging Support

- Use **meaningful error messages** instead of generic ones.
- Implement proper **logging mechanisms** for debugging.
- Offer a built-in **debug mode** for detailed logs.

✅ **Example:**

```
enum SDKError: Error {
    case invalidAPIKey
    case networkError(message: String)
}
```

---

3. What are some best practices for maintaining backward compatibility in an SDK?

Maintaining backward compatibility is crucial to ensure smooth upgrades for developers using the SDK.

1. Avoid Breaking Changes

- Do not remove or rename public APIs without deprecating them first.
- Use **default parameter values** instead of creating new methods.

✅ **Example:** Instead of removing a function:

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
func fetchData() // Old method (do not remove)
```

Deprecate it and introduce a new one:

```
@available(*, deprecated, message: "Use fetchData(completion:)
instead.")
func fetchData() {}

func fetchData(completion: @escaping (Result<Data, Error>) -> Void) {}
```

2. Semantic Versioning (SemVer)

- Follow **Semantic Versioning (MAJOR.MINOR.PATCH)**:
  - **MAJOR:** Breaking changes (e.g., v2.0 → v3.0).
  - **MINOR:** Backward-compatible new features (e.g., v2.1 → v2.2).
  - **PATCH:** Bug fixes (e.g., v2.1.1 → v2.1.2).

3. Maintain Older API Versions

- If a method is deprecated, support both the old and new versions for a transition period.
- Provide clear **migration guides** in documentation.

4. Write Unit & Integration Tests

- Ensure backward compatibility by writing **unit tests** before releasing new versions.
- Run regression tests to prevent accidental breakages.

---

4. How do you ensure an SDK is lightweight and does not impact app performance?

A bulky SDK can slow down the host app. Here's how I keep it lightweight:

1. Reduce External Dependencies

- Use native frameworks instead of adding third-party libraries.
- Offer an **option to exclude unnecessary features**.

2. Optimize Memory Usage

- Avoid **retain cycles** by using `[weak self]` in closures.
- Use **lazy loading** for resources that are not needed immediately.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

✅ **Example:**

```
lazy var imageCache: NSCache<NSString, UIImage> = NSCache()
```

3. Optimize Network Calls

- Use **efficient caching strategies** to minimize API requests.
- Support **gzip compression** to reduce payload size.

4. Support Modular Builds

- Use **dynamic frameworks** to reduce app binary size.
- Allow developers to **enable/disable specific SDK features**.

5. Use Profiling Tools

- Analyze performance with **Xcode Instruments (Time Profiler, Leaks, Allocations, Network Monitor)**.
- Optimize battery consumption and background tasks.

---

5. What are the differences between static and dynamic frameworks in iOS?

Feature	Static Framework	Dynamic Framework
Definition	Compiled into the app's binary at build time.	Loaded at runtime, reducing initial app size.
File Format	`.a` file (with headers).	`.framework` bundle.
Impact on App Size	Increases the app's binary size.	Reduces app binary size since it's loaded dynamically.
Loading Time	Faster because everything is compiled into a single binary.	Slightly slower as it needs to be loaded at runtime.
Code Updates	Requires recompilation of the entire app when updated.	Can be updated independently of the app.
Example Usage	Small utility libraries that don't change often.	Large SDKs that need to be updated frequently (e.g., Firebase).

✅ **When to use Static vs. Dynamic Frameworks?**

- **Use Static Frameworks** for performance-critical code.
- **Use Dynamic Frameworks** for SDKs or large modules to reduce the initial app size.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

iOS Frameworks & Design Patterns

1. Can you explain the differences between MVC, MVVM, and VIPER architectures?

Architectural patterns help organize code in a structured way, making it more scalable and maintainable. Here's how **MVC, MVVM, and VIPER** differ:

Feature	MVC (Model-View-Controller)	MVVM (Model-View-ViewModel)	VIPER (View-Interactor-Presenter-Entity-Router)
Structure	Divides app into **Model, View, and Controller**.	Introduces a **ViewModel**between Model and View.	Breaks app into **five components**for better separation.
Data Flow	Controller manages UI logic & data flow.	ViewModel handles business logic, and View binds to it.	Each layer has a distinct responsibility, reducing dependencies.
Code Reusability	Low, as the Controller often contains both UI and business logic.	Higher, as ViewModel is independent of View.	Very high, due to strict separation of concerns.
Testability	Low, since Controller is tightly coupled with UI.	Medium, since ViewModel is testable.	Very high, as business logic is separate from UI.
Complexity	Simple, but can lead to **Massive View Controller (MVC)** issue.	More structured but requires binding mechanisms.	More complex, requires additional files per feature.
Use Case	Suitable for small projects.	Good for medium to large projects.	Best for large-scale enterprise apps.

✅ **When to Use?**

- **MVC:** Small projects or quick prototypes.
- **MVVM:** Apps with dynamic UI that benefit from **data binding**.
- **VIPER:** Large, scalable applications where maintainability is key.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

2. What are some advantages and disadvantages of using MVVM over MVC?

Aspect	Advantages of MVVM	Disadvantages of MVVM
Separation of Concerns	Keeps UI logic separate from business logic, making code cleaner.	More layers can introduce **overhead**.
Testability	Easier to write unit tests for the ViewModel since it doesn't depend on UIKit.	Requires understanding of **data-binding**techniques.
Reusability	ViewModels can be shared across multiple Views.	May need additional setup for dependency injection.
Scalability	Handles complex UI logic better with data binding.	More code is required compared to MVC.

✅ **When to Choose MVVM?**

- When **unit testing** is a priority.
- When working with **SwiftUI**, since it natively supports data binding.
- When you want to **reuse ViewModels across different screens**.

---

3. Have you worked with UIKit? Can you explain how AutoLayout works?

Yes, I have extensive experience working with UIKit.

What is AutoLayout?

AutoLayout is a **constraint-based layout system** in iOS that allows developers to create adaptive user interfaces across different screen sizes. Instead of using **frame-based layouts**, AutoLayout defines **relationships (constraints)** between UI elements.

Key Concepts of AutoLayout

1. **Constraints** – Define relationships like height, width, alignment, and spacing.
2. **Intrinsic Content Size** – AutoLayout considers a view's natural size (e.g., a UILabel's text).
3. **Priority** – Constraints have priorities (default: **1000**), allowing flexible UI adjustments.
4. **Hugging & Compression Resistance** – Determines how views expand or shrink.

✅ **Ways to Implement AutoLayout**

- **Storyboard & Interface Builder** – Drag-and-drop constraints visually.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Programmatically (NSLayoutConstraint)
swift

```
myButton.translatesAutoresizingMaskIntoConstraints = false
NSLayoutConstraint.activate([
    myButton.centerXAnchor.constraint(equalTo: view.centerXAnchor),
    myButton.centerYAnchor.constraint(equalTo: view.centerYAnchor),
    myButton.widthAnchor.constraint(equalToConstant: 200),
    myButton.heightAnchor.constraint(equalToConstant: 50)
])
```

- **Using NSLayoutAnchor** (Recommended for clarity).
- **Using UIStackView** – Simplifies layout by managing subviews automatically.

✅ **Benefits of AutoLayout**

- Supports **dynamic screen sizes** (iPhone, iPad, Dynamic Type).
- Works with **localization & right-to-left (RTL) languages**.
- Ensures accessibility compliance.

---

4. What is Combine in Swift? How does it compare to RxSwift?

What is Combine?

Combine is Apple's **reactive programming framework** that enables declarative handling of asynchronous events and data streams.

✅ **Key Features of Combine:**

- Works with **Publishers & Subscribers**.
- Allows **chaining operators** to transform data.
- Reduces reliance on **delegates and completion handlers**.
- Supports **backpressure handling** to manage data flow.

Example of Combine Usage

```
import Combine

let publisher = Just("Hello, Combine!")
let subscriber = publisher.sink { value in
    print(value)  // Output: Hello, Combine!
}
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Combine vs. RxSwift

Feature	Combine	RxSwift
Developed by	Apple (built into iOS 13+)	Community-driven (Open-source)
Installation	No need to install (native)	Requires CocoaPods or Swift Package Manager
Performance	Optimized for Apple's platforms	Slightly more overhead due to cross-platform support
Operator Support	Limited compared to RxSwift	More operators and community support
Community Support	Less mature but growing	Established and widely used

✅ **When to Use?**

- Use **Combine** if targeting iOS 13+ and want a native solution.
- Use **RxSwift** if you need **cross-platform support** (iOS, Android, Web).

---

5. What are the advantages of using SwiftUI over UIKit?

SwiftUI is Apple's modern declarative UI framework, replacing UIKit for building user interfaces.

Feature	SwiftUI	UIKit
Development Style	**Declarative** – UI is described with Swift code.	**Imperative** – Requires manual UI updates.
Code Simplicity	Requires less code for UI updates.	More verbose and requires boilerplate.
Live Previews	Supports **real-time previews** in Xcode.	No live preview support.
Data Binding	Uses `@State`, `@Binding`, `@ObservedObject` for reactive UI updates.	Relies on delegates and target-action pattern.
Cross-Platform	Works on iOS, macOS, watchOS, and tvOS.	Mostly focused on iOS and macOS.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Backward Compatibility	Requires **iOS 13+**.	Works with older iOS versions.
Performance	Optimized for modern Apple devices.	More mature, better optimized for older devices.

Example of SwiftUI Code vs. UIKit

✅ **SwiftUI (Declarative UI)**

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

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

✅ **UIKit (Imperative UI)**

```
class ViewController: UIViewController {
    var count = 0
    let label = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        label.text = "Count: \(count)"
    }

    @objc func increment() {
        count += 1
        label.text = "Count: \(count)"
    }
}
```


Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

✅ **When to Use SwiftUI?**

- For new projects targeting **iOS 13+**.
- When needing **faster UI prototyping**.
- For **cross-platform** Apple ecosystem apps.

✅ **When to Use UIKit?**

- If supporting **iOS 12 or earlier**.
- If requiring advanced **custom UI behaviors**.
- If integrating with legacy UIKit-based projects.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Networking & API Integration

1. How do you handle network calls in iOS using URLSession?

`URLSession` is Apple's built-in networking framework for making HTTP requests, handling downloads, and managing sessions.

Steps to Make a Network Call with URLSession

1. **Create a URL**
2. **Create a URLRequest (optional for customizing headers, HTTP method, etc.)**
3. **Create a URLSession data task**
4. **Handle the response and parse the data**

Here's a simple example of making a **GET request**:

```
import Foundation

func fetchData() {
    guard let url = URL(string: "https://api.example.com/data") else {
return }

    let task = URLSession.shared.dataTask(with: url) { data, response,
error in
        if let error = error {
            print("Error fetching data:
\(error.localizedDescription)")
            return
        }

        guard let httpResponse = response as? HTTPURLResponse,
(200...299).contains(httpResponse.statusCode) else {
            print("Invalid response")
            return
        }

        if let data = data {
            do {
```

```
                let json = try JSONSerialization.jsonObject(with:
data, options: [])
                print("Response JSON: \(json)")
            } catch {
                print("Failed to parse JSON:
\(error.localizedDescription)")
            }
        }
    }

    task.resume() // Don't forget to start the task
}
```

Handling POST Requests with URLSession

For sending data (e.g., login credentials), we create a `URLRequest` with a `POST` method:

```
var request = URLRequest(url: URL(string:
"https://api.example.com/login")!)
request.httpMethod = "POST"
request.addValue("application/json", forHTTPHeaderField:
"Content-Type")

let body = ["username": "user", "password": "pass"]
request.httpBody = try? JSONSerialization.data(withJSONObject: body)

let task = URLSession.shared.dataTask(with: request) { data, response,
error in
    // Handle response...
}
task.resume()
```

✅ **Best Practices with URLSession**

- Always **check HTTP status codes** in responses.
- Use `URLSessionConfiguration` for timeout handling and caching.
- Use **async/await** (iOS 15+) for cleaner async code.
- Handle **errors properly** (network failure, timeout, invalid response).

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

---

2. Have you worked with Alamofire? What are its advantages over URLSession?

Yes, I have experience using **Alamofire**, which is a powerful Swift-based networking library that simplifies working with HTTP requests.

Advantages of Alamofire Over URLSession

Feature	Alamofire	URLSession
Ease of Use	Requires less boilerplate code	Requires manual setup for headers, parameters, etc.
Built-in Features	Supports JSON parsing, authentication, and network retry logic	Must implement manually
Request & Response Handling	Provides **chainable request builders**	Requires completion handlers
Download & Upload Handling	Simplified methods for **multipart uploads**	Requires `URLSessionDownloadTask`
Code Readability	More readable and concise	Requires more setup

Example: Simple GET Request in Alamofire

```
import Alamofire

AF.request("https://api.example.com/data")
    .validate()
    .responseJSON { response in
        switch response.result {
        case .success(let data):
            print("Success: \(data)")
        case .failure(let error):
            print("Error: \(error)")
        }
    }
```


Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

✅ **When to Use Alamofire?**

- When working with **complex API requests** (headers, authentication, multipart uploads).
- When needing **built-in request validation, caching, and error handling**.
- When using **background request handling**.

✅ **When to Use URLSession?**

- When you need **more control over networking** (fine-tuning session configurations).
- When minimizing dependencies in your project.

---

3. How do you handle API authentication in a mobile SDK?

Authentication is crucial for securing API requests in an SDK. The most common authentication methods are:

1️⃣ **OAuth 2.0 (Token-Based Authentication)**

OAuth 2.0 uses **access tokens** to authenticate requests securely.

- The app requests a **token** from an authentication server.
- The token is **stored securely** (e.g., in the Keychain).
- Each API request includes the token in the `Authorization` header.

✅ **Example of Token Authentication in Swift:**

```
var request = URLRequest(url: URL(string:
"https://api.example.com/protected")!)
request.addValue("Bearer \(accessToken)", forHTTPHeaderField:
"Authorization")

let task = URLSession.shared.dataTask(with: request) { data, response,
error in
    // Handle response...
}
task.resume()
```

2️⃣ **API Key Authentication**

- The SDK includes an API key in the request header.
- API keys should be **secured and never hardcoded** in the app.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

✅ **Example:**

```
request.addValue("API_KEY_HERE", forHTTPHeaderField: "x-api-key")
```

3️⃣ **JWT (JSON Web Token) Authentication**

JWT tokens are **signed and encoded JSON objects** used for authentication.

- The client gets a JWT from the server and includes it in API requests.
- JWT can **expire** and be **refreshed** as needed.

✅ **Best Practices for API Authentication:**

- **Never hardcode API keys** in the SDK (use environment variables).
- **Use Secure Enclave or Keychain** to store tokens.
- **Implement refresh tokens** for OAuth-based authentication.
- **Encrypt sensitive data** before storing or transmitting it.

---

4. What are some common security best practices when handling API responses?

✅ **Best Practices for API Security:**

1️⃣ **Use HTTPS Always**

- Ensure all API requests are made over `https://` to prevent **MITM (Man-in-the-Middle) attacks**.

2️⃣ **Validate SSL Certificates**

- Enable **SSL Pinning** to prevent attackers from intercepting requests.

```
let evaluator = PinnedCertificatesTrustEvaluator()
```

3️⃣ **Secure API Keys & Tokens**

- Store API keys in **Keychain** or **encrypted storage** (not UserDefaults).
- Avoid embedding secrets directly in the SDK.

4️⃣ **Limit API Rate and Throttling**

- Protect APIs against **DDoS attacks** using rate limiting.

5️⃣ **Sanitize and Validate Input**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

- Prevent **SQL Injection** and **Cross-Site Scripting (XSS)**.

6️⃣ **Avoid Logging Sensitive Data**

- Disable logging for authentication headers and API keys.

---

5. How would you implement caching for API responses in an SDK?

Caching improves performance by reducing redundant network requests.

✅ **Approaches to Implement Caching:**

1️⃣ **Using URLCache (Built-in iOS Caching)**

- iOS provides a default `URLCache` to cache HTTP responses.

```
let config = URLSessionConfiguration.default
config.urlCache = URLCache(memoryCapacity: 10 * 1024 * 1024,
diskCapacity: 50 * 1024 * 1024, diskPath: nil)

let session = URLSession(configuration: config)
```

2️⃣ **Custom Disk-Based Caching (Using File Storage or CoreData)**

- Store API responses in a **database (CoreData/SQLite)** or **filesystem** for long-term storage.

3️⃣ **Using Third-Party Libraries (Like Cache or Realm)**

- The **Cache** library provides an easy-to-use caching layer.

```
import Cache

let storage = try? Storage<String, Data>(
    diskConfig: DiskConfig(name: "APICache"),
    memoryConfig: MemoryConfig(expiry:
.date(Date().addingTimeInterval(3600))),
    transformer: TransformerFactory.forData()
)

// Save Data
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
try? storage?.setObject(responseData, forKey: "userProfile")

// Fetch Data
let cachedData = try? storage?.object(forKey: "userProfile")
```

✅ **When to Use Caching?**

- **For API responses that don't change frequently** (e.g., static content).
- **To improve app performance** and reduce server load.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Dependency Management

1. What are the differences between CocoaPods, Carthage, and Swift Package Manager?

CocoaPods, Carthage, and Swift Package Manager (SPM) are three popular dependency managers for iOS development, each with its own advantages and use cases.

Feature	CocoaPods	Carthage	Swift Package Manager (SPM)
Installation	Modifies the Xcode project	Builds frameworks without modifying Xcode project	Integrated with Xcode
Dependency Resolution	Uses a central `Podfile` and `Podfile.lock`	Requires manual framework linking	Uses `Package.swift`
Binary Support	Supports prebuilt frameworks	Supports prebuilt frameworks	Limited support (SPM 5.3+ supports binary targets)
Speed	Can be slow due to managing dependencies in `Pods/` folder	Faster as it only downloads what is needed	Faster as it's built into Xcode
Framework Type	Uses static or dynamic frameworks	Prefers dynamic frameworks	Uses static linking by default
Ease of Use	Simple but modifies the project structure	Requires manual setup but is lightweight	Easiest as it integrates natively with Xcode
Future Support	Still widely used, but declining	Less popular now	Recommended by Apple

When to Use Which?

- **CocoaPods**: If you need extensive library support and want easy dependency resolution.
- **Carthage**: If you want more control over dependency management and prefer manual linking.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

- **SPM**: The preferred choice for new projects as it is natively integrated with Xcode and officially supported by Apple.

---

2. How do you manage dependencies in an SDK while keeping it lightweight?

When developing an iOS SDK, it is crucial to manage dependencies efficiently to keep it **lightweight, maintainable, and fast**. Here's how:

1️⃣ Minimize External Dependencies

- **Avoid heavy third-party libraries** unless absolutely necessary.
- Prefer **native iOS frameworks** (`Foundation`, `Combine`, `CoreGraphics`, etc.).
- If needed, use **lightweight third-party libraries** instead of bloated ones.

2️⃣ Use Swift Package Manager (SPM) When Possible

- SPM does not require additional installation and does not modify the project structure.
- Supports modular dependency resolution, making the SDK easier to integrate.

3️⃣ Use Dynamic Frameworks for Optional Dependencies

- If your SDK requires optional dependencies, use **dynamic frameworks** so the app developer can choose to include them.

Example:

```
#if canImport(Alamofire)
import Alamofire
#endif
```

4️⃣ Optimize Code for Performance

- Use **lazy loading** and **on-demand resource fetching** instead of bundling large assets.
- **Minimize memory footprint** by using weak references and avoiding unnecessary data storage.

5️⃣ Provide Multiple Integration Options

- Offer installation via **CocoaPods, Carthage, and SPM** to make integration flexible for developers.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Example `Package.swift` for SPM:

```
let package = Package(
    name: "MySDK",
    platforms: [.iOS(.v13)],
    products: [
        .library(name: "MySDK", targets: ["MySDK"])
    ],
    dependencies: [],
    targets: [
        .target(name: "MySDK", dependencies: [])
    ]
)
```

✅ **Key Takeaways:**

- Use native frameworks where possible.
- Avoid excessive third-party dependencies.
- Offer dynamic framework support for optional dependencies.
- Use **Swift Package Manager** for best performance.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Testing & Debugging

1. What is XCTest, and how do you write unit tests for an iOS SDK?

What is XCTest?

`XCTest` is Apple's official testing framework for unit testing and UI testing in iOS, macOS, watchOS, and tvOS applications. It allows developers to **write, run, and analyze tests** to ensure code correctness, reliability, and performance.

How to Write Unit Tests for an iOS SDK?

1️⃣ **Create a Test Target**

- In Xcode, go to **File > New > Target** and add a **Unit Testing Bundle**.
- Ensure the SDK target is included in the **"Target Membership"** of the test files.

2️⃣ **Import XCTest and Your SDK**

- Create a test case file: `MySDKTests.swift`.

Import XCTest and the SDK:

```
import XCTest
@testable import MySDK  // Allows testing internal SDK components
```

3️⃣ **Write a Unit Test Example**

A basic test function follows the `XCTestCase` subclass structure:
swift

```
class MySDKTests: XCTestCase {
    func testExample() {
        let expectedValue = 5
        let result = MySDK.someFunction()
        XCTAssertEqual(result, expectedValue, "Result should be 5")
    }
}
```

4️⃣ **Run Tests in Xcode**

- Open **Xcode Test Navigator (⌘ + 6)** → Click the play button next to the test case.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

- Alternatively, run tests using **Cmd + U**.

5️⃣ **Mock Dependencies**

If your SDK depends on APIs or databases, use **mock objects** to isolate logic. Example using a mock network request:

```
class MockAPIClient: APIClient {
    override func fetchData() -> String {
        return "Mock Data"
    }
}
```

---

2. How do you perform UI testing in an iOS app?

What is UI Testing?

UI Testing verifies the behavior of the app's interface by simulating user interactions (e.g., tapping buttons, entering text). XCTest provides **XCUITest** to automate these interactions.

Steps for UI Testing

1️⃣ **Create a UI Test Target**

- In Xcode, go to **File > New > Target** and select **UI Testing Bundle**.

2️⃣ **Write a Basic UI Test**

Example test that verifies if a button tap updates a label:

```
import XCTest

class MyAppUITests: XCTestCase {
    let app = XCUIApplication()

    override func setUp() {
        continueAfterFailure = false
        app.launch()
    }

    func testButtonTapUpdatesLabel() {
        let button = app.buttons["TapButton"]
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
        let label = app.staticTexts["ResultLabel"]

        button.tap()
        XCTAssertEqual(label.label, "Button Pressed", "Label should
update after button tap")
    }
}
```

3️⃣**Use Accessibility Identifiers**

Ensure UI elements have **accessibility identifiers**:

```
button.accessibilityIdentifier = "TapButton"
label.accessibilityIdentifier = "ResultLabel"
```

4️⃣**Run the UI Tests**

- Run using **Cmd + U** or via **Xcode Test Navigator (⌘ + 6)**.
- Xcode launches the app and simulates interactions.

---

3. What are some common debugging tools you use in Xcode?

1️⃣**LLDB Debugger (Console & Breakpoints)**

- Allows stepping through code, inspecting variables, and evaluating expressions.

Example: Set a breakpoint and use `po` to print variables:

```
po myVariable  // Prints object description
```

2️⃣**Xcode Instruments**

- Used for **profiling memory, CPU usage, and performance bottlenecks**.
- Common Instruments:
    - **Leaks** → Detects memory leaks
    - **Time Profiler** → Measures CPU performance
    - **Allocations** → Tracks memory usage

3️⃣**View Debugger**

- Allows inspecting UI hierarchies by pausing the app at a breakpoint and clicking **Debug View Hierarchy**.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

4️⃣ **Network Debugging (Using Charles Proxy or Xcode Network Monitor)**

- Xcode provides a **Network Instrument** to inspect API requests.
- Example: Use **Breakpoints on URLSession** to inspect outgoing network calls.

5️⃣ **Console Logs & OSLog**

Use `print()` for basic logs, but prefer **os_log** for production-level logging:

```
import os

let logger = Logger(subsystem: "com.myapp", category: "network")
logger.info("API Request Started")
```

-

---

4. How do you analyze memory leaks in an iOS app?

Memory leaks occur when objects are **retained in memory** but never released. iOS uses **Automatic Reference Counting (ARC)**, but retain cycles can cause leaks.

How to Detect Memory Leaks?

1️⃣ **Use Xcode Instruments - Leaks Tool**

- Open **Xcode > Product > Profile**.
- Select **Leaks** and monitor memory allocations in real time.
- If an object is **not deallocated**, it may be leaking.

2️⃣ **Check Retain Cycles Using Debug Memory Graph**

- Run the app in Debug mode and **click "Debug Memory Graph"**.
- Inspect which objects are still in memory unexpectedly.

3️⃣ **Break Strong Reference Cycles**

Common issues occur with **strong references in closures**:

```
class SomeClass {
    var closure: (() -> Void)?

    func setup() {
        closure = { [weak self] in
            self?.doSomething()
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
        }
    }
}
```

Using `[weak self]` prevents strong references to `self`.

4️⃣ Use Weak or Unowned References in Delegates

Always use `weak` for delegate properties to prevent retain cycles:

```
protocol MyDelegate: AnyObject {
    func didFinishTask()
}

class SomeClass {
    weak var delegate: MyDelegate?
}
```

5️⃣ Test for Memory Leaks in Unit Tests

XCTest can check if an object is deallocated correctly:

```
func testMemoryLeak() {
    weak var object = MyClass()
    XCTAssertNil(object, "Object should be deallocated")
}
```


Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Performance Optimization & CI/CD

1. What are some techniques to optimize an SDK for better performance?

Optimizing an SDK ensures that it remains **lightweight, efficient, and easy to integrate** without affecting the performance of the host application. Below are some key techniques:

✅ **Reduce Binary Size**

- **Use Bitcode and App Thinning**: Allows Apple to optimize the final build size dynamically.

Strip Unused Architectures: If distributing a binary framework, remove unnecessary architectures (e.g., Simulator slices) using:
sh

```
lipo -remove x86_64 MySDK.framework/MySDK -output
MySDK.framework/MySDK
```

-
- **Use Static Libraries Where Possible**: Static frameworks avoid runtime overhead compared to dynamic ones.

✅ **Optimize Memory Usage**

- **Avoid Strong Reference Cycles**: Use `[weak self]` in closures to prevent retain cycles.
- **Use Value Types Instead of Classes**: Prefer `structs` where possible to reduce heap allocations.
- **Lazy Loading & Caching**: Load heavy assets **only when needed** to reduce memory consumption.

✅ **Reduce CPU Usage**

- **Optimize Data Processing**: Use efficient algorithms and avoid unnecessary computations.

Leverage Background Threads: Perform heavy tasks (e.g., networking, data processing) on background threads using:

```
DispatchQueue.global(qos: .background).async {
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

```
    // Heavy computation
    DispatchQueue.main.async {
        // Update UI
    }
}
```

- **Batch API Calls**: Reduce the number of network requests by batching them when possible.

✅ **Optimize Network Requests**

- **Use HTTP/2 and Keep-Alive Connections**: Reduces latency and improves network efficiency.
- **Enable Compression**: Use `Accept-Encoding: gzip, deflate` in API requests to reduce payload size.
- **Implement Efficient Caching**: Store API responses and images to minimize repeated network calls.

✅ **Use Instruments for Profiling**

- Use **Time Profiler** to detect CPU bottlenecks.
- Use **Memory Graph Debugger** to detect memory leaks.

---

2. How do you measure and optimize battery efficiency in an iOS application?

Battery efficiency is crucial for mobile applications and SDKs. Poorly optimized code can drain battery life by using excessive CPU, network, and GPU resources.

✅ **Measuring Battery Efficiency**

- **Use Xcode Energy Gauge** (`Debug Navigator > Energy Impact`) to monitor energy consumption.
- **Profile with Instruments – Energy Log** to analyze CPU, network, and background activity.
- **Use Power Metrics Instruments** to check CPU wake-ups, background tasks, and energy impact.

✅ **Optimization Techniques**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

1️⃣ **Reduce CPU & GPU Usage**

- Use efficient data structures and algorithms.
- Offload complex tasks to background threads (`DispatchQueue.global`).
- Reduce UI redraws by minimizing unnecessary animations and layout changes.

2️⃣ **Optimize Network Requests**

- **Batch Requests**: Combine multiple small requests into one to reduce network activity.
- **Use Background Fetch & Push Notifications** instead of frequent polling.
- **Implement HTTP Caching** to avoid unnecessary downloads.

3️⃣ **Minimize Location & Bluetooth Usage**

- Use **"when in use"** authorization instead of **"always"**.

Reduce GPS updates with `desiredAccuracy` and `distanceFilter`:
swift

```
locationManager.desiredAccuracy = kCLLocationAccuracyKilometer

locationManager.distanceFilter = 500
```

- Avoid unnecessary Bluetooth scanning to prevent battery drain.

4️⃣ **Reduce Background Activity**

- **Use Background Modes Wisely**: Don't keep unnecessary tasks running in the background.
- **Limit Timer Intervals**: Avoid frequent timers that wake up the CPU unnecessarily.

---

3. Have you worked with CI/CD pipelines for iOS development? How would you automate SDK releases?

Yes, CI/CD (Continuous Integration & Continuous Deployment) is essential for automating builds, tests, and deployments in iOS SDK development.

✅ **Setting Up a CI/CD Pipeline for iOS SDK Releases**

1️⃣ **Choose a CI/CD Platform**

- Popular CI/CD tools for iOS:
  - **GitHub Actions**
  - **Bitrise**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

- **CircleCI**
- **Jenkins**
- **Fastlane**

2️⃣ **Automate Builds and Tests**

Use `xcodebuild` or `Fastlane` to build and test the SDK:
sh

```
xcodebuild clean build -workspace MySDK.xcworkspace \
  -scheme MySDK -destination 'generic/platform=iOS'
```

Run unit tests with XCTest:
sh

```
xcodebuild test -scheme MySDK -destination 'platform=iOS Simulator,name=iPhone 15'
```

3️⃣ **Automate Versioning**

Use `agvtool` to automatically increment build numbers:
sh

```
agvtool next-version -all
```

4️⃣ **Package the SDK**

Use `xcodebuild archive` to generate the framework:
sh

```
xcodebuild archive -scheme MySDK -archivePath MySDK.xcarchive
```

Export `.xcframework`:
sh

```
xcodebuild -create-xcframework \
  -framework MySDK.xcarchive/Products/Library/Frameworks/MySDK.framework \
  -output MySDK.xcframework
```

5️⃣ **Automate Deployment**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)

Publish the SDK to CocoaPods:
sh

```
pod trunk push MySDK.podspec
```

Publish to Swift Package Manager (SPM) by tagging a release in GitHub:
sh

```
git tag 1.0.0

git push origin 1.0.0
```

- Deploy artifacts to a private/internal repo (e.g., Artifactory, GitHub Packages).

6️⃣ Automate Documentation

Use **Jazzy** to generate documentation:
sh

```
jazzy --clean --author "Your Name" --author_url
"https://yourwebsite.com"
```

7️⃣ Code Signing & Security

- Secure API keys & credentials using **environment variables or encrypted secrets** in CI/CD.

Use **Apple Developer Certificates** and manage provisioning profiles automatically via Fastlane:
sh

```
fastlane match development
```


Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - +91 9228877722)