Senior iOS Developer SwiftUI Interview Questions (2025 Edition)

✅ Q1: How would you structure a large-scale SwiftUI application?

A large-scale SwiftUI application should be structured using modular, scalable, and maintainable practices. A common structure would be:

1. **Adopt MVVM or Clean Architecture**
    - **Model**: Represents the data layer (Core Data, API responses, etc.).
    - **ViewModel**: Handles business logic and state management.
    - **View**: The SwiftUI components, driven by the ViewModel.
2. **Use Feature-Based Modules**
    - Separate different parts of the app into modules (e.g., Authentication, Dashboard, Profile, etc.).
    - Each module has its own Model, ViewModel, and View.
3. **Dependency Injection (DI)**
    - Use `@EnvironmentObject`, `@StateObject`, or manual DI to pass dependencies.
    - Leverage DI frameworks like Resolver or Factory pattern.
4. **Leverage Protocol-Oriented Programming**
    - Abstract dependencies using protocols to improve testability and flexibility.
5. **Manage Global State Efficiently**
    - Use `@StateObject` for ownership.
    - Use `@EnvironmentObject` for shared state.
    - Consider `ObservableObject` and `Combine` for reactivity.

✅ Q2: How do you implement the MVVM pattern in SwiftUI? Why is it preferred over MVC?

Implementing MVVM in SwiftUI

Model (Data representation):

```
struct User: Identifiable {
    let id: UUID
    let name: String
    let age: Int
}
```

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

ViewModel (Handles logic and state):

```
class UserViewModel: ObservableObject {
    @Published var users: [User] = []

    func fetchUsers() {
        users = [User(id: UUID(), name: "John Doe", age: 25)]
    }
}
```

1.

View (SwiftUI UI representation):

```
struct UserView: View {
    @StateObject private var viewModel = UserViewModel()

    var body: some View {
        List(viewModel.users) { user in
            Text("\(user.name), \(user.age) years old")
        }
        .onAppear {
            viewModel.fetchUsers()
        }
    }
}
```

2.

Why is MVVM preferred over MVC?

- **Separation of Concerns**: MVVM ensures UI logic (ViewModel) is separate from UI rendering (View).
- **Easier Testing**: Business logic is isolated in ViewModel, making unit testing simpler.
- **Better State Management**: SwiftUI's `@StateObject` and `@Published` work seamlessly with ViewModel.
- **Scalability**: With MVC, `UIViewController` often becomes a massive file (Massive View Controller problem). MVVM prevents this by separating concerns.

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

✅ Q3: How does SwiftUI manage the view hierarchy, and what optimizations would you recommend for a complex UI?

SwiftUI's View Hierarchy Management

- SwiftUI is declarative and re-renders views only when state changes.
- It uses a **diffing mechanism** to update only the necessary subviews.
- Views are lightweight because they are just **structs** (no heavy memory footprint).

Optimizations for Complex UI

1. **Use `@State`, `@StateObject`, `@ObservedObject`, and `@EnvironmentObject` wisely**
   - Avoid unnecessary re-renders by placing state at the right level.
2. **Lazy Loading for Large Lists**
   - Use `LazyVStack` or `LazyHStack` inside a `ScrollView` to load views only when needed.

swift

```
ScrollView {
    LazyVStack {
        ForEach(0..<1000, id: \.self) { index in
            Text("Item \(index)")
        }
    }
}
```

3. **Minimize View Updates with `EquatableView`**
   - Use `EquatableView` to avoid unnecessary redraws of complex views.
4. **Decompose Large Views into Smaller Components**
   - Instead of one large view, split it into multiple smaller reusable subviews.
5. **Use `GeometryReader` Sparingly**
   - It triggers re-computation frequently. Use only when layout customization is needed.
6. **Optimize Animations**
   - Prefer `withAnimation` and avoid heavy animations inside lists.

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

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

✅ Q4: How do you handle dependency injection in SwiftUI?

SwiftUI supports dependency injection through `@EnvironmentObject`, `@StateObject`, and manual DI.

1. Using `@EnvironmentObject` (Recommended for global state)

Define a shared object:

```
class AuthManager: ObservableObject {
    @Published var isLoggedIn = false
}
```

Inject it into the environment:

```
@main
struct MyApp: App {
    @StateObject private var authManager = AuthManager()

    var body: some Scene {
        WindowGroup {
            ContentView().environmentObject(authManager)
        }
    }
}
```

Access it in views:

```
struct DashboardView: View {
    @EnvironmentObject var authManager: AuthManager

    var body: some View {
        Text(authManager.isLoggedIn ? "Welcome Jemini" : "Please Log
In")
    }
}
```

2. Using Constructor Injection (Best for Local Dependencies)

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

```
struct UserProfileView: View {
    let viewModel: UserProfileViewModel

    var body: some View {
        Text(viewModel.username)
    }
}

struct ContentView: View {
    var body: some View {
        UserProfileView(viewModel: UserProfileViewModel(userID: 1))
    }
}
```

3. Using Resolver (Third-Party DI Library)

- Use **Resolver** for dependency injection to avoid manually passing objects.

✅ Q5: What are the best practices for organizing reusable SwiftUI components?

1. **Follow the Single Responsibility Principle**
    - Each component should have a specific purpose.
2. **Extract Common UI Elements**
    - Example: A reusable button component.

```
struct CustomButton: View {

    let title: String
    let action: () -> Void

    var body: some View {
        Button(action: action) {
            Text(title)
                .padding()
                .background(Color.blue)
                .foregroundColor(.white)
```

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

```
                .cornerRadius(10)
        }
    }
}
```

3. **Use ViewModifiers for Styling**

```
  struct TitleStyle: ViewModifier {

    func body(content: Content) -> some View {
        content
            .font(.largeTitle)
            .foregroundColor(.blue)
    }
}

extension View {
    func titleStyle() -> some View {
        self.modifier(TitleStyle())
    }
}
```

4. **Create Custom Environment Objects for Global State**
    - Store global UI settings in an `EnvironmentObject`.
5. **Use Previews Extensively**
    - Provide multiple preview cases for different UI states.

✅ Q6: How does SwiftUI's diffing algorithm work, and how can you optimize view updates?

SwiftUI's Diffing Algorithm

- SwiftUI **compares the new and old state of the view hierarchy** and updates only the parts that changed.
- It uses **identity (`id`) and value comparisons** to determine which views should be updated, added, or removed.
- Views are **lightweight structs**, so SwiftUI efficiently replaces them when needed.
- If a view's identity remains the same, SwiftUI **updates only the modified properties** rather than re-creating the entire view.

Optimizing View Updates

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

Use Identifiable Data Models

```
struct User: Identifiable {
    let id: UUID
    let name: String
}
```

1. Ensures SwiftUI can track each item's state efficiently.
2. **Use `@State` and `@Binding` for Local State Changes**
    - Avoid unnecessary `@StateObject` usage, which triggers global updates.

Use `EquatableView` for Expensive Views

```
struct MyView: Equatable {
    let value: Int

    var body: some View {
        Text("Value: \(value)")
    }

    static func == (lhs: MyView, rhs: MyView) -> Bool {
        lhs.value == rhs.value
    }
}
```

3. Prevents unnecessary re-renders when values haven't changed.

Minimize Computed Properties Inside `body`

```
var body: some View {
    Text("\(expensiveCalculation())")  // Avoid
}
```

4. Move computations **outside** the view to prevent recalculations.
5. **Use `@StateObject` Wisely**
    - Place `@StateObject` in a parent view to **prevent recreation of objects on re-render**.

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

✅ Q7: What is the difference between `@State`, `@Binding`, `@StateObject`, and `@ObservedObject` in terms of performance?

Property	Ownership	Used For	Performance Considerations
`@State`	View owns it	Simple local state	Best for small, simple states (primitives, booleans)
`@Binding`	Passed from parent	Child modifies parent state	Avoids unnecessary re-renders of parent views
`@StateObject`	View owns it	ObservableObject lifetime management	Prevents object recreation on every re-render
`@ObservedObject`	Passed into a View	ObservableObject from elsewhere	View refreshes whenever object changes

Performance Best Practices

- Use `@State` for lightweight properties inside a view.
- Use `@Binding` to pass state **without recreating objects**.
- Use `@StateObject` when a view **creates and owns the object**.
- Use `@ObservedObject` when injecting an object **from outside** the view hierarchy.

✅ Q8: How would you improve scrolling performance in a `List` with thousands of items?

Performance Optimization Strategies

1. **Use `LazyVStack` inside `List`**
    - Default `List` is already optimized, but use `LazyVStack` if using a `ScrollView`.

Avoid ForEach without `id`
swift

```
List(users) { user in
    Text(user.name)
}
```

2. Ensures SwiftUI efficiently tracks changes.

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

Use `@StateObject` Instead of `@ObservedObject`
swift

```
struct UserListView: View {
    @StateObject private var viewModel = UserViewModel()
}
```

3.
    - Prevents the ViewModel from being recreated on every re-render.
4. **Use Pagination or Lazy Loading**
    - Load only the required data when scrolling reaches the end.

Use `redacted(reason:)` for Placeholder Loading
swift

```
Text("Loading...").redacted(reason: .placeholder)
```

5. Enhances perceived performance.

Optimize Images with Async Loading

```
AsyncImage(url: URL(string: user.imageUrl)) { image in
    image.resizable().scaledToFit()
} placeholder: {
    ProgressView()
}
```

6. Prevents blocking the main thread.

✅ Q9: When should you use `LazyVStack` vs `VStack`, and why?

Property	`VStack`	`LazyVStack`
View Loading	Loads all views at once	Loads only visible views
Performance	Heavy for large lists	Optimized for large lists
Use Case	Small static UI elements	Long scrollable content

When to Use `LazyVStack`

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

✅ When working with a `ScrollView` containing **a large number of items**.
✅ When **memory efficiency** is needed (e.g., dynamic content).

When to Use `VStack`

✅ When views **must all be loaded immediately** (e.g., static layouts).
✅ When **content size is small** (e.g., form inputs).

Example:

```
ScrollView {
    LazyVStack {
        ForEach(1..<1000) { index in
            Text("Item \(index)")
        }
    }
}
```

- Prevents memory spikes compared to using a regular `VStack`.

✅ Q10: What are the common performance pitfalls in SwiftUI and how do you avoid them?

1. Unnecessary View Re-Renders

❌ **Bad:**

```
var body: some View {
    Text("\(expensiveCalculation())")
}
```

✅ **Fix:**

```
let computedValue = expensiveCalculation()
var body: some View {
    Text("\(computedValue)")
}
```

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

2. Using `ForEach` Without Identifiable Data

❌ **Bad:**

```
ForEach(0..<1000) { index in
    Text("Item \(index)")
}
```

✅ **Fix:**

```
ForEach(items, id: \.id) { item in
    Text(item.name)
}
```

3. Overusing `GeometryReader`

- `GeometryReader` recalculates layout frequently.
  ✅ **Use only when necessary** for layout adjustments.

4. Blocking the Main Thread with Heavy Computations

❌ **Bad:**

```
var body: some View {
    Text(complexFunction()) // Runs on main thread
}
```

✅ **Fix:**

```
@State private var result: String = ""

var body: some View {
    Text(result)
        .onAppear {
            DispatchQueue.global().async {
                let computed = complexFunction()
                DispatchQueue.main.async {
```


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

```
                    result = computed
                }
            }
        }
}
```

5. Excessive `@StateObject` Usage

- `@StateObject` should only be used when the view **creates and owns** the object.
  ✅ **Fix:** Use `@ObservedObject` when injecting an object from elsewhere.

✅ Q11: How does SwiftUI handle concurrency with `async/await`?

SwiftUI integrates seamlessly with Swift's `async/await` to perform asynchronous tasks efficiently without blocking the main thread.

How SwiftUI Uses `async/await`

- Used for **network calls, file I/O, and background tasks** without blocking the UI.

OnAppear for async tasks:

```
struct ContentView: View {
    @State private var data: String = "Loading..."

    var body: some View {
        Text(data)
            .onAppear {
                Task {
                    data = await fetchData()
                }
            }
    }

    func fetchData() async -> String {
        try? await Task.sleep(nanoseconds: 2_000_000_000) // Simulate
delay
```

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

```
        return "Fetched Data"
    }
}
```

- **Button with async function:**

```
  Button("Fetch Data") {

    Task {
        data = await fetchData()
    }
}
```

- `Task {}` ensures SwiftUI doesn't block the main thread.

✅ Q12: How would you integrate `Combine` with SwiftUI?

`Combine` is a reactive framework that integrates well with SwiftUI's state-driven updates.

1. Using `@Published` and `ObservableObject` with SwiftUI

```
import Combine

class ViewModel: ObservableObject {
    @Published var data: String = "Loading..."
    private var cancellables = Set<AnyCancellable>()

    func fetchData() {
        Just("Fetched Data")
            .delay(for: .seconds(2), scheduler: DispatchQueue.main)
            .sink { [weak self] value in
                self?.data = value
            }
            .store(in: &cancellables)
    }
}

struct ContentView: View {
    @StateObject private var viewModel = ViewModel()
```

```
    var body: some View {
        Text(viewModel.data)
            .onAppear {
                viewModel.fetchData()
            }
    }
}
```

2. Combine for Networking with SwiftUI

```
import Combine

class NetworkManager: ObservableObject {
    @Published var data: String = "Loading..."
    private var cancellables = Set<AnyCancellable>()

    func fetchData() {
        URLSession.shared.dataTaskPublisher(for: URL(string:
"https://api.example.com")!)
            .map { String(decoding: $0.data, as: UTF8.self) }
            .replaceError(with: "Error")
            .receive(on: DispatchQueue.main)
            .sink { [weak self] value in
                self?.data = value
            }
            .store(in: &cancellables)
    }
}
```

- Combine helps handle asynchronous data streams efficiently in SwiftUI.

✅ Q13: What is the role of `Task`, `@MainActor`, and `async let` in SwiftUI concurrency?

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

Concept	Purpose
`Task {}`	Runs an **async** operation in a structured manner.
`@MainActor`	Ensures code **runs on the main thread** (important for UI updates).
`async let`	Runs multiple **independent** async calls **concurrently** for better performance.

1. `Task {}` for Managing Asynchronous Work

```
Task {
    let data = await fetchData()
    print(data)
}
```

- Ensures work is done **on a background thread** and updates happen safely.

2. `@MainActor` for UI Updates

```
@MainActor
class ViewModel: ObservableObject {
    @Published var data: String = "Loading..."

    func fetchData() async {
        let result = await networkRequest()
        data = result
    }
}
```

- Ensures `data` updates **on the main thread**, preventing UI issues.

3. `async let` for Concurrent Calls

```
async let data1 = fetchData()
async let data2 = fetchMoreData()
```

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

```
let results = await (data1, data2)
```

- Runs tasks **concurrently**, improving performance.

✅ Q14: How do you handle network calls efficiently in a SwiftUI app?

1. Use `async/await` for Clean Networking

```
struct APIService {
    static func fetchData() async throws -> String {
        let url = URL(string: "https://api.example.com")!
        let (data, _) = try await URLSession.shared.data(from: url)
        return String(decoding: data, as: UTF8.self)
    }
}


class ViewModel: ObservableObject {
    @Published var data: String = "Loading..."

    func loadData() async {
        do {
            data = try await APIService.fetchData()
        } catch {
            data = "Error fetching data"
        }
    }
}
```

2. Use Background Task with `Task {}`

```
Task {
    await viewModel.loadData()
}
```


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

- Prevents **blocking the UI** while fetching data.

3. Use Caching to Improve Performance

- Use `NSCache` or `UserDefaults` to cache frequently used API responses.

4. Use `Combine` for Real-Time Streaming

```
URLSession.shared.dataTaskPublisher(for: URL(string:
"https://api.example.com")!)
    .map { String(decoding: $0.data, as: UTF8.self) }
    .replaceError(with: "Error")
    .receive(on: DispatchQueue.main)
    .sink { print($0) }
```

- Useful for **real-time updates** like stock prices.

✅ Q15: What are `TaskGroup` and `Actors`, and how would you use them in SwiftUI?

1. `TaskGroup` for Parallel Task Execution

- **Used to launch multiple concurrent tasks efficiently.**

✅ **Example:** Fetching multiple APIs concurrently

```
import Foundation

struct APIService {
    static func fetchData(_ id: Int) async -> String {
        return "Data \(id)"
    }
}

func fetchAllData() async {
    await withTaskGroup(of: String.self) { group in
        for i in 1...3 {
```


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

```
            group.addTask {
                return await APIService.fetchData(i)
            }
        }

        for await result in group {
            print(result)
        }
    }
}
```

- **Runs multiple API calls concurrently** and gathers results efficiently.

---

2. `Actors` for Thread-Safe State Management

- **Ensures safe access to shared data across threads** without using locks.

✅ **Example:**

```
actor DataStore {
    private var data: [String] = []

    func addData(_ value: String) {
        data.append(value)
    }

    func fetchData() -> [String] {
        return data
    }
}

let store = DataStore()

Task {
    await store.addData("New Item")
    let allData = await store.fetchData()
```


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

```
    print(allData)
}
```

- **Prevents race conditions** when multiple tasks access shared data.

✅ Q16: How do you implement an API layer using `async/await` in SwiftUI?

Creating a dedicated API layer improves **code organization, reusability, and testability**.

1. Define the API Client

```
import Foundation

struct APIClient {
    static let shared = APIClient()

    func fetchData<T: Decodable>(from url: URL) async throws -> T {
        let (data, response) = try await URLSession.shared.data(from:
url)

        guard let httpResponse = response as? HTTPURLResponse,
httpResponse.statusCode == 200 else {
            throw URLError(.badServerResponse)
        }

        return try JSONDecoder().decode(T.self, from: data)
    }
}
```

2. Create a Service for Fetching Data

```
struct User: Codable, Identifiable {
    let id: Int
    let name: String
}
```

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

```
struct UserService {
    static let baseURL = "https://jsonplaceholder.typicode.com"

    static func getUsers() async throws -> [User] {
        guard let url = URL(string: "\(baseURL)/users") else {
            throw URLError(.badURL)
        }
        return try await APIClient.shared.fetchData(from: url)
    }
}
```

3. Use it in a SwiftUI ViewModel

```
@MainActor
class UserViewModel: ObservableObject {
    @Published var users: [User] = []
    @Published var errorMessage: String?

    func loadUsers() async {
        do {
            users = try await UserService.getUsers()
        } catch {
            errorMessage = error.localizedDescription
        }
    }
}
```

4. Integrate with SwiftUI View

```
struct ContentView: View {
    @StateObject private var viewModel = UserViewModel()

    var body: some View {
        List(viewModel.users) { user in
            Text(user.name)
        }
```


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

```
        .task {
            await viewModel.loadUsers()
        }
        .alert(viewModel.errorMessage ?? "", isPresented:
.constant(viewModel.errorMessage != nil)) {
            Button("OK", role: .cancel) { viewModel.errorMessage = nil
}
        }
    }
}
```

✅ Q17: How do you cache API responses efficiently in a SwiftUI app?

1. Use NSCache for Memory Caching

```
class APICache {
    static let shared = NSCache<NSString, NSData>()

    func getData(for key: String) -> Data? {
        return APICache.shared.object(forKey: key as NSString) as
Data?
    }

    func setData(_ data: Data, for key: String) {
        APICache.shared.setObject(data as NSData, forKey: key as
NSString)
    }
}


func fetchCachedData(from url: URL) async throws -> Data {
    if let cachedData = APICache.shared.getData(for:
url.absoluteString) {
        return cachedData
    }

    let (data, _) = try await URLSession.shared.data(from: url)
```

```
    APICache.shared.setData(data, for: url.absoluteString)
    return data
}
```

2. Use `UserDefaults` for Lightweight Caching

```
func saveToUserDefaults(_ data: String) {
    UserDefaults.standard.set(data, forKey: "cachedResponse")
}

func loadFromUserDefaults() -> String? {
    return UserDefaults.standard.string(forKey: "cachedResponse")
}
```

3. Use File Storage for Large Data

```
func saveDataToFile(_ data: Data, filename: String) {
    let fileURL =
FileManager.default.temporaryDirectory.appendingPathComponent(filename
)
    try? data.write(to: fileURL)
}

func loadDataFromFile(filename: String) -> Data? {
    let fileURL =
FileManager.default.temporaryDirectory.appendingPathComponent(filename
)
    return try? Data(contentsOf: fileURL)
}
```

✅ Q18: How do you handle pagination in SwiftUI when fetching data from a server?

1. Update the API Service to Support Pagination

```
struct UserService {
```

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

```
    static func getUsers(page: Int) async throws -> [User] {
        let url = URL(string:
"https://api.example.com/users?page=\(page)")!
        return try await APIClient.shared.fetchData(from: url)
    }
}
```


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

2. Modify the ViewModel to Handle More Pages

```
@MainActor
class UserViewModel: ObservableObject {
    @Published var users: [User] = []
    @Published var currentPage = 1
    private var isLoading = false

    func loadMoreUsers() async {
        guard !isLoading else { return }
        isLoading = true
        do {
            let newUsers = try await UserService.getUsers(page:
currentPage)
            users.append(contentsOf: newUsers)
            currentPage += 1
        } catch {
            print("Error: \(error.localizedDescription)")
        }
        isLoading = false
    }
}
```

3. Load More Data When Scrolling to Bottom

```
struct UserListView: View {
    @StateObject private var viewModel = UserViewModel()

    var body: some View {
        List {
            ForEach(viewModel.users) { user in
                Text(user.name)
            }
            if !viewModel.users.isEmpty {
                ProgressView()
                    .task {
                        await viewModel.loadMoreUsers()
                    }
```


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

```
            }
        }
        .task {
            await viewModel.loadMoreUsers()
        }
    }
}
```

✅ Q19: How do you handle API failures and display proper error messages in SwiftUI?

1. Define API Errors

```
enum APIError: LocalizedError {
    case badURL, decodingError, serverError(Int)

    var errorDescription: String? {
        switch self {
        case .badURL: return "Invalid URL"
        case .decodingError: return "Failed to decode response"
        case .serverError(let code): return "Server Error: \(code)"
        }
    }
}
```

2. Handle Errors in API Client

```
func fetchData<T: Decodable>(from url: URL) async throws -> T {
    let (data, response) = try await URLSession.shared.data(from: url)

    guard let httpResponse = response as? HTTPURLResponse,
httpResponse.statusCode == 200 else {
        throw APIError.serverError((response as?
HTTPURLResponse)?.statusCode ?? 500)
    }

    do {
```

```
            return try JSONDecoder().decode(T.self, from: data)
        } catch {
            throw APIError.decodingError
        }
    }
```

3. Display Errors in SwiftUI View

```
@MainActor
class UserViewModel: ObservableObject {
    @Published var users: [User] = []
    @Published var errorMessage: String?

    func loadUsers() async {
        do {
            users = try await UserService.getUsers()
        } catch let error as APIError {
            errorMessage = error.localizedDescription
        } catch {
            errorMessage = "Unknown error occurred"
        }
    }
}
```

✅ Q20: How do you secure API keys and sensitive data in an iOS app?

1. Store API Keys in a `Config.xcconfig` File

Create a `.xcconfig` file:
ini

```
API_KEY = "your-secret-api-key"
```

- Access it in code:

  ```
  let apiKey = Bundle.main.object(forInfoDictionaryKey: "API_KEY")
  as? String
  ```

- **Use Secure Storage (Keychain)**

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

```
import Security

func saveToKeychain(key: String, value: String) {
    let data = Data(value.utf8)
    let query: [String: Any] = [
        kSecClass as String: kSecClassGenericPassword,
        kSecAttrAccount as String: key,
        kSecValueData as String: data
    ]
    SecItemAdd(query as CFDictionary, nil)
}

func readFromKeychain(key: String) -> String? {
    let query: [String: Any] = [
        kSecClass as String: kSecClassGenericPassword,
        kSecAttrAccount as String: key,
        kSecReturnData as String: true,
        kSecMatchLimit as String: kSecMatchLimitOne
    ]
    var result: AnyObject?
    SecItemCopyMatching(query as CFDictionary, &result)
    return (result as? Data).flatMap { String(data: $0, encoding:
.utf8) }
}
```

✅ Q21: How do you use `UIViewControllerRepresentable` and `UIViewRepresentable` in SwiftUI?

SwiftUI provides `UIViewControllerRepresentable` and `UIViewRepresentable` to integrate UIKit components into SwiftUI.

1. Using `UIViewControllerRepresentable` for a UIKit ViewController

Example: Embedding `UIImagePickerController` for image selection.

```
import SwiftUI
import UIKit
```

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

```
struct ImagePicker: UIViewControllerRepresentable {
    @Binding var selectedImage: UIImage?
    @Environment(\.presentationMode) private var presentationMode

    class Coordinator: NSObject, UINavigationControllerDelegate,
UIImagePickerControllerDelegate {
        var parent: ImagePicker

        init(_ parent: ImagePicker) {
            self.parent = parent
        }

        func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey:
Any]) {
            if let image = info[.originalImage] as? UIImage {
                parent.selectedImage = image
            }
            parent.presentationMode.wrappedValue.dismiss()
        }
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }

    func makeUIViewController(context: Context) ->
UIImagePickerController {
        let picker = UIImagePickerController()
        picker.delegate = context.coordinator
        return picker
    }

    func updateUIViewController(_ uiViewController:
UIImagePickerController, context: Context) {}
}
```

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

Usage in SwiftUI:

```
struct ContentView: View {
    @State private var selectedImage: UIImage?

    var body: some View {
        VStack {
            if let image = selectedImage {
                Image(uiImage: image).resizable().scaledToFit()
            }
            Button("Select Image") {
                isShowingImagePicker = true
            }
        }
        .sheet(isPresented: $isShowingImagePicker) {
            ImagePicker(selectedImage: $selectedImage)
        }
    }
}
```

2. Using `UIViewRepresentable` for a UIKit View

Example: Embedding `UIActivityIndicatorView` (Spinner) in SwiftUI.

```
import SwiftUI
import UIKit

struct ActivityIndicator: UIViewRepresentable {
    func makeUIView(context: Context) -> UIActivityIndicatorView {
        let spinner = UIActivityIndicatorView(style: .large)
        spinner.startAnimating()
        return spinner
    }

    func updateUIView(_ uiView: UIActivityIndicatorView, context:
Context) {}
}
```

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

Usage in SwiftUI:

```
struct ContentView: View {
    var body: some View {
        ActivityIndicator()
    }
}
```

✅ Q22: How do you pass data between UIKit and SwiftUI components?

There are multiple ways to pass data between SwiftUI and UIKit:

1. Using `@Binding` to Update Data from UIKit to SwiftUI

- Example: A UIKit-based `UISlider` updates a SwiftUI `@Binding` value.

```
import SwiftUI
import UIKit

struct SliderView: UIViewRepresentable {
    @Binding var value: Float

    class Coordinator: NSObject {
        var parent: SliderView
        init(_ parent: SliderView) { self.parent = parent }

        @objc func valueChanged(_ sender: UISlider) {
            parent.value = sender.value
        }
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }

    func makeUIView(context: Context) -> UISlider {
        let slider = UISlider()
```


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

```
        slider.addTarget(context.coordinator, action:
#selector(Coordinator.valueChanged(_:)), for: .valueChanged)
        return slider
    }

    func updateUIView(_ uiView: UISlider, context: Context) {
        uiView.value = value
    }
}
```

Usage in SwiftUI:

```
struct ContentView: View {
    @State private var sliderValue: Float = 0.5

    var body: some View {
        VStack {
            SliderView(value: $sliderValue)
            Text("Value: \(sliderValue)")
        }
    }
}
```

2. Using Delegate Pattern for Communication

UIKit ViewController can send data to SwiftUI via a delegate.

✅ Q23: When should you use UIKit in a SwiftUI-based project?

SwiftUI is powerful, but UIKit is still needed in some scenarios:

1. **Missing Features in SwiftUI:**
    - `UICollectionView` (for advanced collection layouts).
    - `UIScrollView` (for complex nested scrolling).
2. **Third-Party SDKs:**
    - Some libraries (like FirebaseAuth, MapKit) provide UIKit-based APIs.
3. **Advanced Animations & Gestures:**

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

    - UIKit's `CAAnimation` offers more flexibility.
4. **Custom UI & Interactions:**
    - Some controls (like `UITextView` with rich text editing) are easier in UIKit.
5. **Integration with Existing UIKit Codebases:**
    - Gradual migration to SwiftUI in legacy projects.

✅ Q24: How would you migrate an existing UIKit project to SwiftUI?

Migrating a UIKit project to SwiftUI can be done in steps:

1. Start with New Screens in SwiftUI

- Instead of rewriting everything, implement new screens using SwiftUI.

2. Use `UIHostingController` to Embed SwiftUI in UIKit

- Example:

```
import SwiftUI

class HostingVC: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let swiftUIView = UIHostingController(rootView: ContentView())
        addChild(swiftUIView)
        swiftUIView.view.frame = view.bounds
        view.addSubview(swiftUIView.view)
        swiftUIView.didMove(toParent: self)
    }
}
```

3. Use `UIViewControllerRepresentable` to Embed UIKit in SwiftUI

- Example: If you already have a UIKit-based Camera module, wrap it with `UIViewControllerRepresentable`.

4. Replace ViewControllers with SwiftUI Views

- Convert each screen one by one while keeping UIKit navigation if needed.

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

5. Handle Navigation & State Management

- If using `UINavigationController`, migrate to SwiftUI's `NavigationStack`.

6. Test & Optimize Performance

- Use Instruments to measure SwiftUI's performance.

✅ Q25: Can you embed a SwiftUI view inside a UIKit-based application?

Yes! Use `UIHostingController` to embed SwiftUI views inside UIKit.

1. Create a SwiftUI View

```
import SwiftUI

struct SwiftUIView: View {
    var body: some View {
        Text("Hello from SwiftUI!").font(.largeTitle)
    }
}
```

2. Embed It in a `UIViewController`

```
import UIKit
import SwiftUI

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let swiftUIView = UIHostingController(rootView: SwiftUIView())
        addChild(swiftUIView)
        swiftUIView.view.frame = view.bounds
        view.addSubview(swiftUIView.view)
        swiftUIView.didMove(toParent: self)
    }
}
```

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

3. Push SwiftUI View in Navigation

```
let swiftUIView = UIHostingController(rootView: SwiftUIView())
navigationController?.pushViewController(swiftUIView, animated: true)
```

✅ Q26: How does the SwiftUI App lifecycle (`@main`, `Scene`, `WindowGroup`) differ from UIKit?

SwiftUI introduced a **declarative app lifecycle** that differs from the traditional UIKit lifecycle:

Feature	SwiftUI (`@main`, `Scene`, `WindowGroup`)	UIKit (`UIApplicationDelegate`, `UIWindow`)
Entry Point	`@main` struct	`UIApplicationDelegate`
App Lifecycle	Uses `Scene` and `WindowGroup`	Uses `AppDelegate` and `SceneDelegate`
Multiple Windows	`WindowGroup` manages windows automatically	`UIWindow` and `UISceneDelegate` handle multiple windows manually
State Management	Uses `@State`, `@Environment`, `@SceneStorage`	Uses `AppDelegate` methods (`applicationDidBecomeActive`)
Navigation	`NavigationStack` in SwiftUI	`UINavigationController` in UIKit

Example of SwiftUI App Lifecycle

```
import SwiftUI

@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
```

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

```
    }
}
```

This eliminates the need for `AppDelegate.swift` and `SceneDelegate.swift`, simplifying app initialization.

✅ Q27: How do you persist user preferences and app state in SwiftUI?

SwiftUI provides multiple ways to persist data:

1. **`@AppStorage` (for simple key-value storage)**
    - Uses `UserDefaults` under the hood.
    - Best for user settings and small preferences.

```
struct SettingsView: View {

    @AppStorage("username") var username: String = "Guest"

    var body: some View {
        TextField("Enter your name", text: $username)
    }
}
```

2. **`@SceneStorage` (for temporary UI state)**
    - Stores values per scene (like text field contents).

```
struct NotesView: View {
    @SceneStorage("draftText") var draftText: String = ""

    var body: some View {
        TextEditor(text: $draftText)
    }
}
```

3. **Core Data (for structured storage)**
    - Best for relational data storage.

```
@main
```


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

```
struct MyApp: App {
    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            ContentView().environment(\.managedObjectContext,
persistenceController.container.viewContext)
        }
    }
}
```

4. **CloudKit (for syncing data across devices)**
    - Ideal for syncing user data.

```
import CloudKit

let container = CKContainer.default()
```

✅ Q28: When would you use `@AppStorage`, `UserDefaults`, `Core Data`, or `CloudKit` for state persistence?

Storage Method	When to Use
`@AppStorage` (`UserDefaults`)	Small user preferences like theme, username, toggle settings.
`UserDefaults` (without `@AppStorage`)	When you need more control over data types or syncing.
`Core Data`	For complex data models, relationships, and large datasets.
`CloudKit`	When you need to sync user data across devices securely.

✅ Q29: How do you handle deep linking and push notifications in a SwiftUI app?

1. Handling Deep Linking

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

SwiftUI apps handle deep links using `onOpenURL`:

```
struct ContentView: View {
    var body: some View {
        Text("Welcome to SwiftUI")
            .onOpenURL { url in
                handleDeepLink(url)
            }
    }

    func handleDeepLink(_ url: URL) {
        if url.scheme == "myapp" && url.host == "profile" {
            print("Open profile screen")
        }
    }
}
```

✅ **Configure deep links in `Info.plist`:**

xml

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>myapp</string>
        </array>
    </dict>
</array>
```

2. Handling Push Notifications

- **Register for notifications in SwiftUI's `App` struct:**

```
import UserNotifications

@main
```

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

```
struct MyApp: App {
    init() {
        requestPushNotificationPermissions()
    }

    func requestPushNotificationPermissions() {

UNUserNotificationCenter.current().requestAuthorization(options:
[.alert, .badge, .sound]) { granted, error in
            if granted {
                DispatchQueue.main.async {

UIApplication.shared.registerForRemoteNotifications()
                }
            }
        }
    }

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

- **Handle push notification payloads in `AppDelegate`:**

```
import UserNotifications
import UIKit

class AppDelegate: NSObject, UIApplicationDelegate,
UNUserNotificationCenterDelegate {
    func application(_ application: UIApplication,
didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
        print("Registered for Push Notifications")
    }
```

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

```
    func userNotificationCenter(_ center: UNUserNotificationCenter,
didReceive response: UNNotificationResponse, withCompletionHandler
completionHandler: @escaping () -> Void) {
        let userInfo = response.notification.request.content.userInfo
        print("Push Notification Data: \(userInfo)")
        completionHandler()
    }
}
```

- Assign `AppDelegate` in `@UIApplicationDelegateAdaptor`:

```
@main
struct MyApp: App {
    @UIApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

✅ Q30: How do you manage memory efficiently in a large SwiftUI application?

1. **Use `@StateObject` for Long-Lived Instances**
    - Avoid using `@ObservedObject` when object creation should persist.

```
class DataModel: ObservableObject {

    @Published var items: [String] = []
}

struct ContentView: View {
    @StateObject private var model = DataModel()
```

```
}
```

2. **Lazy Loading for Large Lists**
    - Use `LazyVStack` instead of `VStack` to improve scrolling performance.

```
ScrollView {
    LazyVStack {
        ForEach(0..<1000) { i in
            Text("Item \(i)")
        }
    }
}
```

3. **Optimize Image Loading**
    - Use **async image loading** to prevent blocking the UI.

```
AsyncImage(url: URL(string: "https://example.com/image.jpg")) { image
in
    image.resizable().scaledToFit()
} placeholder: {
    ProgressView()
}
```

4. **Use `weak` References in Closures**
    - Avoid retain cycles in async operations.

```
class APIManager {
    func fetchData(completion: @escaping () -> Void) {
        DispatchQueue.global().async { [weak self] in
            guard self != nil else { return }
            completion()
        }
    }
}
```

5. **Defer Heavy Computation**
    - Use `task` to avoid blocking the main thread.

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

```
struct ContentView: View {
    @State private var data: [String] = []

    var body: some View {
        List(data, id: \.self) { Text($0) }
            .task {
                data = await fetchData()
            }
    }

    func fetchData() async -> [String] {
        await Task.sleep(2_000_000_000) // Simulating delay
        return ["Apple", "Banana", "Cherry"]
    }
}
```

✅ Q31: How do you debug performance issues in a SwiftUI application?

Debugging performance in SwiftUI requires identifying bottlenecks, unnecessary view updates, and slow computations. Here's how you can approach it:

1. **Use `print()` to Track View Updates**
    - Add `print()` inside the `body` property to detect unnecessary re-renders.

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

    var body: some View {
        print("ContentView body updated") // Check how often the view
updates
        return VStack {
            Text("Count: \(count)")
            Button("Increment") {
                count += 1
            }
```

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

```
        }
    }
}
```

2. **Use Instruments for CPU & Memory Profiling**
    - Open **Instruments** → Use **SwiftUI View Body** to track re-renders.
    - Use **Time Profiler** to detect slow operations.
3. **Minimize View Updates Using `@State`, `@Binding`, and `@ObservedObject` Correctly**
    - **Bad Practice** (causes unnecessary updates):

```
struct BadView: View {
    var text: String

    var body: some View {
        Text(text)  // Updates every time the parent updates
    }
}
```

4. **Good Practice** (prevents extra updates):

```
struct GoodView: View {
    let text: String  // Declared as `let` to avoid unnecessary
updates

    var body: some View {
        Text(text)
    }
}
```

5. **Optimize Lists with `LazyVStack` and `LazyHStack`**
    - Large lists should always use **lazy stacks** to avoid loading offscreen views.
6. **Use `.drawingGroup()` for Complex Graphics**
    - For views with intensive graphics (e.g., `Canvas` or `Shape`):

```
Circle()
    .fill(Color.blue)
    .frame(width: 100, height: 100)
```

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

```
    .drawingGroup() // Offloads rendering to GPU
```

✅ Q32: What tools do you use for profiling and debugging SwiftUI apps?

1. **Xcode Instruments**
    - **SwiftUI View Body**: Checks unnecessary re-renders.
    - **Time Profiler**: Identifies slow function calls.
    - **Memory Leaks**: Detects memory leaks in `@StateObject` and closures.
2. **View Debugger in Xcode**
    - **Enable Debug View Hierarchy** (Cmd + Shift + D) to inspect the view tree.
3. **Environment Overrides in Previews**
    - Simulate UI changes without running the full app.

```
ContentView()
    .previewDevice("iPhone 14 Pro")
    .environment(\.sizeCategory, .accessibilityExtraLarge)
```

4. **Swift Concurrency Checker**
    - Enables runtime warnings for concurrency issues:

```
@MainActor class ViewModel: ObservableObject {
    @Published var data = ""
}
```

✅ Q33: How do you write unit tests and UI tests for SwiftUI views?

1. Unit Testing SwiftUI View Models (`XCTest`)

- View models should be tested independently.

```
import XCTest
@testable import MyApp

class ViewModelTests: XCTestCase {
    func testIncrementCounter() {
        let viewModel = CounterViewModel()
        viewModel.increment()
```

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

```
        XCTAssertEqual(viewModel.count, 1)
    }
}
```

2. UI Testing SwiftUI Views (`XCUITest`)

- Interact with UI elements and verify behavior.

```
import XCTest

class MyAppUITests: XCTestCase {
    func testButtonTapIncrementsCounter() {
        let app = XCUIApplication()
        app.launch()

        let button = app.buttons["Increment"]
        button.tap()

        let label = app.staticTexts["Count: 1"]
        XCTAssertTrue(label.exists)
    }
}
```

3. Snapshot Testing (via Third-Party Libraries)

- **Use `swift-snapshot-testing` to compare view appearances.**

```
import SnapshotTesting

func testViewSnapshot() {
    let view = ContentView()
    assertSnapshot(matching: view, as: .image)
}
```

✅ Q34: How do you test asynchronous code in Swift?

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

1. **Using `XCTestExpectation`**
    - Useful when testing async network calls.

```
func testFetchData() {
    let expectation = XCTestExpectation(description: "Fetch data")

    fetchData { result in
        XCTAssertNotNil(result)
        expectation.fulfill()
    }

    wait(for: [expectation], timeout: 5.0)
}
```

2. **Using `async/await` in Tests**
    - Swift 5.5 allows cleaner async tests.

```
func testAsyncFetchData() async throws {
    let data = try await fetchData()
    XCTAssertFalse(data.isEmpty)
}
```

✅ Q35: What are `PreviewProviders`, and how do you use them for testing UI in SwiftUI?

`PreviewProvider` allows live previews of SwiftUI views inside Xcode.

1. Basic Example

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

2. Multiple Previews

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

- You can preview different styles or devices.

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .previewDevice("iPhone 14 Pro")

            ContentView()
                .previewDevice("iPad Air (5th generation)")
        }
    }
}
```

3. Using `PreviewLayout`

- Preview a specific frame size:

```
struct SmallView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
            .previewLayout(.sizeThatFits)
    }
}
```

4. Passing Mock Data

- Great for testing different states.

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(viewModel: ViewModel(data: "Mock Data"))
    }
}
```

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

✅ Q36: How do you handle code reviews and ensure SwiftUI best practices in a team?

1. **Establish Clear Code Review Guidelines**
    - Follow SwiftUI best practices (e.g., state management, avoiding excessive view updates).
    - Use consistent naming conventions and folder structures.
    - Ensure accessibility support (`.accessibilityLabel()`, `.accessibilityHint()`).

Use PR Templates for Reviews
Example template:
markdown

```
## PR Summary
- [ ] Implements new feature
- [ ] Fixes bug
- [ ] Includes tests
```

2. **Leverage SwiftLint for Style Enforcement**
    - Enforce formatting rules (`line_length`, `force_unwrapping`, etc.).

yaml

```
line_length: 120
force_unwrapping: warning
```

3. **Automate Code Reviews with Static Analysis**
    - Use **Danger** to comment on PRs with suggestions.
4. **Encourage Discussion and Knowledge Sharing**
    - Conduct team meetings on SwiftUI patterns and optimizations.
    - Write internal documentation on best practices.

✅ Q37: How do you structure a SwiftUI project for scalability in a large team?

1. **Modular Architecture**

Split the app into feature-based modules:
markdown

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

```
📂 Features/
    📂 Home/
        - HomeView.swift
        - HomeViewModel.swift
    📂 Profile/
        - ProfileView.swift
        - ProfileViewModel.swift
📂 Core/
    - Networking.swift
    - CacheManager.swift
```

    -

2. **Use MVVM for State Management**
    - Separate logic from views to keep code maintainable.

```
class HomeViewModel: ObservableObject {
    @Published var articles: [Article] = []
    func fetchArticles() { /* API Call */ }
}
```

3. **Component-Based UI**
    - Reusable UI components to avoid duplication.

```
struct CustomButton: View {
    var title: String
    var action: () -> Void

    var body: some View {
        Button(title, action: action)
            .padding()
            .background(Color.blue)
            .cornerRadius(8)
    }
}
```

4. **Use Dependency Injection**
    - Inject dependencies rather than hardcoding them.

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

```
struct ContentView: View {
    @StateObject var viewModel: HomeViewModel
}
```

5. **Define Clear Navigation Flow**
    - Use `NavigationStack` instead of deep `NavigationLinks`.

✅ Q38: What are the best strategies for handling version control conflicts in SwiftUI projects?

1. **Follow a Feature Branch Workflow**
    - Use `main` (stable) → `develop` → `feature/branch-name`.
2. **Use View Previews to Identify UI Issues Before Merging**
    - Validate layout changes without running the full app.
3. **Break Down Large PRs**
    - Keep pull requests under **500 lines** to avoid complex conflicts.
4. **Resolve Merge Conflicts Efficiently**
    - Prefer Git's rebase over merge to keep history clean.

sh

```
git checkout feature-branch
git fetch origin
git rebase origin/main
```

5. **Use `.gitattributes` for Xcode Projects**
    - Avoid unnecessary conflicts in `.pbxproj` files.

✅ Q39: How do you mentor junior iOS developers in adopting SwiftUI?

1. **Pair Programming & Code Reviews**
    - Work together on SwiftUI projects to demonstrate best practices.
2. **Hands-on Mini Projects**
    - Assign small UI tasks (e.g., build a login screen, create a reusable button).
3. **Encourage Reading Apple's SwiftUI Documentation**
    - Start with the **SwiftUI Essentials** guide.
4. **Teach Debugging Techniques**
    - Show how to use Instruments and `print()` to track state updates.

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

5. **Explain Common SwiftUI Mistakes**
    - Avoid overusing `@State` when `@Binding` is needed.
    - Optimize large lists with `LazyVStack`.

✅ Q40: Have you ever led a SwiftUI migration project? How did you handle challenges?

Yes! When leading a SwiftUI migration, I followed a structured approach:

1. **Audit the Existing UIKit Codebase**
    - Identify reusable components and areas that can be directly converted.
2. **Start with Low-Risk Screens**
    - Begin with settings/profile screens before complex views.
3. **Use `UIViewControllerRepresentable` for Gradual Migration**
    - Embed UIKit inside SwiftUI while transitioning.

```
struct OldUIKitView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> SomeUIKitController
{
        return SomeUIKitController()
    }

    func updateUIViewController(_ uiViewController:
SomeUIKitController, context: Context) {}
}
```

4. **Optimize Performance Issues**
    - Replaced `UITableView` with `List` for dynamic UI updates.
5. **Refactor Navigation and State Management**
    - Moved from `Coordinator` patterns to **MVVM** with `@StateObject`.
6. **Train the Team on SwiftUI Concepts**
    - Hosted workshops and live coding sessions.
7. **Monitor and Debug Issues Post-Migration**
    - Used **Instruments** to track SwiftUI view re-renders.

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