iOS Coding Best Practices (2025)

Introduction

iOS development in 2025 is dominated by Swift and the declarative SwiftUI framework for building UIs. Apple has steadily moved away from Objective-C, encouraging developers to embrace Swift-only code. Writing clean, safe, and maintainable Swift code is crucial – especially as apps grow and user expectations rise. In this guide, we'll detail modern best practices for iOS development – from recommended patterns and language idioms to common pitfalls to avoid – based on Apple's guidelines and expert community insights. We'll focus on general iOS coding practices (independent of any specific architecture) as well as SwiftUI usage. Code examples, Apple documentation references, and GitHub resources are included to illustrate both good and bad practices. The target audience is beginner to intermediate iOS developers looking to level up their coding approach.

(Note: Many of these practices assume you're using recent iOS SDKs (iOS 16/17+) and Swift 5.5+ or Swift 6, which introduce modern features like async/await and SwiftUI improvements.)

Swift Language Best Practices

Embrace Type Safety and Modern Swift Syntax: Swift is strongly typed, which helps catch errors early. Prefer *type inference* to needless verbosity – let the compiler infer types for constants and variables when it's obvious[1]. For example, write let message = "Hello" instead of explicitly specifying the type String. This makes code cleaner while still safe. Use Swift's *literals* and concise closure syntax for clarity. For instance, use trailing closures and shorthand argument names in closures for brevity and readability[2]. A simple map example:

```
let numbers = [1, 2, 3, 4]
let squared = numbers.map { $0 * $0 }  // concise closure syntax
```

This is preferred over a verbose closure with explicit types. Overall, aim for code that is clear and "Swifty" rather than overly verbose or overly terse – clarity at the call site is most important (a principle from Apple's API Design Guidelines[3]).

Effective Use of Optionals: Optionals are Swift's safeguard against null values, but they must be used correctly. Always safely unwrap optionals using if let or guard let to handle the nil case[4].

For example:

```
guard let url = URL(string: urlString) else {
   print("Invalid URL")
   return
}
// use `url` safely here
```

Avoid force-unwrapping (value!) unless you are absolutely certain a value isn't nil – force unwraps can cause runtime crashes if you're wrong[5]. In 2025, Swift has improved the developer experience with optionals (for instance, the ?? nil-coalescing operator and optional chaining), so use those to write safe, concise code. Bad Practice: doing something like let data = try! someOptional!.get() is a double-force-unwrap that will crash if someOptional is nil or an error is thrown – it's far better to handle the nil or error via guard or do/catch. In summary, check for nil and handle it gracefully; don't assume it away.

Value Types vs Reference Types: Prefer structs (value types) over classes for modeling data when reference semantics aren't needed. Structs are copied on assignment, which can prevent unintended side effects and makes reasoning about state easier. Apple encourages using structs for simple data models[6]. For instance, a User model with just data fields can be a struct. Use classes (reference types) only when you require shared mutable state or identity (e.g., an object that is passed around and mutated by various parts of the app). Value types also have performance benefits (stack allocation, no ARC overhead) and thread-safety advantages. A common pattern is to make your model types struct and only use classes for things like view controllers or other entities that truly need to be reference types. This aligns with Swift's design philosophy of safe defaults.

Protocol-Oriented Design: Swift is built with protocol-oriented programming in mind. Define protocols to represent abstractions or capabilities, and use protocol extensions to provide default implementations[7][8]. This promotes code reuse and flexibility. For example, if you have many types that should be able to greet(), define a Greetable protocol with a greet() requirement, and then provide a default implementation in an extension so types can adopt it easily[7][8]. Protocols enable a form of polymorphism without inheritance, which often leads to cleaner architectures than deep class hierarchies. Use them to decouple components. (But avoid overusing protocols for every little thing – use when it makes sense, such as defining an interface for a service, delegate, or common behavior across different types.)

Naming and Style Conventions: Follow Swift's standard naming conventions and style guidelines for consistency. This means camelCase for variable and function names, PascalCase (capitalized) for type and protocol names, and meaningful, descriptive names that make code self-documenting[9]. Boolean properties should read like assertions (e.g. isHidden, hasPrefix, canSend[10]). Function names and parameters should form a fluent phrase at call sites (for example, func addSubscriber(_ email: String, to mailingList: List) reads well). Apple's Swift API Design Guidelines stress that clarity is paramount – code is read far more often than written[3]. It's also good practice to include documentation comments for public APIs in your code. Apple encourages writing a brief doc comment for every declaration; if you can't explain what something does in simple terms, that's a sign the API might need rethinking[11]. Tools like SwiftLint can help enforce style rules (spacing, naming, etc.) consistently across your team.

Leverage Modern Swift Features: Swift is evolving rapidly; using its newer features can make your code safer and more efficient. A few recommendations in 2025:

- Use Codable for JSON and data serialization – it dramatically simplifies encoding/decoding of structs and classes to external representations[12]. You can often replace manual JSON parsing with a few lines of Decodable structs, which is less error-prone.
- Adopt Swift's concurrency (async/await) – introduced in Swift 5.5, async/await allows writing asynchronous code that looks like sequential code, improving readability and reducing the chance of callback-related bugs. For example, use let (data, _) = try await URLSession.shared.data(from: url) rather than completion handlers[13]. Async/await handles many threading details for you and works seamlessly with structured concurrency (Tasks and task groups). This leads to clearer code flow for network calls, database access, etc., and helps avoid pyramid-of-doom nested closures.
- Structured Concurrency and Actors: Building on async/await, Swift's concurrency model includes features like actors (for isolating mutable state across threads) and Sendable types (for compiler-checked thread safety). If your app involves multithreading or shared state, consider using actors to ensure only one task accesses mutable state at a time. This can prevent race conditions without needing explicit locks. Use @MainActor for any code that updates UI to guarantee it runs on the main thread (more on that in a moment).

Staying up-to-date with Swift's features (for example, Swift 6+ may introduce additional improvements or even macros to eliminate boilerplate) will help you write more robust and succinct code. Apple's own WWDC sessions often highlight new language capabilities that can be adopted for better performance or safety – e.g. WWDC 2025 introduced InlineArray and Span types in Swift 6.2 for low-level memory optimization[14], which advanced developers can use in performance-critical code. While beginners need not dive into such advanced types immediately, being aware of the direction Swift is moving helps inform good practices.

General iOS Development Best Practices

Choose a Sensible Architecture & Keep Components Focused: iOS doesn't mandate a single app architecture, but a clear separation of concerns is key to avoid the infamous "Massive View Controller" problem. In Apple's classic MVC, view controller classes often ended up doing too much (network calls, data formatting, navigation, etc.)[15]. To combat this, prefer an architecture that divides responsibilities cleanly. Many teams use MVVM (Model-View-ViewModel) in 2025, especially with SwiftUI, to move UI state and logic out of the view controllers/views and into view model objects[16][17]. This keeps UI code simpler and improves testability (since ViewModels can be tested without UI). Others in the community use VIPER or The Composable Architecture, but as a beginner, MVVM is a gentle step up from MVC that aligns well with SwiftUI's design. The main point: whichever pattern you use, ensure your classes/structs each have a single clear purpose (following the Single Responsibility Principle from SOLID)[18]. For example, networking code should live in a networking or data layer, not scattered in view controllers. UI layout code goes in views or view controllers, while business/domain logic goes in models or managers. Keeping concerns separated makes your code easier to maintain and scale.

Dependency Injection over Singletons: A common anti-pattern is overusing singletons or global state to pass data around (e.g., a singleton `DataManager` accessed everywhere). This can lead to tight coupling and difficulties in testing. Instead, use dependency injection (DI) – pass required dependencies into an object, preferably via initializers or at least setters[19][20]. For instance, if a view controller needs a view model or a network service, inject it during initialization rather than having the view controller reach out to a global. Example in Swift: `let profileVC = ProfileViewController(viewModel: profileVM)`. This makes dependencies explicit and allows you to substitute mocks in tests. Apple's frameworks themselves use DI in many places (e.g., you supply a data source object to a table view rather than the table view fetching data globally). Use singletons only for truly global, app-wide state that inherently has a single instance (e.g., an `Analytics` tracker, if needed). Even then, consider if a static dependency (like Swift's

UserDefaults.standard) is more appropriate. The Swift Forums and documentation note that delegates are typically weak to avoid retain cycles[21], which is a form of DI (the delegate object is injected and stored weakly). Adopting DI will naturally decouple your code and reduce "spaghetti" global usage.

Handle App Data with Stores or Services: Alongside UI and model layers, a good practice is to have dedicated manager or service classes (sometimes called "stores" or "repositories") for things like network calls, database access, etc. For example, a UserService class could handle fetching and caching user profiles. Your views or view models then call into this service. This encapsulation means if you switch from a REST API to GraphQL or to a local database, much of the app doesn't need to change – only the service layer does. It also makes mocking data easier (you can provide a fake service in tests). The Futurice iOS Guide suggests an architecture where *Stores* vend model data and handle caching/networking, which keeps view controllers lighter[22][23]. In SwiftUI apps, you might use Observable Objects (view models) that internally use such services to load data asynchronously. The key is to avoid scattering data-fetching logic across view controllers or views – centralize it.

State Management in SwiftUI: If you're building UI with SwiftUI, managing state correctly is critical. SwiftUI is declarative and uses a data-driven approach: your UI reflects the state of your data. Use the appropriate property wrapper for each kind of state:

- @State for simple local view state (value types) that belongs to a single view.
- @StateObject for reference-type view models that the view owns (this ensures the object is created only once per view lifecycle)[24][25]. For example, a @StateObject var viewModel = ProfileViewModel() in a view's declaration.
- @ObservedObject for passing in an object from a parent or elsewhere – use this when the object's lifetime is owned by someone else (the parent view or an environment)[24]. Using @ObservedObject on a newly created object in a body is a common mistake – that can cause reinitialization on every view update. Instead, @StateObject is the correct choice in that case[26].

- @EnvironmentObject for objects placed into the environment (globally available to a subtree of views), often used for app-wide shared data like user settings or app model.

By choosing the right property wrapper, you avoid subtle bugs like a view model resetting unexpectedly or not updating the UI. An example of a bad practice is creating a view model with @ObservedObject inside a view's body – this will recreate the VM on each render, resetting state. The fix is to create it with @StateObject once[26]. Apple's docs confirm that @StateObject initializes the object only once and preserves it through the view's life (even if the view is recreated in the UI hierarchy)[27].

Additionally, keep SwiftUI views lightweight: UI should ideally be a function of state. Avoid performing heavy computations in the body of a view; if you must, do them in a background thread or in an onAppear or .task modifier, and then update state. SwiftUI will recompute view bodies whenever state changes, so heavy work there can slow your UI. Use *pure functions* or property getters for computed display data whenever possible, and leverage SwiftUI's identity (the Identifiable protocol and id in ForEach) to help it diff views efficiently.

SwiftUI View Optimization: To ensure your SwiftUI app remains responsive, follow a few best practices from experienced developers:

- Avoid unnecessary re-renders. SwiftUI's reactive nature means that changing a state variable will re-evaluate the entire body of a view. If your view is large, this could be expensive. Structure your views by breaking out subviews for portions that don't need to redraw every time. For example, if only a label's text changes, isolate complex subviews so they don't rebuild on every state change[28][29]. In code above[30][31], the "ExpensiveAnimationView" was decoupled from a changing counter by extracting it into a CounterDisplay subview. This way, incrementing the counter doesn't cause the animation view to reconstruct unnecessarily[32][33]. Use the Self._printChanges() debug tool to detect what triggers re-renders in development[34].

- Prefer Lazy containers for large lists. If you have a scrollable list of many items, use LazyVStack/LazyHStack or SwiftUI's List rather than rendering all items upfront. Lazy stacks only create views as they scroll into view, dramatically reducing memory and initial load time for long lists[35][36]. A test showed that using a LazyVStack cut memory usage by 60% for 500 items compared to a regular VStack that created all subviews at once[37]. This is a huge win for performance.
- Avoid AnyView unless absolutely necessary. Wrapping views in AnyView type-erases them but introduces a performance cost, and it can obscure SwiftUI's ability to differentiate view types. Instead of using AnyView in conditional branches, use conditional statements in view builders to return different view types[38][39]. E.g. write if condition { Text("Hello") } else { Image("icon") } directly, or wrap in a Group with the conditional, rather than AnyView(Text("Hello")). Using type-safe conditionals lets SwiftUI analyze the view tree at compile time, leading to better performance.
- Use the Environment and Preferences for Cross-Cutting Concerns: SwiftUI environment values (like color schemes, layout direction, etc.) and your own @EnvironmentObject can be used to avoid passing the same data down through many layers of views. But be cautious: overusing global environment objects can make the data flow less clear. A balance is needed – use them for truly global app state (e.g., user authentication status) but for most other data, prefer explicit view model passing or the new Observation framework (if using iOS 17+ which introduced the @Observable macro to integrate with SwiftUI).

UIKit Best Practices (if applicable): Many projects still use UIKit or mix it with SwiftUI via UIViewRepresentable. If you're using UIKit (storyboards or programmatic):

- Design your view hierarchy with Auto Layout constraints or SwiftUI, rather than absolute frames. This ensures your UI adapts to different screen sizes, orientations, and Dynamic Type sizes. Use size classes and traits to make responsive layouts[40][41]. Apple's Human Interface Guidelines (HIG) emphasize adaptable UI design, so follow those recommendations for a polished app.

- Always perform UI updates on the main thread. UIKit (and even SwiftUI under the hood) is not thread-safe for UI operations. A very common beginner mistake is updating UI in a background thread (e.g., inside a network callback without hopping to main thread), which can lead to glitches or no update at all. Make sure to dispatch UI work to the main queue[42][43]. In practice, if you're using async/await, annotate your UI-updating code with @MainActor or call await MainActor.run { ... } to ensure it runs on the main thread. For example:

```
// Using GCD explicitly (UIKit scenario)
DispatchQueue.main.async {
   self.tableView.reloadData()
}
```

Many modern APIs like URLSession calls their completion on a background thread, so you *must* hop to main for UI. (Some libraries do auto-handoff to main, but never assume – always check documentation or explicitly dispatch). With Swift concurrency, any code in an @MainActor context will automatically run on the main thread, which is a safer model.

- Use Weak Delegates and Avoid Retain Cycles: In UIKit patterns like delegation (e.g., UITableViewDelegate), always declare delegate properties as weak to prevent strong reference cycles[21]. For instance, if a view controller holds a reference to a child object and that child has a delegate back to the view controller, making that delegate weak breaks the cycle. The Swift documentation explicitly notes this as a best practice ("delegates are declared as weak references"[21]). Similarly, when using closures that capture self (e.g., an animation completion block or a Combine subscriber), consider capture lists like [weak self] to avoid memory leaks[44]. A retain cycle through a closure is common if an object retains a closure that retains the object. By capturing self weakly and optionally binding it inside the closure, you prevent the cycle and allow deallocation if needed[45][46].

- Accessibility & Localization: A good practice (often overlooked as “coding” but very important) is to make your app accessible and localizable. Use SwiftUI’s accessibility modifiers or UIKit’s accessibility properties to label UI elements for VoiceOver, support Dynamic Type (scaling fonts), and test using Accessibility Inspector. Following these guidelines not only broadens your audience but often leads to cleaner code (e.g., separating content from presentation). Likewise, design your strings to be localizable (using NSLocalizedString or SwiftUI’s localization support) rather than hardcoding text, which usually means your UI text is not baked into code logic.

Error Handling and Safety: Swift’s error handling with do/try/catch (and now async throws) should be used to handle failures explicitly. Best practice is not to ignore errors – handle them appropriately, whether by showing an error message to the user, retrying, or propagating the error up. Using Swift’s Result<Type, Error> type can be helpful for asynchronous APIs (though with async/await, returning a throws function often is cleaner). Decide on a consistent error handling strategy in your app: for example, for network requests, you might have a centralized error handler that inspects the error and decides if it’s user-facing (e.g., show “No internet connection” alert) or a silent failure. Bad Practice to avoid: catching an error and then doing nothing (e.g., an empty catch block) – this can make debugging nightmares. Even if you just log the error, it’s better than suppressing it. Also avoid using exceptions for flow control – stick to Swift’s native error handling or optionals for expected fail cases.

Testing and Continuous Improvement: Incorporate testing into your development process. Apple has made testing more powerful with the introduction of the Swift Testing framework (a new open-source package as of WWDC 2024) which augments XCTest[47][48]. You can now write test functions with a simple @Test attribute and use macros like #expect for assertions[49][50]. This modern testing API embraces Swift features (like concurrency support, async tests, and even checkable macros) to make tests more expressive. Whether you use Swift Testing or the traditional XCTest, the best practice is to write unit tests for your core logic: data parsing, business rules, view model functions, etc. For UI, you can use Xcode's UI Test recorder or write SwiftUI view tests (there are ways to render a view and assert on its output state). Testing not only catches bugs early but also encourages better code structure (if a class is hard to test, it might be doing too much). Set up Continuous Integration (with Xcode Cloud, GitHub Actions, etc.) to run tests on each commit – this helps catch regressions. Testing in SwiftUI can also involve using Previews as a kind of visual test – if a view's preview crashes or doesn't display as expected, that's immediate feedback to fix an issue. Aim for at least basic coverage of critical functionality.

Another aspect of quality is using tools like Instruments (part of Xcode) to profile memory and performance. Apple provides Instruments for memory leaks (Leaks instrument), CPU usage, and more. It's a best practice to test your app with Instruments, especially if you notice slowness or memory growth. For example, if you suspect a retain cycle leak, run the Leaks tool or memory graph debugger. If a certain screen is slow to appear, profile it with Time Profiler to see what functions are hot. Apple's WWDC talks (e.g. *"Improve memory usage and performance with Swift"*) demonstrate how choosing the right algorithms and data structures can significantly speed up code[51][52]. In Swift, sometimes seemingly harmless choices (like using an array vs a Set, or too much copy-on-write overhead) can impact performance – Instruments will guide you to those bottlenecks. Adopt a mindset of measurement: don't guess at performance, use the tools to find out for sure.

Build Configuration and Package Management: By 2025, the default way to manage external libraries is the Swift Package Manager (SPM). Prefer SPM for adding dependencies over older tools like CocoaPods or Carthage, since SPM is integrated into Xcode and supported by Apple. It keeps your project cleaner (no workspace juggling) and simplifies CI builds. For code organization within your app, you can even use SPM to modularize features of your app into local packages – this enforces separation of concerns and can improve build times by limiting what needs recompilation after changes. Speaking of build times, Apple's documentation notes that some coding practices can slow compile times. For instance, large Swift files or complex expressions can strain the compiler. A few tips: break up very large files (so parallel build tasks can operate), mark classes final where possible (it helps the compiler optimize and reduces the size of the generated binary), and avoid overly generic code or deep type inference that confuses the compiler. Apple specifically advises reducing the number of symbols your code exports and giving the compiler explicit information to improve build efficiency[53]. In practice, this means limiting public or open declarations to only what's necessary (extra public symbols increase the dynamic linker's work and compile times)[54][55], and using fileprivate/internal for most things. It also means preferring straightforward code over clever, extremely generic implementations when you don't truly need the generality – sometimes writing two simple functions is better than one complicated generic function that's harder to compile. Keep an eye on Xcode's build timing summary to identify slow compilation files and refactor if needed. For instance, heavy use of protocol extensions with many conditional conformances could slow builds – weigh the trade-offs.

Common Pitfalls and Bad Practices to Avoid

Even knowing the best practices, it's easy to fall into some common traps. Here's a checklist of what not to do in iOS/Swift development, and how to avoid these issues:

- Forcing Optionals Unsafely: Avoid the use of ! to force-unwrap optionals unless you're 100% sure (by logic or documentation) that the value cannot be nil at runtime. A forced optional that unexpectedly is nil will trigger a crash (Fatal error: Unexpectedly found nil). Instead, handle nil cases with optional binding (if let/guard let)[4][5] or provide a default with ??. Bad: let text: String = textField.text! – this will crash if text is nil. Good: guard let text = textField.text else { return /* or show error */ }. A related bad practice is force-casting types (using as!); use optional cast (as?) and gracefully handle failure.

- Massive View Controller (MVC abuse): Don’t overload your view controllers with too many responsibilities (network calls, data parsing, etc.). This “Massive VC” anti-pattern makes code hard to maintain. If you find a view controller swelled to hundreds or thousands of lines, consider refactoring: move data logic to model objects or services, and move complex UI logic to custom views or view models. Apple’s MVC is a UI pattern, not an excuse to put everything in the controller. The existence of “Massive View Controller” was a prime motivation for patterns like MVVM[56][57] – in MVVM, for example, much of the view state and logic goes into the ViewModel, leaving the ViewController (or SwiftUI View) relatively slim[56][57]. Tip: If you have code in a view controller that doesn’t touch the UI at all (e.g., computing business data or contacting a server), that code likely belongs elsewhere (model or manager). By keeping controllers lean, you also make them easier to test and less error-prone. A massive view controller is also a red flag for low cohesion – break it into multiple controllers or objects if it manages multiple distinct pieces of functionality or UI.

- Excessive Use of Singletons/Globals: Using singletons for everything (e.g., a singleton for API calls, a singleton for user data, etc.) can create implicit dependencies and ordering issues. It also hinders testability, as singletons carry state between tests. Limit singletons to truly global concepts (and even then, consider if a dependency injection approach is cleaner). If you have to use them, reset their state between uses (especially in tests). Alternative: Use dependency injection (passing instances around), or SwiftUI’s Environment for globally needed objects. Another bad practice is using global variables or constants as a makeshift state storage – this is not thread-safe and makes reasoning about state difficult. Encapsulate state in classes/structs instead, and pass references where needed.

- Neglecting Threading Rules: As mentioned, *always do UI work on the main thread*. Another aspect is not handling background work properly – doing heavy tasks on the main thread will freeze the UI (and can even cause the watchdog to kill your app if it hangs too long). Common mistakes include parsing large JSON on the main thread or heavy image processing on main. Use background queues (DispatchQueue.global() or an OperationQueue) or Swift concurrency (Task { ... } which by default is on a background thread pool) for heavy lifting. Then switch to MainActor for updating UI. Also, avoid calling asynchronous APIs synchronously (like using semaphores to wait for a URLSession, which can deadlock) – embrace the asynchronous nature with async/await or completion blocks. *Misunderstanding concurrency* can lead to race conditions or crashes. If you are using locks or other primitives, be very careful to avoid deadlocks and always test on a device (emulators sometimes mask timing issues). Use higher-level concurrency abstractions (actors, Operation, DispatchQueues) which are safer. A pitfall is updating shared mutable data from multiple threads without synchronization – this can corrupt data. Either protect it with locks or, better, use actors or ensure all writes happen on a designated queue.

- Memory Leaks via Retain Cycles: A retain cycle occurs when object A retains B and B retains A (directly or indirectly), so neither is freed. In iOS, common cases are a closure capturing self strongly while self also retains that closure (e.g., UIView.animate closures, or Combine subscriptions stored in the object)[58][59]. To avoid this, use [weak self] in the capture list of closures that might cause cycles[46]. Similarly, as noted earlier, delegates should usually be weak. Symptom: If your view controllers or other objects aren't deinitializing when you expect, check for strong reference cycles. Use Xcode's memory graph debugger to spot strong reference cycles. In SwiftUI, you have to be mindful if you hold self in a long-running Task or ObservableObject – you might need [weak self] there too, or ensure to cancel tasks on deinit. Another bad practice is failing to invalidate timers or observers – e.g., if you add an entry to NotificationCenter and don't remove it (in iOS 9+ the center doesn't retain the observer if you use the block API, but with selectors it does). Always pair creation and teardown of such resources, typically in viewWillDisappear or deinit.

- Poor Error Handling (or None at All): As mentioned, swallowing errors is a mistake. For example, catching an error from a network call and then doing nothing (thus the app silently fails a task without the user knowing) provides a bad UX and complicates debugging. Always handle errors in some way: log them for debugging, and surface something to the user if it affects them (even if it's a generic "Oops, something went wrong" message). Also avoid overusing try! which will crash on error – only use it in scenarios where an error truly cannot happen or would be a programmer mistake (like try! regex(pattern:) when you're sure the pattern is valid). A better pattern is to use do/try/catch and maybe assert in the catch if it's an "impossible" error – that way you don't crash in release, but you'd catch it in testing.
- Hardcoding Values and Magic Numbers: Scattering hard-coded constants (like API URLs, layout dimensions, etc.) throughout code is a bad practice. It complicates changes and makes the code less readable. Use constants (e.g., let defaultTimeout = 30.0) or configuration files for such values. This also applies to strings – avoid user-facing strings inline in code; use localized string tables. It's not just about translation – it also centralizes copy changes. Magic numbers (like using 3.14159 in code instead of a named constant pi) reduce clarity. Always aim for self-explanatory code.
- Using Deprecated APIs or Outdated Techniques: Keep an eye on deprecation warnings. For example, using UIWebView or WKWebView? UIWebView was long deprecated – the modern practice is to use WKWebView exclusively. Similarly, certain older Core Data patterns (like manually calling save() on the main context on every minor change) have been supplanted by newer guidance (batch updates, background contexts, or by 2025, using SwiftData – a new persistence framework introduced in iOS 17 that sits on Core Data but is more Swifty). Make sure you're using updated frameworks: e.g., don't use DispatchQueue.global(qos:.background) for UI delays when you could use Task.sleep in Swift concurrency; don't use older C APIs when Swift provides safer

alternatives. Apple’s documentation and Xcode’s updated templates are good indicators of current best practices.

- Lack of Comments and Documentation: While code should be self-documenting as much as possible via clear naming, completely eschewing comments can be a pitfall too. Document why something is done if it’s not obvious. And use Swift’s markdown-friendly doc comments (///) for any public API of frameworks or complex pieces of logic. Future you (or teammates) will appreciate it. However, avoid obvious or redundant comments (don’t write // increment i on a line that says i += 1). Instead, focus on high-level comments for tricky logic or important rationale that isn’t immediately clear from code.

By being mindful of these pitfalls and adhering to the best practices above, you’ll write iOS apps that are not only functional but also maintainable, scalable, and robust. Remember that best practices evolve – what was optional or advanced a few years ago (like using async/await, or SwiftUI) is now the norm. Keep learning from Apple’s developer documentation and community blogs, as 2025 will surely bring new recommendations (for instance, Swift’s evolution may introduce new patterns like opaque result types or additional macro magic – which you should adopt if they make your code safer/cleaner).

Conclusion

Building a great iOS app involves more than just making it work – it's about writing code that future you (and others) can understand, extend, and optimize. In 2025, this means embracing Swift's powerful features and the SwiftUI framework, following proven coding conventions, and steering clear of common mistakes. We've covered a broad range of best practices: from fundamental Swift language idioms (optionals, structs, protocols, naming) to architectural and high-level considerations (MVVM, dependency injection, modularization), from UI-specific tips (efficient SwiftUI views, main-thread updates, accessibility) to performance and testing strategies (concurrency, Instruments, unit tests). Adopt these practices gradually in your projects – even small improvements in how you structure a view or handle state can lead to significantly more robust code. Always cite official Apple documentation and reputable community sources (like those we linked) for deeper dives into each topic, and keep an eye on the latest WWDC videos for the newest best practices each year. With a solid foundation in these modern iOS coding practices, you'll be well-equipped to write apps that are not only functional, but also elegant under the hood – a joy for both users and developers alike. Happy coding in Swift! 🚀

Sources:

Best practices and examples sourced from Apple Developer Documentation and iOS community experts: Apple's Swift API design guidelines[9][3], SwiftUI state management tips[24][60], iOS architecture guidance[61][62], thread and memory management advice[42][44], and more, as linked throughout this document. Each link provides additional context and sample code to deepen your understanding of the topic.

---

[1] [2] [4] [5] [6] [7] [8] [9] [10] [12] [13] Writing Swift Like Apple: Best Practices for Clean & Elegant Code | Swiftfy

https://medium.com/swiftfy/writing-swift-like-apple-best-practices-for-clean-and-elegant-code-943b9b1e6594

[3] [11] API Design Guidelines | Swift.org

https://www.swift.org/documentation/api-design-guidelines/

[14] [51] [52] Improve memory usage and performance with Swift - WWDC25 - Videos - Apple Developer

https://developer.apple.com/videos/play/wwdc2025/312/

[15] [16] [17] [18] [19] [56] [57] [61] Modern iOS Architecture Patterns & Best Practices for Scalable Apps | Swiftfy

https://medium.com/swiftfy/modern-ios-architecture-patterns-and-best-practices-e1ae397b0603

[20] [22] [23] [40] [41] [62] GitHub - futurice/ios-good-practices: Good ideas for iOS development, by Futurice developers.

https://github.com/futurice/ios-good-practices

[21] Should a delegate property passed into a struct also be declared as weak in the struct? - Using Swift - Swift Forums

https://forums.swift.org/t/should-a-delegate-property-passed-into-a-struct-also-be-declared-as-weak-in-the-struct/52766

[24] [25] [26] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [60] 5 SwiftUI Best Practices Every iOS Developer Should Master | by Apix | Medium

https://medium.com/@vrxrszsb/5-swiftui-best-practices-every-ios-developer-should-master-in-2024-4af4096a0856

[27] What is the difference between ObservedObject and StateObject in ...

https://stackoverflow.com/questions/62544115/what-is-the-difference-between-observedobject-and-stateobject-in-swiftui

[42] [43] The 10 Most Common Mistakes iOS Developers Don't Know They're Making | Toptal®

https://www.toptal.com/developers/ios/top-ios-development-mistakes

[44] [45] [46] [58] [59] Q&A: Is using [weak self] always required when working with closures? | Swift by Sundell

https://www.swiftbysundell.com/questions/is-weak-self-always-required/

[47] [48] [49] [50] Meet Swift Testing - WWDC24 - Videos - Apple Developer

https://developer.apple.com/videos/play/wwdc2024/10179/

[53] Improving build efficiency with good coding practices

https://developer.apple.com/documentation/xcode/improving-build-efficiency-with-good-coding-practices

[54] [55] Minimizing Your Exported Symbols

https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/ReducingExports.html