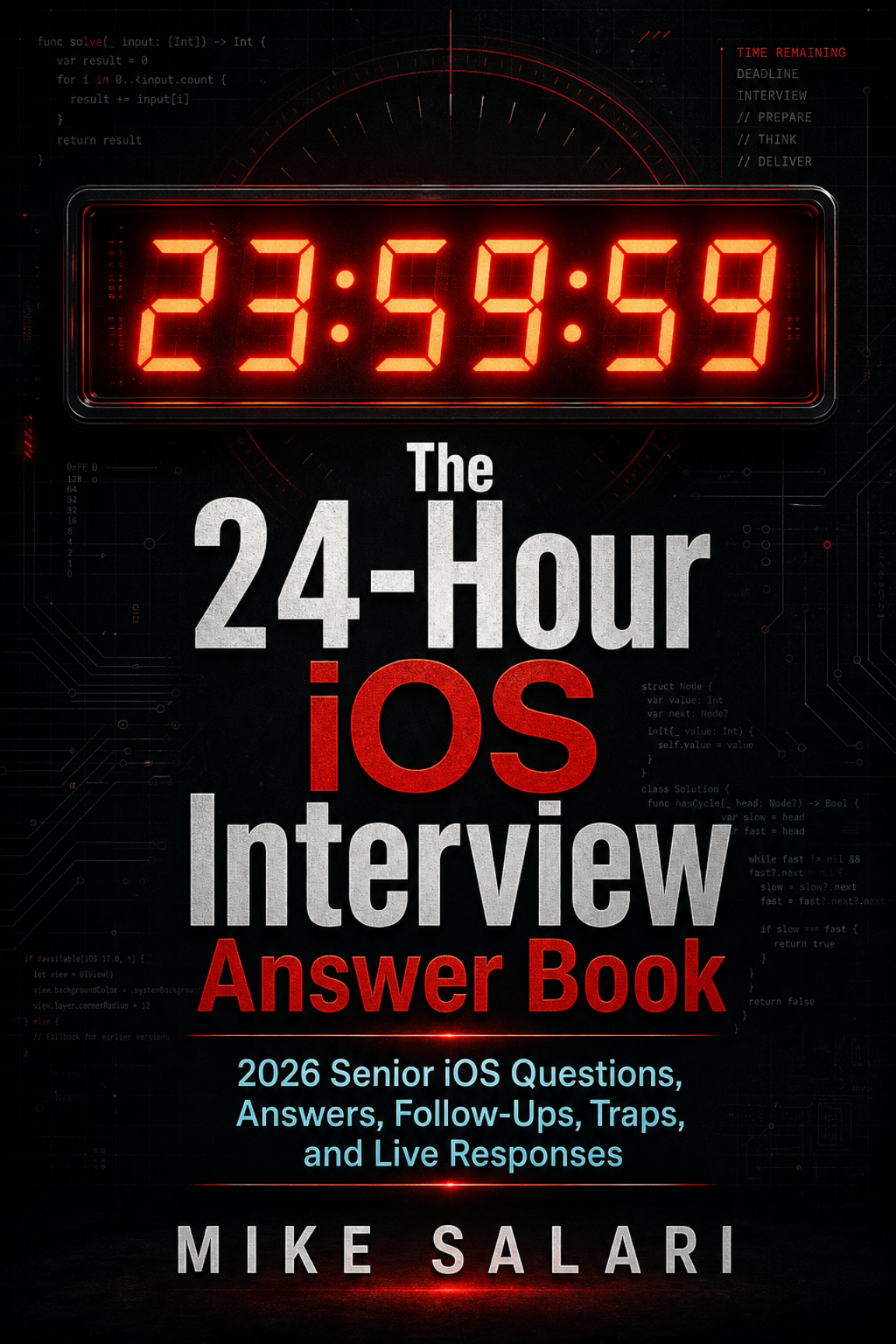

The 24-Hour iOS Interview Answer Book

2026 Senior iOS Questions, Answers, Follow-Ups, Traps, and Live Responses

MIKE SALARI

The 24-Hour iOS Interview Answer Book

Sample edition

Author: Mike Salari
Edition: 2026 Release Edition sample
Full book: 157 senior iOS interview answer cards, Top 50 live interview behavior, 5 simulations, final 30-minute review, and technical source appendix.

This sample is a preview of the full book's structure and voice. It is designed to show how the answer cards work: short answer, senior answer, trap, likely push, recovery line, and wording you can say out loud in a real interview.

About the author

Over more than fifteen years, I have built software at Apple, Adobe, Cisco, Visa, Mastercard, and startups. Most of that time, people depended on what I built without ever knowing my name.

I started badly: a cheap laptop, far too much confidence, and no idea what I was doing. That matters, because everyone who can do this started out unable to do this.

Since then, I have worked on systems where mistakes were expensive, reliability mattered, and small product decisions shaped how people experienced the work. A bad deploy was not a funny story. It was a call at three in the morning, a production incident, and people depending on us to understand the problem quickly.

I have shipped things I am proud of and things that still make me wince. I have hired engineers, mentored them, reviewed their code, watched the best ones grow past me, and sat in rooms where very smart people argued over one line of code for an hour, sometimes correctly.

I am not precious about tools. I have written low-level code and I have written prompts. I do not care how the work gets done. I care that it gets done well, that it does not break, and that the person who built it understands what they built.

That last part is the reason I wrote this book.

Website: salari.dev
Email: mike@salari.dev

What this sample includes

Inside this sample:

- A short preview of the 24-hour study method.
- A preview of what interviewers are actually measuring, and why strong engineers lose offers they were qualified for.
- 8 representative answer cards from Swift, concurrency, SwiftUI, system design, networking, testing, performance, and behavioral rounds, each with the weak answer, the interviewer push, and the recovery line.
- A short live interview simulation excerpt.
- A preview of what the full book contains.

This is not a beginner Swift tutorial. It assumes you can already build iOS apps and need interview-ready answers fast.

How to use this sample

Do not read passively. For each card:

1. Read the short answer.
2. Say the live interview wording out loud.
3. Read the trap and recovery line.
4. Answer the interviewer push without looking.

The goal is not to memorize paragraphs. The goal is to build answers you can say under pressure without sounding like a search result.

What interviewers are actually looking for (preview)

Most prep treats an interview like an exam: learn the facts, recite them, pass. Senior loops do not work that way. The interviewer is quietly answering one question for their team: “do I want this person in the room when something is on fire?” Every question is a proxy for that.

Almost no one fails a senior loop for knowing too little. They fail for behavioral reasons hiding inside technical answers:

- **Overclaiming.** You say something slightly too strong (“Swift 6 makes data races impossible”), they poke it, and now they are wondering what else you would say too confidently. One overclaim discounts everything after it.
- **Rambling.** You know the answer, so you keep talking. Length reads as uncertainty. Three clear sentences sound more senior than three minutes.
- **Freezing on the push.** The first answer was fine; the follow-up is the actual test. Senior loops are won on the second answer, not the first.
- **Solving the wrong problem.** Naming patterns before asking a single requirement is the clearest mid-level tell in a design round.

The fix runs through every card in this book: say the precise version, name the trade-off, and stay calm when they push. “I have not measured that, so I will not guess, here is how I would find out” is not a weak answer at senior level. It is one of the strongest things you can say.

The full book includes the complete chapter, plus “The 20 answers that win offers” and a set of real candidate mistakes that decided an offer in a single exchange.

Sample card 1: Struct or class, and how do you choose?

Must-know · Very common · Mid to senior · Technical screen

Short answer

Default to structs for value semantics and predictable copies. Reach for a class when you need shared identity, reference features like inheritance, or a `deinit` hook.

Memorize this

Default to structs. Value semantics are shallow, so a struct holding a reference type still shares it.

Senior answer

Value types make local reasoning easier because a copy cannot be mutated out from under you. Classes earn their place when identity matters, like a single source of truth many parts of the app observe, or when you need a deinitializer or Objective-C interop.

The nuance is important: value semantics only hold as deep as the stored properties. A struct that contains a reference type still shares that inner object. That is why “structs are thread-safe” is not a senior answer.

Common trap

Saying “structs are thread-safe.” They are not automatically thread-safe. They reduce accidental sharing only when the stored properties also preserve value semantics.

Interviewer push

“When does choosing a struct actually bite you?”

Recovery line

“When it holds a reference type. The value semantics only go one level deep, so a copied or captured struct still shares that inner

object. That is why I do not treat struct as automatically safe to share across threads."

Live interview wording

"I default to structs because copies are predictable and easy to reason about, and I move to a class when I need shared identity, inheritance, or a deinit. The one nuance I keep in mind is that a struct holding a reference type still shares that inner object, so value semantics are not automatically thread safety."

What earns the point

Knowing value semantics are shallow, and not claiming structs are thread-safe.

Sample card 2: Explain actor reentrancy.

Senior-depth · Very common · Senior · Concurrency deep dive

Short answer

Actors protect actor-isolated state during synchronous execution, but when an actor method hits `await`, the actor can run other work before the original method resumes. That is actor reentrancy.

Senior answer

The dangerous pattern is “check, await, then act.” For example, an actor checks that a balance is high enough, awaits a payment call, then deducts. During the await, another task can enter the actor and change the balance. When the original task resumes, the old assumption may be stale.

The fix is to avoid carrying fragile invariants across suspension points. Either re-check after the await, reserve the state before suspending, or restructure the method so mutation does not straddle the suspension.

Common trap

Saying “actors run one thing at a time, so this is safe.” That is only true until a suspension point.

Interviewer push

“So if the actor is isolated, why can the balance still be wrong?”

Recovery line

“Because isolation is not the same thing as a transaction across awaits. The actor is exclusive during a synchronous run, but after an await another task can run and change the state. I either re-check after the await or reserve the value before suspending.”

Live interview wording

“Actor reentrancy means an actor method can suspend at await, let another task run on the actor, then resume with assumptions that may be stale. I avoid check-then-act across awaits, or I re-check the invariant before mutating.”

What earns the point

Knowing the exact boundary of actor isolation: exclusive during a synchronous run, not across suspension.

Sample card 3: What did Swift 6 change for concurrency?

Must-know · Very common · Mid to senior · Technical screen

Short answer

Swift 6 makes strict concurrency checking much more serious. It catches many unsafe-sharing problems around actor isolation and `Sendable` at compile time, but it does not make all concurrency bugs disappear.

Senior answer

The senior answer is careful. Swift 6 helps the compiler reject many patterns that could cross isolation boundaries unsafely. It makes `Sendable`, actor isolation, main actor correctness, and captured mutable state harder to ignore.

But it is not a proof that your program is correct. Logical races, stale responses, actor reentrancy, cancellation bugs, priority inversions, deadlocks, and bad business logic are still your responsibility.

Common trap

Saying “Swift 6 guarantees data-race safety now.” That is too strong.

Interviewer push

“So data races are gone?”

Recovery line

“No. Swift 6 catches many unsafe-sharing problems at compile time, especially around isolation and Sendable, but it does not prove the algorithm correct. I still design for cancellation, stale results, reentrancy, and ordering.”

Live interview wording

“Swift 6 makes strict concurrency checking much stronger, so many isolation and Sendable problems are caught earlier. I would not say it guarantees correctness. Logical races, cancellation bugs, and actor reentrancy still need design and tests.”

What earns the point

Precision. Do not overclaim Swift 6 as magic.

Sample card 4: @StateObject vs @ObservedObject, and @State with @Observable?

Must-know · Very common · Mid to senior · SwiftUI round

Short answer

`@StateObject` creates and owns an `ObservableObject` once and survives parent re-renders. `@ObservedObject` observes an object owned elsewhere. With `@Observable`, the owning view usually uses `@State` to create and own the model.

Senior answer

The classic bug is creating an object you intend to own with `@ObservedObject` inline. The parent re-renders, the object is recreated, and the state resets. `@StateObject` exists to tie object creation to the view's lifetime.

In the `@Observable` world, the ownership rule stays the same even though the syntax changes. The view that creates and owns the model uses `@State`; a child that receives it can store it as a plain value.

Common trap

Choosing wrappers by habit instead of ownership. The right question is not "which wrapper is modern?" It is "who owns this state?"

Interviewer push

"Why did my view model reset when the parent changed?"

Recovery line

"Because it was probably created inline with ObservedObject, so the parent re-render recreated it. The owning view should use StateObject for ObservableObject, or State with an Observable model, so the object is created once for that view lifetime."

Live interview wording

“StateObject creates and owns the object once; ObservedObject only observes one owned elsewhere. With Observable, the same ownership rule applies: State owns the model, children receive it. I pick by ownership, not by habit.”

What earns the point

Diagnosing the ownership bug, not just naming property wrappers.

Sample card 5: Design an image caching and loading system.

Senior-depth · Very common · Senior · System design

Short answer

Use memory cache plus disk cache, request coalescing, cancellation, downsampling at decode time, and stable cache keys. Keep image decode and data prep off the main thread.

Senior answer

The core design is a pipeline:

1. Build a stable cache key from URL plus requested size or variant.
2. Check memory cache.
3. Check disk cache.
4. Coalesce in-flight requests so the same image is not downloaded twice.
5. Download, validate, downsample to display size, and store.
6. Cancel work when cells disappear or requests are no longer needed.

`NSCache` is useful for memory because it is pressure-aware, but it is not a strict LRU cache. Disk cache needs explicit eviction by size, age, or both. For scroll performance, the critical detail is downsampling before rendering, not decoding a giant image and scaling it in the cell.

Common trap

Only saying “I use URLCache” or “I put images in NSCache.” That misses downsampling, request coalescing, disk eviction, and cancellation.

Interviewer push

“Why does the list still jank if the image is cached?”

Recovery line

“Because cached bytes are not the same as a prepared image. If decode or resizing happens on the main thread during scrolling, the cell can still hitch. I downsample and prepare off-main, then update the UI on the main actor.”

Live interview wording

“I use memory plus disk cache, stable cache keys, request coalescing, cancellation on reuse, and downsampling at decode time. NSCache is pressure-aware, not strict LRU, so disk eviction is explicit. For smooth scrolling, decode and resize happen off-main.”

What earns the point

Separating bytes, decoded image, display size, and main-thread cost.

Sample card 6: How do you handle token refresh under concurrency?

Senior-depth · Very common · Senior · Networking round

Short answer

Use a single-flight refresh. If several requests get 401 at once, they all await the same refresh task. Replay each original request once, and never loop forever on repeated 401s.

Senior answer

The dangerous case is a refresh stampede: ten requests fail at once and all try to refresh the token. That creates races, invalid tokens, and sometimes account lockouts.

I keep token state behind an actor or other serialized boundary. When refresh starts, I store the in-flight task. Other callers await that same task instead of starting another refresh. After refresh succeeds, each original request retries once. If it fails again with 401, I stop and force re-auth rather than looping.

Common trap

Refreshing independently per request.

Interviewer push

“What happens when five requests all get 401 at the same time?”

Recovery line

“They share one refresh. The first request creates the refresh task; the others await it. After it succeeds, each request replays once. If replay still returns 401, I stop and send the user through login.”

Live interview wording

“Concurrent 401s share one refresh task, usually protected by an actor. I replay the original request once after refresh, and I never

loop on repeated 401s. If refresh fails, I clear auth state and require login."

What earns the point

Naming the stampede, the single-flight fix, replay-once behavior, and the no-infinite-loop rule.

Sample card 7: What is Swift Testing, and how does it compare to XCTest?

Must-know · Common · Mid to senior · Testing round

Short answer

Swift Testing is Apple's newer Swift-native testing framework with `@Test`, `#expect`, `#require`, async tests, parameterized tests, traits, and tags. XCTest still matters for existing suites, UI tests, and performance baselines.

Senior answer

The important answer is not "Swift Testing replaces XCTest." It is that Swift Testing improves expressiveness for Swift code and can coexist with XCTest. `#expect` records failures cleanly, `#require` unwraps or stops the test, and parameterized tests reduce repetition.

For an existing app, I would not migrate everything just to migrate. I would use Swift Testing for new unit-level tests where it improves clarity, keep XCTest where the project already depends on it, and preserve UI/performance coverage that XCTest handles well.

Common trap

Turning the answer into framework hype. Interviewers care about whether your tests are deterministic, meaningful, and maintainable.

Interviewer push

"Would you migrate an existing XCTest suite?"

Recovery line

"Not blindly. I would add Swift Testing where it improves new tests, especially parameterized or async unit tests, and keep

stable XCTest coverage. Migration is only worth it if it reduces maintenance or improves clarity."

Live interview wording

"Swift Testing is Swift-native and gives Test, expect, require, parameterization, traits, tags, and async support. I would use it for new unit tests where it improves clarity, but XCTest still matters for existing suites, UI tests, and performance baselines."

What earns the point

Showing testing judgment instead of chasing the newest framework.

Sample card 8: Tell me about a failure.

Must-know · Very common · Senior · Behavioral round

Short answer

Own the failure, explain the impact, show the fix, and name the system change that prevented it from recurring. Do not blame the team, the reviewer, or the process as if you were outside it.

Senior answer

A strong senior failure story has four beats:

1. What happened.
2. Why it mattered.
3. What you did immediately.
4. What changed afterward.

The best stories are specific. “We shipped a memory leak” is better than “communication could have been better.” A senior answer shows technical ownership and process maturity: you fixed the bug, added regression coverage, improved review checklists, added metrics, or changed rollout policy.

Common trap

Choosing a fake weakness or blaming others. “I care too much” is not an answer. “QA missed it” is worse.

Interviewer push

“What would you do differently now?”

Recovery line

“I would have treated the missing signal as part of the bug. The code issue mattered, but the real fix was adding the regression test and rollout metric so we could catch it before every user got the release.”

Live interview wording

“We shipped a memory leak from a closure retaining a view model. I owned the investigation, found the retain cycle, patched it, and added a regression test that asserts the screen deallocates. The bigger fix was adding memory checks to the release checklist and watching MetricKit for OOM signals during phased rollout.”

What earns the point

Owning the miss and fixing the system, not just the line of code.

Live interview simulation excerpt

Interviewer: Quick warm-up. When would you use a struct versus a class?

Candidate: Structs for value semantics, classes for reference. I default to structs, they are thread-safe and easier to reason about.

> "Thread-safe" is the overclaim. The interviewer's job now is to test whether the candidate actually believes it.

Interviewer: Thread-safe? So if two threads touch the same struct, I am fine?

Candidate: Let me be careful with that word. Value semantics reduce accidental sharing, but only as deep as the stored properties. If my struct holds a reference type, both copies still point at the same object, so it is not automatically thread-safe. I default to structs for predictable copies, and reach for a class when I need shared identity or a deinit.

> Clean recovery. The candidate heard the overclaim, paused, and walked it back to the precise version without getting rattled.

Interviewer: Can you give me a concrete example where that bites?

Candidate: Sure. Say I have a `struct Request` that holds a `Config` class. I capture the request in a closure expecting a snapshot, then mutate `request.config.timeout` later. The closure sees the new value because the struct copy shared the same `Config` object. Capturing the struct only copied one level deep.

How it scores: A senior screen is not about never stumbling. It is about recovering precisely when pushed.

What the full book adds

The full version includes:

- The full chapter on what interviewers are actually looking for: why strong engineers lose offers, why overclaiming kills trust, and what sounds senior versus mid-level.
- The 50 questions most likely to decide your loop.
- The 15 questions that separate senior from mid-level.
- The 10 answers you must not get wrong, and the 20 answers that win offers.
- 157 answer cards across Swift, ARC, concurrency, SwiftUI, architecture, system design, networking, persistence, testing, performance, security, accessibility, release, live coding, behavioral, and leadership.
- Full live interview behavior on all Top 50 cards: the weak answer, the interviewer push, the recovery line, and what earns the point.
- Company answers shown three ways, bad, good, and great, for Apple, startups, fintech, and big tech.
- Real candidate mistakes that decided an offer in a single exchange.
- 5 complete live interview simulations.
- A senior interview readiness scorecard, a final 30-minute review, and a morning-of checklist.
- A technical notes and sources appendix checked against official Apple, Swift.org, and Swift documentation.

If you only have one evening, the full book is designed to help you search, skim, rehearse, and walk into the interview with answers you can actually say out loud.

End of sample.