

# WISE ROUND 3: Technical / System Design Interview — Complete Guide

60 minutes | Two Engineers | Zoom

---

## INTELLIGENCE SUMMARY

### What the Candidate Pack Says

"The interviewers will provide you with details on a **hypothetically existing feature in a mobile app** — they'll ask you to identify how they might expect this to be built, then give details about **proposed improvements** to this feature and look for how you would approach building it."

They're looking at:

- How you **break it down** into discrete units of work
- What **libraries** you'd use
- What **questions** you'd ask about the underlying implementation
- What **concerns** you'd have for backend, product, and design

### What Reddit/Glassdoor/Blind Confirms

Source	What Was Asked
Reddit	"Pattern was <b>contention and scaling writes/reads</b> with a <b>financial twist</b> . They led most of it. I didn't justify my trade-offs."
Glassdoor Mar 2025	"Event-driven architecture and data buckets" using Excalidraw
Glassdoor Dec 2024	"Design a <b>payment system</b> " (candidate got offer)
Glassdoor Nov 2025	"Should the experience be immediate synchronous or async with notification?"
Glassdoor May 2025	"More <b>practical</b> questioning, not textbook"
Glassdoor Budapest May 2025	"Software design interview was pretty fun — decided my level"
Taro Singapore	"2 algo questions + background discussion in 90 min onsite"
Wise's own page	"For <b>iOS/Android</b> , candidates may be asked to complete a <b>take-home coding problem</b> prior to the interview. Be prepared to <b>walk us through your solution.</b> "
Wise EM Tips	"Architectural perspective, high-level design choices, documentation, best practice"

## The Reddit Candidate's Fatal Mistake

"I didn't justify my trade-offs and got confused"

**This is the #1 thing to avoid.** Every design decision needs a "because..."

---

## THE FORMAT (Mobile-Specific)

Unlike standard backend system design, Wise's technical interview for mobile roles focuses on:

1. **A mobile app feature scenario** — they describe an existing feature
2. **You explain how you'd expect it's built** (architecture, patterns, layers)
3. **They propose improvements/changes** — you adapt your design
4. **Deep dive into concerns** — backend interactions, caching, offline, edge cases

It's NOT "design Twitter" style. It's "here's our transfer feature, how would you architect this on mobile, and what would you change when we add X?"

---

## STEP-BY-STEP FRAMEWORK

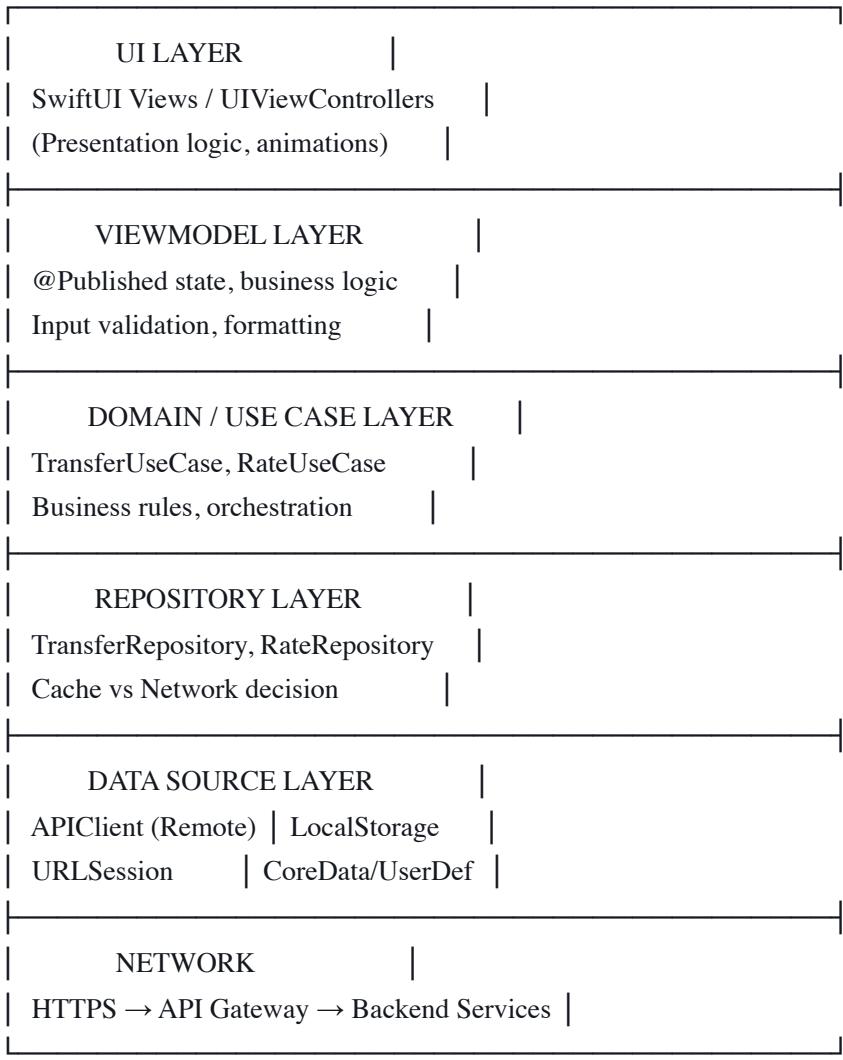
### Step 1: CLARIFY (2-3 min)

Ask these questions EVERY time:

- "How many users are we designing for? Thousands or millions?"
- "What platforms — iOS only, or cross-platform?"
- "What's the expected latency tolerance for the user?"
- "Is offline support required?"
- "Are there compliance/regulatory constraints?"
- "Is there an existing backend API, or am I designing that too?"

### Step 2: HIGH-LEVEL MOBILE ARCHITECTURE (5 min)

Draw the layers:



### Step 3: DEEP DIVE on the critical component (20-25 min)

This is where the interview lives. They'll push you on:

- Data flow (how does data get from API to screen?)
- State management (loading, error, success, empty)
- Caching strategy (when to cache, TTL, invalidation)
- Error handling (network failure, API errors, timeout)
- Concurrency (parallel requests, race conditions)
- **Trade-offs** (this is where most people fail)

### Step 4: PROPOSED IMPROVEMENTS (10-15 min)

They'll say "now what if we add feature X?" and watch you adapt.

- Don't panic — this is designed to test flexibility
- Show your architecture can handle changes without rewrite
- Discuss what you'd need from backend/product/design

## Step 5: TRADE-OFFS DISCUSSION (5 min)

The Reddit candidate failed here. Always use this template:

"I chose [A] over [B] because [reason]. The downside is [limitation], but given [Wise's context], that trade-off is acceptable. If requirements changed to [scenario], I'd reconsider."

## SCENARIO 1: International Money Transfer Feature (MOST LIKELY)

The prompt they might give:

"Wise has a transfer feature where users send money internationally. Walk me through how you'd expect the mobile architecture to work. Then we'll discuss improvements."

Your answer:

### Clarifying Questions

- "Is this a send-only flow, or also receive/request?"
- "Do we show a live exchange rate that updates in real-time?"
- "How long is a quoted rate guaranteed?"
- "Does the user need to add a recipient, or is that pre-saved?"
- "What payment methods — bank transfer, card, Apple Pay?"

### Transfer Flow (User Journey)

1. User enters amount → 2. App fetches rate + fee → 3. User sees quote  
→ 4. User confirms → 5. Transfer created → 6. Status tracking

### Mobile Architecture for Transfer

#### Screen: Transfer Amount Entry

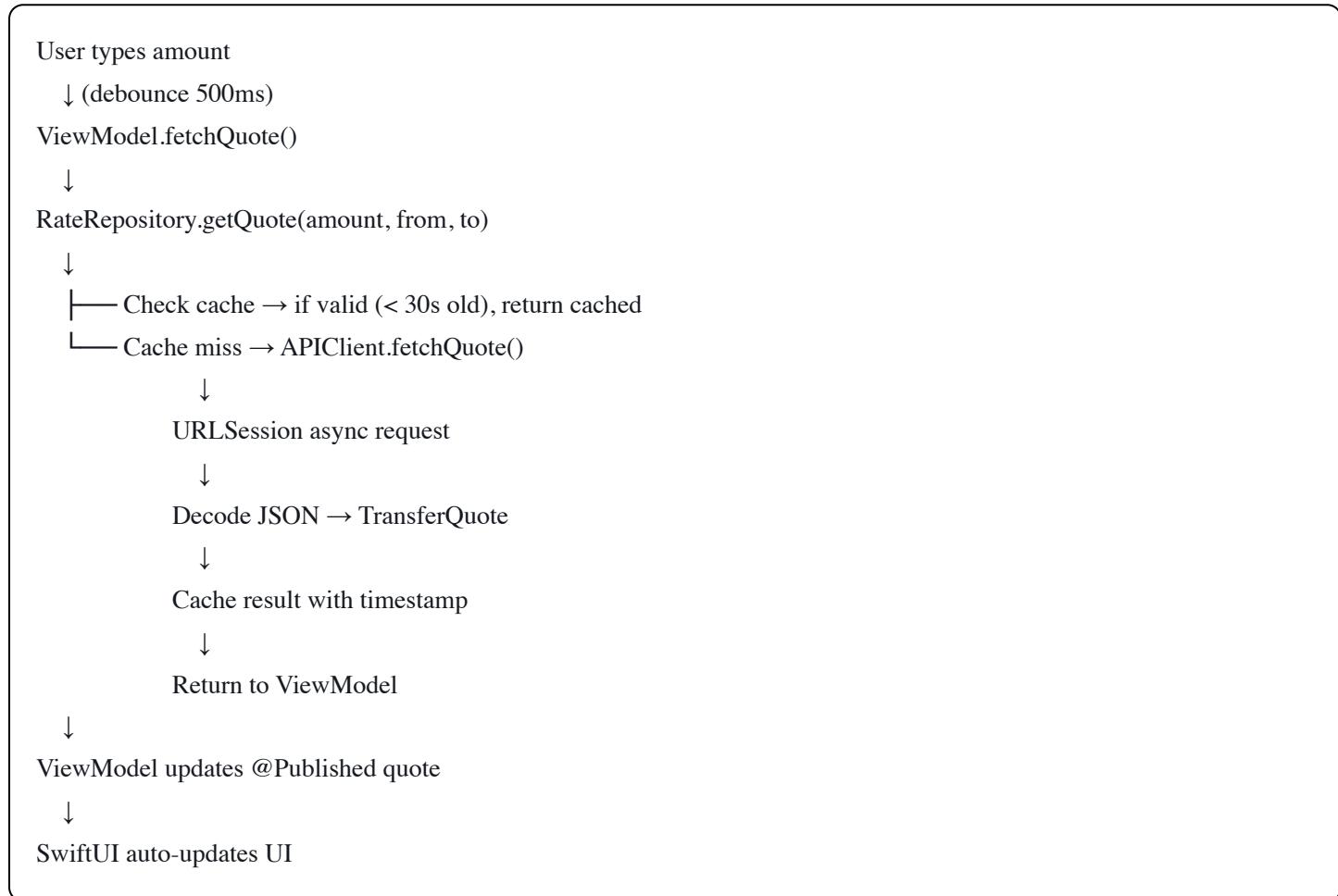
```
TransferAmountView
└── TransferAmountViewModel
    ├── @Published amount: String
    ├── @Published selectedCurrency: CurrencyPair
    ├── @Published quote: TransferQuote?
    ├── @Published isLoading: Bool
    ├── @Published error: TransferError?
    |
    └── fetchQuote() → debounce 500ms → RateRepository
        └── confirmTransfer() → TransferRepository
```

#### Rate Fetching Strategy:

- User types amount → debounce 500ms (don't fetch on every keystroke)
- Fetch quote from API: amount + fee + rate + expiry timestamp
- Cache rate with TTL (30 seconds — matches Wise's actual TTL)
- Show countdown timer on screen
- When expired → auto-refetch or grey out confirm button

**Why debounce?** "If the user types '1000', we'd fire 4 API calls (1, 10, 100, 1000) without debouncing. That wastes bandwidth, loads the backend unnecessarily, and causes UI flickering. 500ms debounce waits until the user stops typing."

### Data Flow:



### Transfer Confirmation — The Critical Part

- User taps "Confirm"
- ↓
1. Check quote not expired (client-side)

↓

  2. Generate idempotency key (UUID) — stored locally

↓

  3. POST /v1/transfers (with idempotency key in header)

↓

  4. Backend validates:
    - Rate still valid? (server-side check)
    - Sufficient balance?
    - Compliance checks (KYC/AML)
    - Fraud scoring

↓

  5. Response: Transfer created (status: PROCESSING)

↓

  6. Mobile navigates to status tracking screen

↓

  7. Poll or WebSocket for status updates

**Why idempotency key?** "If the network drops after the user taps confirm but before we receive the response, the user might tap again. Without idempotency, we'd create two transfers. With it, the backend recognizes the duplicate and returns the same transfer. In fintech, this is non-negotiable."

### Trade-off: Poll vs WebSocket for status updates?

"I'd start with polling every 5 seconds. WebSocket is more real-time but adds complexity — connection management, reconnection logic, and background/foreground handling on iOS. For transfer status that changes on a minutes-not-milliseconds timescale, polling is simpler and sufficient. If we later need real-time (like a live rate ticker), WebSocket makes more sense."

### What libraries would you use?

- **Networking:** URLSession (no third-party dependency for a fintech app — fewer supply chain risks)
- **JSON Parsing:** Codable (built-in, type-safe)
- **Image Loading:** Kingfisher or SDWebImage (for recipient avatars)
- **Caching:** NSCache for in-memory, FileManager for disk
- **State Management:** Combine + @Published (native, no third-party)
- **DI:** Protocol-based manual injection (not Swinject — too magical)
- **Analytics:** Firebase Analytics or Wise's own telemetry SDK

**Why URLSession over Alamofire?** "In a fintech app, I want minimal third-party dependencies for security audit reasons. URLSession with async/await is clean enough. Alamofire adds convenience but also adds a dependency to audit and maintain."

## What questions would you ask backend/product/design?

### For Backend:

- "What's the rate guarantee window? 30 seconds?"
- "What does the API response look like when a rate expires mid-transfer?"
- "Is there a webhook for transfer status changes, or do I poll?"
- "How is the idempotency key validated — per user, per session, or global?"

### For Product:

- "Should we show the breakdown (amount → fee → converted amount) live as user types?"
- "What happens if user leaves the app mid-transfer? Resume or restart?"
- "Do we need to support saving draft transfers?"

### For Design:

- "What's the loading state UX — skeleton, spinner, or shimmer?"
  - "How do we show rate expiry — countdown timer, or just grey out?"
  - "What's the error UX for insufficient balance vs expired rate?"
- 

## SCENARIO 2: Multi-Currency Wallet / Balance Feature

### Prompt:

"Wise shows users their balances in multiple currencies. How would you architect this on mobile?"

### Key Architecture Decisions:

#### Data Model:

```
swift

struct WalletBalance {
    let currency: String
    let amount: Decimal
    let lastUpdated: Date
}

struct WalletState {
    let balances: [WalletBalance]
    let totalInBaseCurrency: Decimal
    let baseCurrency: String
}
```

## Caching Strategy:

- Cache balances locally (encrypted — this is financial data)
- Show cached data immediately on app launch
- Fetch fresh data in background
- Update UI when fresh data arrives (optimistic UI)
- Show "Last updated: 2 min ago" indicator

## Pull-to-Refresh:

```
App Launch → Show cached balances → Fetch fresh → Update UI  
Pull-to-Refresh → Show loading indicator → Fetch fresh → Update UI  
Background → SceneWillEnterForeground → Fetch fresh silently
```

## Trade-off: Optimistic UI vs Wait for fresh data?

"I'd show cached balances immediately because waiting for a network call on every app open creates perceived slowness. The risk is showing stale balance — but we mitigate this with the 'Last updated' timestamp. If a user just completed a transfer, we optimistically deduct the amount locally while the backend confirms."

## Offline Handling:

- Show cached balances with offline indicator
- Disable transfer button if cached data is too old (> 5 min)
- Queue balance refresh for when connectivity returns

---

## SCENARIO 3: Real-Time Exchange Rate Display

### Prompt:

"We want to show live exchange rates that update every few seconds. How would you design this?"

### Architecture:

#### Option A: Polling (Simple)

```
Timer (every 5s) → API call → Update cache → Update UI
```

Pros: Simple, works with REST APIs

Cons: Unnecessary requests when rate hasn't changed, battery drain

#### Option B: WebSocket (Real-time)

Connect WebSocket → Receive rate events → Update cache → Update UI

Pros: True real-time, server pushes only when rate changes

Cons: Connection management complexity, background/foreground handling

### Option C: Server-Sent Events (SSE)

Open SSE connection → Receive rate stream → Update cache → Update UI

Pros: Simpler than WebSocket, works with HTTP

Cons: One-directional (server → client only)

### My Recommendation for Wise:

"I'd use WebSocket for the rate ticker screen and polling for other screens. The rate ticker is where real-time matters. On other screens (transfer form), polling with a 30-second interval is sufficient because the rate is only 'locked' when the user gets a quote."

### Mobile-Specific Concerns:

- **Battery:** Disconnect WebSocket when app backgrounds, reconnect on foreground
- **Memory:** Only keep latest rate per pair in memory, not history
- **UI:** Animate rate changes (green flash for up, red for down)

---

## SCENARIO 4: Card Payment / Wise Card Feature

### Prompt:

"Wise has a debit card. How would you architect the card transaction notification system on mobile?"

### Architecture:

Card swipe → Payment processor → Wise backend → Push notification → Mobile app

↓

Store transaction

↓

Mobile polls/refreshes

### Push Notification Flow:

1. Backend receives transaction event from Visa/Mastercard
2. Backend sends push notification via APNs
3. App receives notification → shows banner
4. User taps → navigate to transaction detail

## 5. Background: update local transaction cache

### Key Concerns:

- **Speed:** Notification should arrive within seconds of card swipe
  - **Accuracy:** Amount shown in notification must match final settlement
  - **Offline:** If phone is offline, APNs queues the notification
  - **Rich notifications:** Show merchant name, amount, category icon
- 

## SCENARIO 5: Event-Driven Architecture (Confirmed Glassdoor Mar 2025)

### What was asked:

"Design for event-driven architecture and data buckets"

### How to approach:

#### Event-Driven on Mobile:

Backend events (Kafka) → API/WebSocket → Mobile event handler → Update state

Example events:

- transfer.status.changed
- rate.updated
- balance.changed
- card.transaction.created
- kyc.status.updated

### Mobile Event Processing:

swift

```

// Event router pattern
class EventRouter {
    private var handlers: [String: [(Any) -> Void]] = [:]

    func register<T>(event: String, handler: @escaping (T) -> Void) {
        handlers[event, default: []].append { data in
            if let typed = data as? T { handler(typed) }
        }
    }

    func dispatch(event: String, data: Any) {
        handlers[event]?.forEach { $0(data) }
    }
}

// Usage:
// router.register(event: "transfer.status.changed") { (status: TransferStatus) in
//     transferStore.updateStatus(status)
// }

```

**"Data Buckets" likely means:** Partitioning data by domain/type

- Transfer bucket: all transfer-related state
- Balance bucket: wallet balances per currency
- Rate bucket: cached exchange rates
- User bucket: profile, KYC status, preferences

Each "bucket" has its own cache policy, refresh strategy, and TTL.

---

## THE CONTENTION & SCALING PATTERN (Reddit confirmed)

**What they're really asking:**

"Multiple users trying to do the same thing at the same time — how do you handle it?"

**Mobile Perspective on Contention:**

**Problem 1: Rate Race Condition**

- User A and User B both get rate quote at same time
- Rate changes between their quotes and confirmations
- Solution: Quote with UUID, server validates quote hasn't expired

**Problem 2: Balance Race Condition**

- User sends £500 from phone AND tablet simultaneously
- Only £600 in account
- Solution: Backend uses optimistic locking or serialized writes
- Mobile: Show pending state, refresh balance after transfer

### Problem 3: Write Scaling

- Millions of transfers created per day
- Solution:
  - Partitioning by user/region
  - Message queue for async processing
  - Idempotency keys prevent duplicates
  - Mobile: show "Processing..." status, don't block on completion

### How to discuss from mobile perspective:

"The mobile client's role in handling contention is mainly about: 1) generating idempotency keys so retries are safe, 2) showing accurate loading/pending states so users don't double-tap, 3) refreshing data aggressively after writes to show the latest state, and 4) gracefully handling 409 Conflict responses from the API."

---

## WISE-SPECIFIC TECH TO MENTION

### Dynamic Forms (from Wise's tech blog)

Wise built a framework called Dynamic Forms — server-driven UI. Business logic is defined once on the backend and rendered natively on iOS/Android/Web.

"I read about Wise's Dynamic Forms approach — where the backend defines the form structure and the mobile client renders it natively. This is smart for a fintech because regulatory requirements change per country, and you don't want to ship an app update every time a KYC field changes in Lithuania. I've worked with similar patterns at PayPal where server-driven config controls feature rollouts."

### Wise's Actual Stack (from 2025 Tech Blog)

- **Mobile:** Native iOS (Swift) + Android (Kotlin)
- **Backend:** Java microservices
- **Messaging:** Kafka
- **Database:** PostgreSQL + read replicas
- **Cache:** Redis
- **CDN/Static:** CloudFront
- **CI/CD:** Spinnaker

- **Monitoring:** Grafana, custom observability
- 

## TRADE-OFF CHEAT SHEET

Decision	Option A	Option B	When to pick A	When to pick B
<b>Data fetching</b>	Polling	WebSocket	Most screens, simple features	Real-time rates, live status
<b>Caching</b>	In-memory (NSCache)	Disk (CoreData)	Small, volatile data (rates)	Large, persistent data (history)
<b>Architecture</b>	MVVM	VIPER/Clean	Most features	Complex features with many use cases
<b>Navigation</b>	SwiftUI NavigationStack	Coordinator	Simple flows	Complex multi-step flows (transfer)
<b>Networking</b>	URLSession	Alamofire	Fintech (fewer deps)	Non-critical apps
<b>State</b>	@Published + Combine	Redux/TCA	Per-screen state	Shared global state
<b>DI</b>	Manual (protocol)	Swinject	Small/medium apps	Large apps with many deps
<b>Storage</b>	UserDefaults	Keychain	Non-sensitive settings	Tokens, financial data
<b>Images</b>	AsyncImage	Kingfisher	Few images	Image-heavy screens
<b>Updates</b>	Optimistic UI	Wait for server	Balance display	Transfer confirmation

---

## QUESTIONS TO ASK THEM (End of interview)

1. "How does the mobile team handle server-driven UI? Do you use Dynamic Forms for the transfer flow?"
  2. "What's the typical API response time you target for the mobile client?"
  3. "How does the mobile team handle feature flags and gradual rollouts?"
  4. "What's the biggest mobile architecture challenge the team is facing right now?"
  5. "How do you handle backward compatibility when the API changes?"
- 

## PREPARATION CHECKLIST

- Can I draw a clean mobile architecture diagram in 3 minutes?
  - Can I walk through a transfer flow end-to-end (user tap → backend → response → UI)?
  - For every design decision, can I say "because..." and give a trade-off?
  - Do I know Wise's tech stack well enough to mention relevant pieces?
  - Can I discuss caching strategies with TTL for exchange rates?
  - Can I explain idempotency and why it matters for fintech?
  - Can I discuss offline/error/loading states for each screen?
  - Do I have questions about backend, product, and design concerns?
  - Can I discuss event-driven architecture from a mobile perspective?
  - Am I prepared for "now what if we add X?" follow-up changes?
- 

## THE #1 THING THAT WILL SET YOU APART

You work on BFX at PayPal — a system that handles foreign exchange for Visa multi-currency cards. When they describe the transfer feature, you can say:

"This is very similar to what I build at PayPal. In the BFX system, we handle rate quoting, fee calculation, and multi-currency conversion. One thing I learned is that the rate guarantee window is the most critical design decision — too short and users get frustrated re-confirming, too long and the business absorbs FX risk. I'd want to understand what Wise's window is to design the mobile experience around it."

This instantly shows domain expertise that 99% of candidates won't have.