WISE ROUND 3: TECHNICAL INTERVIEW — EXHAUSTIVE QUESTION BANK

Every Possible Feature Scenario with Full Answers

60 min | Two Engineers | Zoom

HOW THIS ROUND ACTUALLY WORKS

Re-reading the candidate pack EXACTLY:

> "The interviewers will provide you with some 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 some details about **proposed improvements** to this feature and look for **how you would approach building it**."

This is a 3-phase conversation:

1. **"Here's a feature. How do you think it's built?"** → Reverse-engineer the architecture
2. **"Now here are improvements we want."** → Extend the design
3. **"How would you break this into work?"** → Show planning/leadership thinking

What they assess:

- 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 need answered from backend, product, design
- Knowledge of **best practices, design patterns, scalable solutions**

CONFIRMED QUESTIONS FROM ALL SOURCES

Source	Question/Pattern	Date
Reddit	Contention and scaling writes/reads with financial twist	2024
Glassdoor (Mar 2025)	Event-driven architecture and data buckets (Excalidraw)	2025
Glassdoor (Dec 2024)	Design a payment system (**got offer**)	2024
Glassdoor (Nov 2025)	"Should the experience be synchronous or async with notification?"	2025
Glassdoor (May 2025)	Practical questioning, not textbook	2025

Source	Question/Pattern	Date
Glassdoor iOS Dev	Build an app with public API + table view (take-home)	Older
Wise Official	"For iOS/Android, may be asked take-home coding problem"	Current
Wise EM Tips	"Architectural perspective, high-level design choices"	Current
Reddit Candidate	"I didn't justify my trade-offs and got lost/confused"	2024

Critical pattern: They LEAD the conversation. You respond. The Reddit candidate said "They led most of it." So don't try to monologue — it's a guided discussion.

EVERY POSSIBLE WISE APP FEATURE AS A SCENARIO

I've mapped every major Wise mobile app feature. They'll pick ONE and go deep.

SCENARIO 1: SEND MONEY TRANSFER (HIGHEST PROBABILITY)

Their prompt:

> "Wise lets users send money internationally. Imagine this feature already exists. How do you think the mobile architecture works?"

Your reverse-engineering answer:

```
USER JOURNEY:
Enter amount → See rate + fee → Choose recipient → Pick payment method
→ Review & confirm → Track status

ARCHITECTURE I'D EXPECT:

┌─ TransferFlowCoordinator (manages multi-step navigation)
│
├─ Step 1: AmountInputView + AmountInputViewModel
│   └── Debounced input → RateRepository → QuoteService
│       └── Cache quote with 30s TTL
│       └── Show: "You send £1000, they get €1,155.23, fee £3.69"
│
├─ Step 2: RecipientSelectionView + RecipientViewModel
│   └── Load saved recipients from local DB
│   └── Search/filter by name
│   └── "Add new recipient" → sub-flow
│
├─ Step 3: PaymentMethodView + PaymentMethodViewModel
│   └── Fetch available methods (bank transfer, card, Apple Pay)
│   └── Show fee difference per method
│   └── Card → tokenized via Stripe/Adyen SDK
│
├─ Step 4: ReviewView + ReviewViewModel
│   └── Final summary with all details
│   └── Rate countdown timer (30s)
│   └── If expired → auto-refetch quote, update amounts
│   └── Confirm button → POST with idempotency key
│
└─ Step 5: StatusTrackingView + StatusViewModel
    └── Poll every 5s OR WebSocket for updates
    └── States: Created → Processing → Sent → Delivered
    └── Push notification on completion
```

Libraries I'd expect:

- `URLSession` with async/await for networking
- `Combine` for reactive state binding
- `CoreData` or SQLite for local recipient storage
- `Keychain` for storing auth tokens
- Stripe/Adyen SDK for card tokenization (if applicable)
- `UNUserNotificationCenter` for push notifications

Questions I'd ask backend:

- "What's the rate guarantee window? 30 seconds?"
- "Is the quote idempotent — same request returns same quote?"
- "What error codes should I handle? Rate expired? Insufficient funds?"
- "Does the POST return immediately or block until processing starts?"

Questions I'd ask product:

- "Can users save a transfer as draft and resume later?"
- "Should the rate auto-refresh or require a manual tap?"
- "What's the maximum time a user spends on this flow typically?"

Questions I'd ask design:

- "How do we show the fee breakdown — expanded by default or collapsible?"
- "What's the loading state between steps?"
- "What accessibility considerations for the amount input?"

Now their improvement:

> "We want to add scheduled/recurring transfers. How would you approach this?"

Your answer:

```
NEW COMPONENTS NEEDED:

1. SchedulePickerView
   - Frequency: one-time future date, weekly, monthly
   - Date/time picker
   - End date (optional for recurring)

2. ScheduledTransferRepository
   - Local storage for scheduled transfers
   - Sync with backend

3. BackgroundScheduler
   - Two options:
     a) Server-side scheduler (backend creates at scheduled time)
        → Simpler mobile, reliable even if phone is off
     b) Client-side with BGTaskScheduler
        → Works offline, but unreliable (iOS kills background tasks)

   → I'd recommend server-side. Send schedule to backend,
     backend creates transfer at the right time.
     Mobile just displays upcoming scheduled transfers.

4. Rate handling for scheduled transfers
   - Can't guarantee today's rate for next week's transfer
   - Two options:
     a) Lock rate at schedule time (business absorbs risk)
     b) Use market rate at execution time (user accepts variance)
   → Ask product which approach Wise prefers

DISCRETE WORK ITEMS:
- [ ] SchedulePickerView UI (2 days)
- [ ] ScheduledTransfer data model + API contract (1 day)
- [ ] Backend API: POST /scheduled-transfers (backend team)
- [ ] Local storage for viewing upcoming schedules (1 day)
- [ ] Push notification: "Your scheduled transfer was sent" (0.5 day)
- [ ] Cancel/edit scheduled transfer flow (1.5 days)
- [ ] Edge case: what if balance insufficient at execution time? (1 day)
- [ ] Unit tests + UI tests (2 days)
```

Trade-off to articulate:

> "I'd do server-side scheduling over client-side because iOS aggressively kills background tasks — BGTaskScheduler is unreliable for time-critical financial operations. If a user schedules a transfer for Monday 9 AM and their phone is dead, the server handles it. The downside is the user needs internet when creating the schedule, but that's a reasonable requirement for a financial operation."

SCENARIO 2: MULTI-CURRENCY BALANCE & CONVERSION

Their prompt:

> "Wise shows users their balances in 40+ currencies with the ability to convert between them. How would you expect this works on mobile?"

Your reverse-engineering:

```
ARCHITECTURE:

BalanceDashboardView
├── BalanceDashboardViewModel
│   ├── @Published balances: [CurrencyBalance]
│   ├── @Published totalInBaseCurrency: Decimal
│   ├── @Published baseCurrency: String (user preference)
│   └── loadBalances()
│
├── Data Flow:
│   ViewModel → BalanceRepository
│   ├── Check cache (encrypted local storage)
│   │   └── If fresh (< 60s) → return cached
│   │   └── If stale → return cached + fetch in background
│   └── Fetch from API → update cache → notify ViewModel
│
├── Conversion Sub-flow:
│   ConvertView + ConvertViewModel
│   ├── Source currency selector
│   ├── Target currency selector
│   ├── Amount input with debounce
│   ├── Live rate display (from RateCache, 30s TTL)
│   └── Confirm → instant conversion (balance-to-balance)

KEY DESIGN DECISIONS:

1. LOCAL STORAGE:
   - Balances = sensitive data → encrypt at rest
   - Use Keychain for small data, encrypted CoreData for history
   - NOT UserDefaults (plaintext)

2. TOTAL CALCULATION:
   - Total in base currency requires rates for ALL held currencies
   - Batch fetch rates: GET /rates?pairs=USD-SGD,EUR-SGD,GBP-SGD
   - Cache rates, recalculate total locally

3. REAL-TIME UPDATES:
   - Balance changes when: transfer sent/received, card payment, conversion
   - Options: poll on foreground, push notification triggers refresh,
     or WebSocket for live updates
   - I'd use: push notification → trigger silent refresh
```

Their improvement:

> "Now we want to add a feature where users can set rate alerts — 'notify me when EUR/SGD reaches 1.50.' How would you add this?"

Your answer:

```
COMPONENTS:

1. RateAlertSetupView
   - Currency pair picker
   - Target rate input
   - Direction: above or below
   - Toggle: one-time or persistent

2. Backend:
   - POST /rate-alerts { pair, targetRate, direction, userId }
   - Backend monitors rates and sends push when condition met
   - Mobile does NOT poll for this — backend pushes

3. Mobile responsibilities:
   - Create/edit/delete alerts UI
   - Receive push notification → deep link to conversion screen
   - Show active alerts in a list

WHY SERVER-SIDE:
"Rate monitoring should be server-side because:
 a) The app might not be running when the rate hits the target
 b) Rates change every few seconds — polling from every user's phone
    would overwhelm both the backend and the user's battery
 c) Backend can batch-check alerts efficiently"
```

SCENARIO 3: WISE CARD TRANSACTIONS & NOTIFICATIONS

Their prompt:

> "When a user pays with their Wise card, they get an instant notification. How do you think the mobile notification and transaction display system works?"

Your reverse-engineering:

```
FLOW:
Card swipe → Visa/Mastercard → Wise payment processor
→ Wise backend → APNs/FCM → Mobile app

MOBILE ARCHITECTURE:

1. Push Notification Handling:
   - AppDelegate / UNUserNotificationCenter
   - Parse notification payload → extract transaction data
   - If app is foreground: show in-app banner (not system notification)
   - If background: show system notification
   - Tap → deep link to transaction detail

2. Transaction List:
   TransactionListView + TransactionListViewModel
   ├── Paginated loading (cursor-based, not offset)
   ├── Pull-to-refresh
   ├── Section by date (Today, Yesterday, This Week, etc.)
   ├── Each item: merchant icon, name, amount, currency, category
   └── Optimistic update: when push arrives, prepend to local list
       before API confirmation

3. Rich Notification:
   - UNNotificationServiceExtension
   - Download merchant logo
   - Show: "💳 Starbucks — SGD 6.50 from your SGD balance"
   - Action buttons: "View" | "Flag as suspicious"

CACHING STRATEGY:
- Last 50 transactions cached locally for instant display
- Older transactions: paginated API fetch
- New transaction from push → insert at top of local cache
- Background refresh on app foreground
```

Their improvement:

> "We want to add spending categories and a monthly spending summary. How would you add this?"

Your answer:

```
1. Categories:
   - Backend assigns category per transaction (ML-based + merchant mapping)
   - Mobile receives category in transaction payload
   - Display: category icon + color coding in list
   - Filter transactions by category

2. Monthly Summary:
   SpendingSummaryView + SpendingSummaryViewModel
   ├── Fetch: GET /spending-summary?month=2026-02
   ├── Response: { categories: [{ name, total, percentage, transactions }] }
   ├── Display: pie/donut chart (use Swift Charts framework)
   ├── Tap category → filtered transaction list
   └── Compare with previous month (% change)

3. Chart Library Decision:
   - Swift Charts (iOS 16+) → native, lightweight, no dependency
   - vs Charts by Daniel Gindi → more features, but third-party
   → I'd use Swift Charts for a fintech app (minimize dependencies)

WORK BREAKDOWN:
- [ ] Category model + API integration (1 day)
- [ ] Category icons and color mapping (0.5 day)
- [ ] Transaction list filter by category (1 day)
- [ ] Monthly summary API + ViewModel (1.5 days)
- [ ] Donut chart with Swift Charts (1 day)
- [ ] Month-over-month comparison (0.5 day)
- [ ] Unit tests (1 day)
Total: ~7 engineering days
```

SCENARIO 4: RECIPIENT MANAGEMENT

Their prompt:

> "Users save recipients they frequently send money to. How would you build the recipient management feature?"

Your reverse-engineering:

```
ARCHITECTURE:

RecipientListView
├── RecipientListViewModel
│   ├── @Published recipients: [Recipient]
│   ├── @Published searchText: String
│   ├── filteredRecipients (computed, filters by search)
│   └── Data source: RecipientRepository
│       ├── Local DB (CoreData) as source of truth for display
│       ├── Sync with backend on app launch + pull-to-refresh
│       └── Offline-first: show local data even without network

RECIPIENT MODEL:
struct Recipient {
    let id: String
    let name: String
    let email: String?
    let bankDetails: BankDetails  // varies by country!
    let currency: String
    let country: String
    let lastUsed: Date?
    let isFavorite: Bool
}

KEY CHALLENGE: Bank details vary by country
- UK: sort code + account number
- EU: IBAN
- US: routing number + account number
- India: IFSC code + account number
- Each country has different validation rules

→ This is where Wise's Dynamic Forms comes in:
  Backend sends form schema per country
  Mobile renders the form dynamically
  No app update needed when adding a new country
```

Their improvement:

> "We want to add Wise-to-Wise transfers using email/phone. How does this change things?"

Your answer:

```
NEW FLOW:
- User enters email or phone instead of bank details
- App checks: does this email/phone belong to a Wise user?
  - Yes → show name for confirmation → instant free transfer
  - No → offer to send invite OR fall back to bank transfer

ARCHITECTURE CHANGES:
1. RecipientResolver: new component
   - POST /recipients/resolve { email/phone }
   - Returns: { isWiseUser: true, name: "...", avatar: "..." }

2. Transfer logic splits:
   - Wise-to-Wise: instant, free, simpler flow
   - Bank transfer: existing flow with fees and rate

3. Privacy concern (ask product!):
   "If I type random emails, can I discover who has a Wise account?
    We need rate limiting and possibly only resolving for contacts
    the user actually knows."
```

SCENARIO 5: KYC / IDENTITY VERIFICATION

Their prompt:

> "Wise needs to verify user identity before they can send money. How would you architect the verification flow on mobile?"

Your reverse-engineering:

```
VERIFICATION FLOW:
1. Upload ID document (passport/license)
2. Selfie verification (liveness check)
3. Address proof (utility bill)
4. Backend review → approved/rejected

MOBILE ARCHITECTURE:

VerificationCoordinator (multi-step flow)
├── DocumentCaptureView
│   ├── Camera with overlay guide
│   ├── Auto-capture when document detected (Vision framework)
│   ├── Image quality check (blur detection, lighting)
│   └── Upload with progress indicator
│
├── SelfieVerificationView
│   ├── Face detection using Vision/ARKit
│   ├── Liveness check: "Turn head left, blink, smile"
│   ├── Prevents photo-of-photo attacks
│   └── Upload selfie + liveness data
│
├── AddressProofView
│   ├── Upload utility bill photo
│   └── OR auto-verify via bank integration
│
└── VerificationStatusView
    ├── Pending → "We're reviewing your documents"
    ├── Approved → "You're verified!" → unlock transfers
    ├── Rejected → "Please re-submit" with reason
    └── Push notification on status change

LIBRARIES:
- AVFoundation for camera
- Vision framework for document/face detection
- Might use 3rd party: Onfido, Jumio, or Wise's own SDK

KEY CONCERNS:
- Image compression before upload (photos are 5-12MB)
- Resume upload if interrupted
- Multi-part form upload with progress
- Privacy: don't cache ID images locally after upload
- Accessibility: voice-guided capture for visually impaired
```

Their improvement:

> "Verification takes 1-24 hours. How do you handle the waiting experience?"

Your answer:

```
1. Status polling: check every 5 minutes when app is in foreground
2. Push notification: backend sends "Verification complete" push
3. Deep link from notification → navigate to result screen
4. Local state machine:
   .notStarted → .documentsUploaded → .underReview → .approved/.rejected
5. Show estimated time: "Usually takes 2-4 hours"
6. Allow user to continue browsing app (see rates, add recipients)
   but block actual transfers until verified
```

SCENARIO 6: TRANSFER STATUS TRACKING (WITH EVENT-DRIVEN ARCHITECTURE)

This maps to the confirmed Glassdoor question about event-driven architecture.

Their prompt:

> "After a user sends a transfer, they can track its progress in real-time. How would you architect the status tracking system?"

Your answer:

```
BACKEND EVENT FLOW (event-driven):

Transfer created → Kafka event: transfer.created
Payment received → Kafka event: transfer.payment.received
Converting currency → Kafka event: transfer.converting
Sending to recipient → Kafka event: transfer.sending
Money delivered → Kafka event: transfer.delivered

Each event → notification service → APNs push to mobile

MOBILE ARCHITECTURE:

TransferStatusView + TransferStatusViewModel
├── Timeline UI showing each step
│   ✅ Payment received
│   ✅ Converting your money
│   🔄 Sending to recipient's bank (current)
│   ⬜ Money delivered
│
├── Data source: TransferStatusRepository
│   ├── Initial load: GET /transfers/{id}/status
│   ├── Updates:
│   │   Option A: Poll every 10s (simpler)
│   │   Option B: WebSocket subscription (real-time)
│   │   Option C: Push notification triggers refresh (recommended)
│   └── Cache locally for offline viewing
│
├── Estimated arrival time
│   └── Backend provides estimate based on route + payment method
│   └── Update estimate as transfer progresses
│
└── Action buttons
    └── Cancel (if still in early stage)
    └── Contact support
    └── Share tracking link with recipient

TRADE-OFF — Push + Refresh vs WebSocket:
"I'd use push notifications triggering a silent background refresh over
WebSocket because:
1. WebSocket requires persistent connection → battery drain
2. Transfer status changes happen minutes apart, not seconds
3. Push notification works even when app is closed
4. Simpler to implement and maintain
5. WebSocket makes sense for rate tickers (second-by-second),
   not for transfer status (minute-by-minute)"
```

```
THE EVENT-DRIVEN DISCUSSION:
"From the mobile perspective, I don't need to know about Kafka directly.
What I care about is: when a backend event occurs, how does the mobile
get notified? The contract between mobile and backend is:

1. Push notification with event type + minimal data
2. Mobile receives push → fetches full status from REST API
3. This decouples mobile from the event system — mobile doesn't need
   to understand Kafka, just the REST contract

The 'data buckets' concept maps to: each domain (transfers, balances,
rates) has its own event stream and its own mobile cache. A transfer
event doesn't invalidate the rate cache."
```

SCENARIO 7: RATE CONVERTER / COMPARISON TOOL

Their prompt:

> "Wise shows users the mid-market rate and compares it with banks. How would you build the rate comparison screen?"

Your answer:

```
ARCHITECTURE:

RateComparisonView + RateComparisonViewModel
├── CurrencyPairSelector (from/to)
├── AmountInput (debounced)
├── WiseRate: { midMarketRate, fee, totalCost, deliveryTime }
├── BankComparisons: [{ bankName, rate, fee, totalCost, savingsVsWise }]
│   └── Backend provides this data (Wise scrapes bank rates)
└── Savings highlight: "You save £42.50 vs your bank"

CACHING:
- Wise's own rate: cache 30s (changes frequently)
- Bank comparison data: cache 1 hour (updates less frequently)
- Separate cache buckets with different TTLs

CHART:
- Show historical rate chart (7 day / 30 day / 90 day)
- Use Swift Charts for the graph
- Data: GET /rates/history?pair=GBP-EUR&period=30d
- Cache aggressively (historical data doesn't change)
```

SCENARIO 8: ACTIVITY FEED / TRANSACTION HISTORY

Their prompt:

> "The activity feed shows all user transactions across transfers, card payments, and conversions. How would you architect this?"

Your answer:

```
ARCHITECTURE:

ActivityFeedView + ActivityFeedViewModel
├── @Published activities: [ActivityItem]
├── Pagination: cursor-based (not offset — items may be inserted)
├── Pull-to-refresh
├── Section headers by date
│
├── ActivityItem (polymorphic):
│   enum ActivityType {
│       case transfer(TransferActivity)
│       case cardPayment(CardActivity)
│       case conversion(ConversionActivity)
│       case feeCharge(FeeActivity)
│   }
│
├── Data flow:
│   1. App launch → load cached activities from local DB
│   2. Background fetch → API call for new activities
│   3. Merge: insert new items, update existing
│   4. Conflict resolution: server is source of truth
│
└── Search/Filter:
    ├── By type (transfers, card, conversion)
    ├── By currency
    ├── By date range
    └── By amount range

KEY DECISIONS:

1. PAGINATION:
  Cursor-based because new transactions can be inserted at any time.
  Offset-based would skip/duplicate items when new data is added.

2. LOCAL STORAGE:
  CoreData or SQLite with FTS (Full-Text Search) for searching
  by merchant name, recipient name, etc.

3. REAL-TIME:
  New activities arrive via push notification → insert at top
  Don't poll — waste of bandwidth for a list that updates infrequently

4. RENDERING:
  Use DiffableDataSource (UIKit) or List with .id (SwiftUI)
  for efficient updates without full reload
```

CROSS-CUTTING CONCERNS (MENTION THESE FOR ANY SCENARIO)

Security

- All API calls over HTTPS with certificate pinning
- Sensitive data (balances, transactions) encrypted at rest
- Auth tokens in Keychain, never UserDefaults
- Biometric auth (Face ID / Touch ID) for sensitive actions
- Screen protection: blur content when app goes to background

Accessibility

- VoiceOver support for all screens
- Dynamic Type for text sizing
- Minimum tap target 44x44 points
- Semantic labels on amounts: "You send one thousand pounds"

Analytics / Monitoring

- Track funnel: start → each step → completion → drop-off
- Crash-free rate target: 99.9%
- API response time tracking (p50, p95, p99)
- Error rate monitoring per endpoint

Testing

- Unit tests: ViewModel + Repository logic
- UI tests: critical flows (transfer, login)
- Snapshot tests: ensure UI consistency
- Mocking: protocol-based DI → inject mock network/storage

Performance

- App launch time < 2 seconds
- Screen transition < 300ms
- Image lazy loading + placeholder
- List virtualization for long transaction history

YOUR BFX ADVANTAGE — USE IT!

For EVERY scenario, connect back to your PayPal experience:

Wise Feature	**Your PayPal Experience**	**What to Say**
Rate quoting with TTL	BFX rate guarantee window	"At PayPal, our BFX system handles rate expiry the same way — we quote a rate with a validity window and check server-side on confirmation"
Multi-currency balance	FRED system	"The FRED system I work on manages currency capabilities across BINs — I understand the complexity of multi-currency data"
Card notifications	RUNE payment feature	"I've worked on real-time payment notifications with the RUNE feature — handling display issues when experiment flags affect the UI"
Transfer status tracking	BFX ramp deployments	"I track country-by-country ramp status for BFX — monitoring dashboards, phased rollouts, error rate tracking"
Fee calculation	Express Checkout pricing	"At PayPal, fee structures vary by payment method and region — I've built systems that handle this complexity"
Cross-timezone coordination	Chennai + San Jose + Singapore	"I coordinate feature delivery across 3 time zones daily"

THE TRADE-OFF TEMPLATE (MEMORIZE THIS)

Every single design decision needs this structure:

> "I'd choose **[A]** over **[B]** because **[reason with context]**. The trade-off is **[what you lose]**, but in Wise's case, **[why that's acceptable]**. If **[requirement changes]**, I'd reconsider."

Examples:

> "I'd use **polling** over **WebSocket** for transfer status because statuses change on a minutes timescale, not seconds. The trade-off is slightly delayed updates, but in Wise's case where most transfers take minutes to hours, 10-second polling is indistinguishable from real-time to the user. If we were building a live trading feature where milliseconds matter, I'd switch to WebSocket."

> "I'd use **URLSession** over **Alamofire** because fintech apps should minimize third-party dependencies for security audit reasons. The trade-off is slightly more boilerplate for complex requests, but with async/await, URLSession is clean enough. If the team was already using Alamofire and had it audited, I'd continue using it rather than rewrite."

> "I'd store balances in **encrypted CoreData** over **Keychain** because Keychain is great for small secrets (tokens) but not designed for complex relational data like transaction history. The trade-off is more setup complexity, but CoreData gives us querying, sorting, and FTS that Keychain can't provide."

FINAL CHECKLIST BEFORE THE INTERVIEW

- [ ] Can I draw a clean mobile architecture for ANY Wise feature in 3 minutes?
- [ ] For every component, do I know WHY I chose it?
- [ ] Can I break any feature into discrete, estimable work items?
- [ ] Do I have specific questions for backend, product, AND design?
- [ ] Can I articulate 3+ trade-offs per scenario?
- [ ] Can I connect at least 2 points to my PayPal/BFX experience?
- [ ] Do I know Wise-specific tech (Dynamic Forms, Kafka, Spinnaker)?
- [ ] Am I ready for "now what if we add X?" without panicking?
- [ ] Can I discuss testing strategy for each feature?
- [ ] Do I mention security, accessibility, and monitoring naturally?