WISE ROUND 3 — FEATURE FLAGS, EXPERIMENTATION & ELMO

The Missing Piece That Will Set You Apart

WHY THIS MATTERS FOR THE INTERVIEW

From the candidate pack:

> "They'll provide details on a hypothetically existing feature... then give details about **proposed improvements** and look for how you would approach building it."

"Proposed improvements" almost always involves feature flags. When Wise says "we want to add X to an existing feature," the mobile architecture answer MUST include how you'd roll it out safely. Feature flags are the answer.

Wise specifically mentions on their system design page:

> "Product thinking: As 'product engineers' at Wise, we need to think from the customer's perspective"

Feature flags = product thinking + engineering safety. This is where your PayPal ELMO experience becomes a massive differentiator.

WHAT IS ELMO? (Your PayPal Experience)

ELMO is PayPal's internal experimentation and feature flag platform. From your work on RUNE:

- Experiment: `"Trmt_p2p_money_movement_flow"`
- Function: `isExperimentEnabled()` checks if user is in treatment group
- `qualifiedTreatment` value determines which UI variant to show
- **The bug you debugged:** Treatment was nil on fresh login but available after app relaunch — because experiment assignments hadn't been fetched/cached yet at login time

This is EXACTLY the kind of real-world problem Wise interviewers want to hear about.

HOW TO BRING THIS UP IN THE INTERVIEW

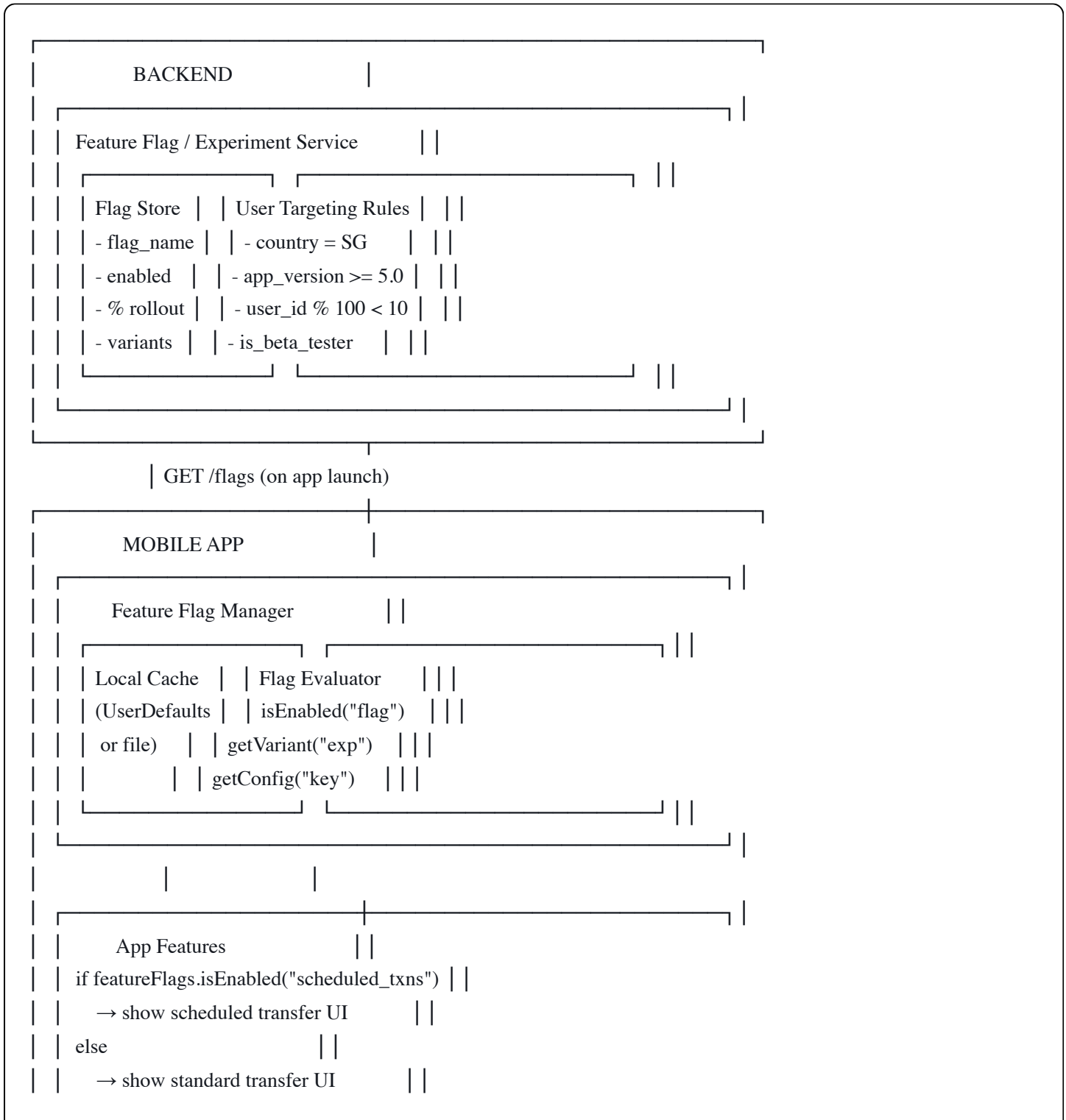
When they describe ANY feature and ask about improvements, say:

> "Before building this improvement, I'd want to understand the rollout strategy. At PayPal, we use an experimentation platform called ELMO for feature flags and A/B testing. I'd wrap this new feature behind a feature flag and roll it out gradually — maybe 5% → 25% → 50% → 100% — monitoring crash rates, conversion, and error rates at each stage.

> I've actually debugged a real production issue with this. On RUNE, we had an experiment flag called `Trmt_p2p_money_movement_flow` that controlled whether users saw the new Send/Request UI. The flag was returning nil on fresh login because the experiment assignments hadn't been fetched yet — the user got the wrong UI. After app relaunch, the cached assignment was available and it worked. This taught me that **experiment initialization timing is critical on mobile** — you need assignments ready before the first screen renders."

FEATURE FLAG ARCHITECTURE FOR MOBILE

How it works (draw this in the interview):



```
| └──────────────────────────────────────────────────────┘ |
└──────────────────────────────────────────────────────────┘
```

Swift Implementation:

```swift
```

```swift
// Feature Flag Manager — what you'd build at Wise
protocol FeatureFlagProvider {
    func isEnabled(_ flag: String) -> Bool
    func variant(_ experiment: String) -> String?
    func config<T: Codable>(_ key: String) -> T?
    func refresh() async
}

class FeatureFlagManager: FeatureFlagProvider {

    // Local cache of flags
    private var flags: [String: FlagValue] = [:]
    private let storage: FlagStorage
    private let apiClient: FlagAPIClient
    private let userId: String

    struct FlagValue: Codable {
        let enabled: Bool
        let variant: String?      // "control", "treatment_a", "treatment_b"
        let config: [String: AnyCodable]?
        let fetchedAt: Date
    }

    init(userId: String, storage: FlagStorage, apiClient: FlagAPIClient) {
        self.userId = userId
        self.storage = storage
        self.apiClient = apiClient

        // 1. Load cached flags FIRST (instant, before network)
        self.flags = storage.loadCachedFlags()
    }

    // Called on app launch — CRITICAL TIMING
    func refresh() async {
        do {
            let serverFlags = try await apiClient.fetchFlags(userId: userId)
            self.flags = serverFlags
            storage.cacheFlags(serverFlags)
        } catch {
            // Network failed — keep using cached flags
            // This is fine. Stale flags > no flags.
            print("Flag refresh failed, using cached: \(error)")
        }
    }

    func isEnabled(_ flag: String) -> Bool {
```

```
        flags[flag]?.enabled ?? false  // Default: OFF if unknown
    }

    func variant(_ experiment: String) -> String? {
        flags[experiment]?.variant
    }

    func config<T: Codable>(_ key: String) -> T? {
        // Return typed config value (e.g., rate_refresh_interval: 30)
        nil // Simplified
    }
}

// Storage protocol — testable
protocol FlagStorage {
    func loadCachedFlags() -> [String: FeatureFlagManager.FlagValue]
    func cacheFlags(_ flags: [String: FeatureFlagManager.FlagValue])
}

// API protocol — testable
protocol FlagAPIClient {
    func fetchFlags(userId: String) async throws -> [String: FeatureFlagManager.FlagValue]
}
```

The ELMO Bug Pattern — Initialization Timing:

swift

```
// ❌ THE BUG YOU FOUND AT PAYPAL:
class AppDelegate {
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions ...) {

        // User logs in...
        authenticateUser()

        // Feature flags fetched ASYNC — not ready yet!
        Task { await featureFlags.refresh() }

        // UI renders BEFORE flags are ready → nil treatment → wrong UI
        showHomeScreen()  // ← Uses flags that aren't loaded yet!
    }
}

// ✅ THE FIX — Ensure flags are ready before first screen:
class AppDelegate {
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions ...) {

        // 1. Load CACHED flags immediately (synchronous)
        featureFlags.loadFromCache()  // Instant — uses last-known values

        // 2. Show splash/loading screen
        showSplashScreen()

        // 3. Fetch fresh flags in background
        Task {
            await featureFlags.refresh()  // Updates cache

            // 4. Now show home screen with correct flags
            await MainActor.run {
                showHomeScreen()
            }
        }
    }
}
```

What to say in the interview:

> "At PayPal, I debugged a production bug where the experiment treatment was nil on fresh login because the flag fetch was async and the UI rendered before it completed. The fix was a two-phase approach: load cached flags synchronously on startup for instant availability, then refresh from the server in the background. This pattern ensures the user always sees a consistent experience — even if we show a slightly stale flag for a few seconds."

FEATURE FLAGS IN EVERY WISE SCENARIO

Here's how to weave feature flags into any Round 3 answer:

Scenario: Transfer Feature Improvements

> **Interviewer:** "We want to add Apple Pay as a payment method for transfers."
> **You:** "I'd wrap Apple Pay behind a feature flag with a gradual rollout:
>
> 1. **Flag:** `transfer_apple_pay_enabled`
> 2. **Initial rollout:** 5% of users in Singapore only
> 3. **Monitoring:** Track Apple Pay success rate vs card success rate, error rates, average transaction value
> 4. **Ramp:** If metrics are healthy after 1 week → 25% → 50% → 100%
> 5. **Kill switch:** If Apple Pay error rate exceeds 2%, auto-disable the flag
>
> On the mobile side:
>
> ```swift
> if featureFlags.isEnabled("transfer_apple_pay_enabled") {
>     paymentMethods.append(.applePay)
> }
> ```
>
> This is exactly how we do BFX rollouts at PayPal — country by country, monitoring error rates at each stage before expanding."

Scenario: New Onboarding Flow

> **Interviewer:** "We're testing a simplified onboarding with fewer steps."
> **You:** "This is a perfect A/B test:
>
> - **Experiment:** `onboarding_simplified_v2`
> - **Control (50%):** Current 5-step onboarding
> - **Treatment (50%):** New 3-step onboarding
> - **Success metric:** Onboarding completion rate
> - **Guard metrics:** KYC failure rate (shouldn't increase), support tickets
>
> ```swift
> switch featureFlags.variant("onboarding_simplified_v2") {
> case "treatment":
>     showSimplifiedOnboarding()
> case "control", nil:
>     showStandardOnboarding()
> }
> ```

> The key is the guard metric — if the simplified flow leads to more KYC failures because we skipped a validation step, the experiment fails even if completion rate improves."

Scenario: Rate Alert Feature

> **Interviewer:** "We want to add rate alerts — notify users when a rate hits their target."
> **You:** "I'd use a feature flag with server-driven config:
>
> - **Flag:** `rate_alerts_enabled`
> - **Config:** `{ max_alerts_per_user: 5, supported_pairs: ["USD-EUR", "GBP-EUR", ...] }`
> - **Rollout:** Start with premium users → all users
>
> The config approach means we can adjust limits without an app update. If we find that too many alerts are causing notification fatigue, we reduce `max_alerts_per_user` server-side, and the app immediately reflects the new limit."

FEATURE FLAG PATTERNS TO KNOW

1. Kill Switch

```swift
// Emergency disable of a problematic feature
if featureFlags.isEnabled("card_payments_enabled") {
    showCardPaymentOption()
}
// If card processor goes down → disable flag →
// all users stop seeing card option → no errors
```

2. Percentage Rollout

```swift
// Server determines: user_id % 100 < rollout_percentage
// Mobile just checks the boolean
if featureFlags.isEnabled("new_transfer_flow") {
    showNewTransferFlow()
}
```

3. A/B/n Test (Multiple Variants)

```swift
```

```
switch featureFlags.variant("checkout_button_experiment") {
case "green_button":
    button.backgroundColor = .green
case "blue_button":
    button.backgroundColor = .blue
case "animated_button":
    button.backgroundColor = .green
    addAnimation(button)
default:
    button.backgroundColor = .systemBlue  // Control
}
```

4. Server-Driven Config (Dynamic Forms style)

```
swift

// Wise's Dynamic Forms is essentially a feature flag on steroids
struct TransferFormConfig: Codable {
    let fields: [FormField]
    let validationRules: [ValidationRule]
    let paymentMethods: [String]
}

if let config: TransferFormConfig = featureFlags.config("transfer_form_sg") {
    renderDynamicForm(config)  // Server controls what the form looks like
}
```

5. Country-Specific Rollout (YOUR BFX EXPERIENCE)

```
swift

// This is EXACTLY what you do with BFX at PayPal
struct CountryRollout: Codable {
    let enabledCountries: [String]
    let rampPercentage: [String: Int]  // "SG": 100, "IT": 50, "LT": 10
}

if let rollout: CountryRollout = featureFlags.config("apple_pay_rollout"),
   rollout.enabledCountries.contains(userCountry) {
    showApplePay()
}
```

COMMON PITFALLS TO MENTION

1. Flag Staleness

> "Flags are fetched on app launch and cached. If the user keeps the app open for days without restarting, they might see outdated flags. Solution: refresh flags periodically (every 30 min) or on significant events (app foreground, after login)."

2. Flag Evaluation Before Login

> "Before a user is authenticated, you don't have their user ID for targeting. Solution: use device-based targeting for pre-login experiments (device ID hash), then switch to user-based after login."

3. Flag Flicker

> "If the cached flag says 'show old UI' but the server says 'show new UI', the user sees a flash of the old UI then switches. Solution: always use cached flags for initial render, and only apply server updates on the NEXT session, not mid-session."

4. Technical Debt

> "Old flags that are 100% rolled out should be removed from the codebase. If you never clean up, you end up with hundreds of dead branches. At PayPal, we have a process where flags are marked for cleanup after 30 days at 100%."

5. Testing with Flags

> "Every flag doubles your test matrix. A feature behind a flag needs tests for both ON and OFF states. In CI, we run tests with flags ON and OFF separately. For QA, we have a debug menu to override any flag."

THE DEBUG MENU (MENTION THIS — IT SHOWS SENIOR THINKING)

swift

```
// In debug/QA builds, add a flag override screen
#if DEBUG
class FeatureFlagDebugView: View {
    @State var overrides: [String: Bool] = [:]

    var body: some View {
        List {
            ForEach(allFlags, id: \.key) { flag in
                Toggle(flag.key, isOn: binding(for: flag.key))
            }
            Button("Reset All Overrides") {
                overrides.removeAll()
            }
        }
        .navigationTitle("Feature Flags")
    }
}
#endif
```

> "In every app I build, I include a debug menu that lets QA and developers override any feature flag locally. This saves massive amounts of time — instead of creating test accounts in different rollout buckets, QA can just toggle the flag on their device."

WISE'S DYNAMIC FORMS = FEATURE FLAGS ON STEROIDS

From Wise's 2022 tech blog:

> "We build our own framework, called Dynamic Forms, that allows us to implement business logic once and then natively render it on devices and web. It is data-driven and declarative and can change from backend without any modification to the mobile application code."

This IS a feature flag system — but instead of just toggling features on/off, the backend sends the entire UI structure. This is crucial for Wise because:

- Regulatory requirements change per country (KYC fields differ)
- They operate in 70+ countries with different rules
- App Store updates take days — server-driven UI is instant

What to say:

> "I read about Wise's Dynamic Forms framework — it's essentially server-driven UI, which is the most powerful form of feature flags. Instead of just toggling a boolean, the backend sends the entire form structure. At PayPal, we have a similar pattern called SPF (Server-driven Product Flow) where the server determines the UI structure, fields, and validation rules. I'm very familiar with this architecture."

MONITORING & METRICS FOR FEATURE FLAGS

When you mention rolling out with flags, also mention what you'd monitor:

Per-Flag Metrics:

Metric	Why	Alert Threshold
Crash-free rate	New code might crash	< 99.5% → auto-disable
API error rate	New feature might hit buggy endpoints	> 2% → alert
Conversion rate	New flow might confuse users	> 10% drop → review
Latency p95	New feature might be slow	> 3 seconds → investigate
Support tickets	Users confused by new UI	> 20% spike → review

A/B Test Metrics:

Metric	Control	Treatment	Significance
Completion rate	72%	78%	$p < 0.05$ ✅
Avg time to complete	45s	38s	$p < 0.01$ ✅
Error rate	1.2%	1.1%	$p = 0.4$ ❌ (not significant)

> "I always define a primary metric (what we're trying to improve), guard metrics (what must NOT get worse), and a minimum sample size before declaring a winner. At PayPal, for BFX rollouts, our primary metric is conversion rate per country, and our guard metric is FX error rate."

HOW YOUR PAYPAL EXPERIENCE MAPS TO WISE

PayPal (Your Experience)	Wise Equivalent	What to Say
ELMO (experiment platform)	Wise's internal experimentation	"I've worked extensively with PayPal's ELMO platform for feature flags and A/B testing"
BFX country-by-country ramp	Wise's country rollouts	"I manage gradual rollouts by country with monitoring at each stage"
RUNE experiment bug (nil treatment on login)	Flag initialization timing	"I debugged a production issue where experiment treatment was nil on fresh login"

PayPal (Your Experience)	Wise Equivalent	What to Say
SPF (server-driven product flow)	**Dynamic Forms**	"PayPal uses SPF for server-driven UI — same concept as Wise's Dynamic Forms"
Express Checkout A/B tests	Transfer flow experiments	"I ran A/B tests on checkout flow, measuring completion rate"
UCP Portal (config deployment)	Remote config	"I deploy config changes via a portal without app updates"

INTERVIEW SCRIPT: HOW TO BRING ALL THIS UP

When they describe a feature and ask about improvements:

> **Step 1 — Acknowledge the improvement:** "Adding scheduled transfers is a great feature. Before I dive into the architecture, let me talk about the rollout strategy, because at PayPal I've learned that HOW you ship is as important as WHAT you ship."
> **Step 2 — Feature flag approach:** "I'd wrap this behind a feature flag — `scheduled_transfers_enabled`. Start with 5% of users, monitor for a week, then ramp up. On the mobile side, it's a simple boolean check that controls whether the schedule option appears in the transfer flow."
> **Step 3 — Connect to your experience:** "This is exactly how I manage BFX rollouts at PayPal. We ramp currency conversion by country — Italy first, then Saudi Arabia, then Lithuania — with monitoring dashboards at each stage. One time we caught a BIN configuration issue in Italy that would have affected Saudi Arabia if we'd launched simultaneously."
> **Step 4 — The ELMO bug story (if they probe deeper):** "I actually debugged a real feature flag issue on RUNE — our experiment treatment was nil on fresh login because the flag fetch was async and hadn't completed before the UI rendered. The fix was loading cached flags synchronously on startup, then refreshing from the server in the background. This taught me that flag initialization timing is the most common mobile-specific pitfall."
> **Step 5 — Dynamic Forms connection:** "For something more complex like changing the transfer form per country, I know Wise uses Dynamic Forms — server-driven UI where the backend defines the form structure. At PayPal we have a similar system called SPF. I'm very comfortable with this architecture."

QUICK REFERENCE: FEATURE FLAG VOCABULARY

Know these terms and use them naturally:

Term	Meaning
Feature flag / toggle	Boolean switch to enable/disable a feature
Experiment / A/B test	Compare control vs treatment(s) with metrics
Treatment / variant	The version of the feature a user sees
Control group	Users who see the existing (unchanged) experience
Rollout percentage	What % of users see the new feature (5% → 100%)
Kill switch	Instant disable of a problematic feature
Remote config	Server-controlled values (not just booleans — numbers, strings, JSON)
Server-driven UI	Backend sends UI structure, mobile renders it (Dynamic Forms)
Guard metric	Metric that must NOT worsen during experiment
Statistical significance	Enough data to trust the result ($p < 0.05$)
Stale flag	Cached flag that no longer matches server state
Flag debt	Accumulated dead flags in codebase that should be removed
Targeting rules	Conditions for who sees a flag (country, version, user segment)
Sticky bucketing	User stays in same variant even across sessions