Take-Home Assignment for Pula Advisors - Senior Mobile App Developer

Summary

- Expected working time: Max 3 hours
- Kotlin (Android) is primary and optional bonus if you add Swift (iOS) component

---

Introduction

You are building the core data layer for a mobile application used by field agents who collect agricultural survey data from farmers in rural areas. The app operates in regions with unreliable or no connectivity, so it must work entirely offline and sync data to the server when a connection becomes available.

Your task: Build a **Survey Response Sync Engine** — the system responsible for storing survey responses locally, queuing them for upload, handling partial failures gracefully, and retrying intelligently.

This is a **data-layer and architecture** exercise. No UI is required.

Domain Context

A field agent uses the app as follows:

1. They select a farmer from a pre-loaded local database
2. They fill in a survey questionnaire — each survey produces a set of answers (question-answer pairs) and may include photo attachments
3. They tap "Submit" which saves the response **locally** (the device may be offline)
4. A background sync process picks up unsubmitted responses and uploads them to the server when connectivity is available
5. The server processes each response individually — some may succeed while others fail in a single sync session
6. Field agents often work in areas where connectivity is intermittent — the network can drop mid-sync

Device and Environment Reality

Field agents operate in rural Sub-Saharan Africa using **low-end Android devices** (typically 1–2 GB RAM, 16–32 GB storage, Android 8–10). Connectivity ranges from no coverage to intermittent 2G/3G. Battery life is critical — agents are in the field for 8–12 hours and may not have access to

charging. Storage fills quickly because surveys include photo attachments. Your architectural decisions should reflect these real-world constraints.

Survey Structure

Surveys are not always flat questionnaires. Some surveys contain **repeating sections** — for example, if a farmer reports having 3 farms, the survey must collect the same set of questions (crop type, area, yield estimate, GPS boundary) for each of the 3 farms. The number of repetitions is determined by a previous answer, so it is not known at survey design time. Your data model should be able to represent this kind of dynamic, nested response structure.

What You Need to Build

Design and implement a sync engine that handles the following real-world scenarios. **How you model the data, structure the code, and handle edge cases is up to you** — we want to see your engineering decisions.

Scenario 1: Offline Storage

A field agent completes 10 surveys throughout the day with no connectivity. All responses must be persisted locally and survive app restarts. When the agent later reaches a town with connectivity, the sync engine should detect there are pending responses and be ready to upload them.

Your solution should address:

- How you model and persist survey response data locally
- How you track which responses have been synced vs. pending
- How media attachments (photos) relate to their parent survey response
- How your data model handles **repeating sections** (e.g., a farmer with 3 farms where the same question set is answered per farm). The number of repetitions is dynamic and driven by a prior answer.
- How you manage **local storage growth** when agents collect 50+ surveys per day with photo attachments on devices with limited storage (16–32 GB total)

Scenario 2: Partial Failure

The sync engine begins uploading 8 pending responses. Responses 1–5 succeed. Response 6 fails due to a server error. Responses 7–8 have not been attempted yet.

Your solution should address:

- The 5 successful responses must not be re-uploaded on the next sync
- The caller must know exactly which responses succeeded and which didn't
- The next sync attempt should only retry the responses that failed or were never attempted

Scenario 3: Network Degradation

The agent is on the edge of cellular coverage. The sync engine starts uploading, but after 3 successful uploads, the 4th fails with a connection timeout. There are still 6 more responses in the queue.

Your solution should address:

- Detecting that the network is likely down (vs. a one-off server error)
- Stopping the sync queue early to **conserve battery and storage** — field agents use low-end devices that must last a full day without charging, and every failed retry wastes both battery and data quota
- Communicating to the caller what happened (what succeeded, why it stopped)

Scenario 4: Concurrent Sync Prevention

The agent opens the app while a background sync is already running. The UI triggers another sync request.

Your solution should address:

- Ensuring only one sync operation runs at a time
- The second caller should not corrupt or duplicate the first sync's work

Scenario 5: Network Error Handling

Various network issues can occur in the field:

- No internet connection at all
- Connection timeout (slow network)
- Server returns an error (400, 500, etc.)
- Unknown/unexpected exceptions

Your solution should address:

- Mapping different failure types into a consistent error model
- Distinguishing between errors that suggest "try again later" vs. "something is wrong with this specific request"

Constraints and Guidelines

- **No real backend needed.** Create fake/mock API services that you can configure to simulate the scenarios above (success, failure on Nth call, timeout errors, etc.)
- **No UI required.** This is a data-layer and architecture exercise.
- **Use Room** (or equivalent) for local persistence. If you use Room, you'll need to handle TypeConverters for non-trivial types.

- **Use Kotlin Coroutines** for async operations.
- **Architecture matters.** We evaluate how you separate concerns, manage dependencies, and structure your packages — not just whether it works.

Testing Requirements

Tests are a **first-class deliverable**, weighted equally to implementation quality.

Write tests that prove your sync engine handles the scenarios described above. At minimum:

- **Sync engine**: All succeed, partial failure, early termination on network issues, empty queue
- **Data layer**: Save/retrieve responses, status tracking
- **Error handling**: Different exception types produce the correct error model

Use whatever testing libraries you're comfortable with (JUnit, MockK, Mockito, Turbine, etc.).

Deliverables

1. A **Git repository** (not a zip) with showing your development process. We want to see how you think, not just the final result. Single-commit submissions is not suggested.
2. All Kotlin source files
3. All test files
4. A `build.gradle` or dependency list showing libraries used
5. An `ARCHITECTURE.md` (400–600 words) answering:

- What architecture did you choose and why? What alternatives did you consider?
- How would you extend this to handle media file uploads where photos must be compressed before uploading?
- Describe a scenario where your network detection logic could make a wrong decision. How would you mitigate it?
- How would your design support **remote troubleshooting** — if a field agent's sync is failing and the support team cannot physically access the device, what data would you log, expose, or report to help diagnose the issue?
- The app is used in agricultural regions where **geospatial data** matters (field boundary mapping, GPS accuracy in rural areas with poor satellite coverage, canopy cover assessment). You are NOT implementing any maps features, but: what technical challenges would you anticipate when adding GPS-based field boundary capture to this app, and how would you validate the accuracy of captured coordinates?
- One thing you would do differently with more time

Bonus (Optional)

Not required within the 3-hour limit, but noted positively:

- **WorkManager integration** for scheduled background sync with network constraints
- **Swift implementation** of the sync engine using async/await and actors
- **Sync progress reporting** so a UI layer could show "Uploading 3 of 8..."
- **Device-aware sync strategy** that adapts behavior based on battery level, available storage, or network type (WiFi vs. metered)

What NOT to Worry About

- UI / Fragments / Activities / Compose
- Full Android project setup (AndroidManifest, resources, running app)
- Real network calls
- Database migrations
- Authentication
- Image compression implementation (discuss in ARCHITECTURE.md only)
- Maps / GPS implementation (discuss in ARCHITECTURE.md only)
- Remote monitoring dashboard or logging infrastructure (discuss in ARCHITECTURE.md only)
- CI/CD configuration

Good luck. We look forward to reviewing your work.