Take home assignment: (SE 1)

Overview

We are excited to see your skills in action! This take-home project is designed to gauge your familiarity with the full stack of mobile development: **Flutter UI**, **Native Platform Interaction (Android/iOS)**, and **Backend Integration**.

We are looking for clean code, logical problem-solving, and the ability to bridge the gap between Flutter and native platform APIs.

Submission Deadline: 1 week or 7 days from receiving this assignment.

Note: This is a self-contained assignment designed to assess your ability to work independently. We will not be able to answer clarifying questions or provide guidance during the assignment period. Part of the evaluation includes how you handle ambiguity and make decisions with incomplete information (see Section 5).

🎯 The Challenge: "Device Vital Monitor"

Your task is to build a simple Flutter application that monitors device sensor data (thermal state, battery level, and memory usage) and logs it to a backend service.

Core Requirements

1. The Backend (API) & Architecture

- **Goal**: Create an API that receives device sensor vitals (thermal, battery, memory) and provides analytics (Rolling Average).
- **Constraint**: The system must be persistent (survive a restart).
- **Constraint**: You must handle "impossible" data values validation for all sensor types.

Your Task:

- Choose a tech stack (.NET or Node.js preferred) and a storage mechanism.

- **Justify your choice**: Why did you pick this database/storage? Why this architecture?
- For this assignment, SQLite, JSON file with file locking, or a simple embedded DB (e.g., LiteDB, LowDB) are all acceptable.

Required Endpoints:

- `POST /api/vitals` - Accept a vital log with structure:

  ```
  {
    "device_id": "string",
    "timestamp": "ISO8601 datetime",
    "thermal_value": "number (0-3)",
    "battery_level": "number (0-100)",
    "memory_usage": "number (0-100)"
  }
  ```

- `GET /api/vitals` - Return historical logs (latest 100 entries)
- `GET /api/vitals/analytics` - Return analytics data (rolling average and other meaningful insights)

Data Validation:

- Reject `thermal_value` outside the range 0-3
- Reject `battery_level` outside the range 0-100
- Reject `memory_usage` outside the range 0-100
- Reject future timestamps
- Reject missing required fields

2. The Native Modules (Android & iOS)

This is the most important part of the test. **Do NOT use existing 3rd-party packages (like `battery_plus`) for retrieving the sensor data.** You must implement **Flutter MethodChannels** to communicate with native code.

Platform Support: You MUST implement at least ONE platform (Android OR iOS) fully. Implementing both is a bonus but not required. Clearly state in your README which platform(s) you targeted.

- **Android (Kotlin/Java)**:
  - Implement method channels to retrieve:
    - **Device Thermal Status**: Use `PowerManager.getCurrentThermalStatus()` (API 29+) or `PowerManager.getThermalHeadroom()` for older versions.

            - Map thermal status: `THERMAL_STATUS_NONE` = 0, `THERMAL_STATUS_LIGHT` = 1, `THERMAL_STATUS_MODERATE` = 2, `THERMAL_STATUS_SEVERE` = 3 (and higher)
        - **Battery Level**: Use `BatteryManager.BATTERY_PROPERTY_CAPACITY` to get percentage (0-100)
        - **Memory Usage**: Use `ActivityManager.MemoryInfo` to calculate used memory percentage
    - Handle devices that don't support these APIs (return sensible defaults or errors).
- **iOS (Swift/Objective-C)**:
    - Implement method channels to retrieve:
        - **Device Thermal State**: Use `ProcessInfo.processInfo.thermalState`
            - Map thermal state: `nominal` = 0, `fair` = 1, `serious` = 2, `critical` = 3
        - **Battery Level**: Use `UIDevice.current.batteryLevel` (0.0-1.0, convert to 0-100)
        - **Memory Usage**: Use `mach_task_basic_info` to calculate used memory percentage

3. The Flutter Application

Build a clean, user-friendly app with the following features:

- **Dashboard Screen**:
    - Display all current sensor readings in real-time (or refreshable):
        - **Thermal State** (numeric value and human-readable label)
        - **Battery Level** (percentage)
        - **Memory Usage** (percentage)
    - A "Log Status" button that sends all current sensor data to your Backend API.
    - Show visual feedback on successful log or error state.
- **History Screen**:
    - Fetch and display the list of historical logs from `GET /api/vitals`.
    - Display the data in a user-friendly way that helps users understand device health trends.
    - Show analytics data from the analytics endpoint in a meaningful way.
- **Architecture**:
    - Use a state management solution of your choice (Provider, Riverpod, Bloc, etc.).
    - Keep your code organized and testable.
    - Separate business logic from UI (Repository pattern recommended).
- **Error Handling**:
    - Show a visible error (Snackbar/Dialog) if the Backend is unreachable.
    - Show a visible error if Native Platform throws a `PlatformException`.
    - Handle network timeouts gracefully.
    - Think about other edge cases and handle them appropriately.

4. Unit Tests

You must include basic unit tests covering:

- **Backend**: Rolling average calculation logic
- **Backend**: Data validation (rejecting invalid sensor values for thermal, battery, and memory)
- **Flutter**: Repository/Service layer logic

Writing widget or integration tests is optional.

5. Handling Ambiguity & Design Decisions

Real-world engineering often involves making decisions when requirements are incomplete or unclear. We want to see how you handle ambiguity.

Please include a `DECISIONS.md` file documenting:

Required Sections:

1. **Ambiguities Identified**:
   - List at least 3 aspects of this assignment where the requirements were vague or open to interpretation
   - Examples: "What happens when a sensor temporarily fails?"
2. **Your Design Decisions**:
   - For each ambiguity, document:
     - The question/ambiguity you identified
     - The options you considered (minimum 2 alternatives)
     - Your final decision and reasoning
     - Trade-offs you accepted
3. **Assumptions Made**:
   - List key assumptions you made about user needs, system behavior, or technical constraints
   - Explain why each assumption is reasonable
4. **Questions You Would Ask**:
   - If this were a real project with a product manager available, what clarifying questions would you ask?
   - Demonstrate that you know when to seek clarification vs. when to make an independent decision

- Note: For this assignment, no clarification is available - you must make decisions independently

Format Example:

```
## Ambiguity 1: Analytics Endpoint Response Format

**Question**: What specific analytics should be returned beyond rolling average?

**Options Considered**:
- Option A: Only rolling average (simple, meets literal requirement)
- Option B: Multiple metrics (min, max, average, trend direction)
- Option C: Time-windowed analytics (last hour, day, week)

**Decision**: I chose Option B because...

**Trade-offs**: More complex response, but provides better insights...

**Assumptions**: Users want to understand device health trends over time...
```

Evaluation:

We will assess:

- **Critical thinking**: Can you identify unstated requirements?
- **Decision-making**: Do you make reasonable choices with sound justification?
- **Communication**: Can you articulate trade-offs clearly?
- **Product sense**: Do your decisions align with likely user needs?

Note: There are no "correct" answers to these ambiguities. We're evaluating your thought process, not whether you picked the same option we would have.

6. AI Collaboration Story

Modern engineering involves knowing *how* to use tools like Copilot, ChatGPT, or Claude effectively. We want to see your "Engineer + AI" workflow.

Please include an `ai_log.md` file covering:

- **The Prompts**: Share 2-3 specific prompts you used (copy-paste verbatim).
- **The Wins**: Which task did AI accelerate? Include before/after context.
- **The Failures**: Share one example where AI output was wrong and how you debugged it.
- **The Understanding**: Pick one AI-generated code block and explain it line-by-line in your own words.

Format Example:

```
---
**Prompt**: "Generate a Flutter MethodChannel to call native Android PowerManager"

**Result**:
[code snippet]

**My Changes**: I had to fix the return type from `Int` to `Double` because...

**Why it works**: The MethodChannel uses platform-specific handlers...
---
```

What if I didn't use AI?

That is perfectly fine! In your `ai_log.md`, simply state: **"No AI usage."**

However, please add a brief explanation on *why* (e.g., "I know these APIs by heart," or "I wanted to challenge my raw knowledge").

✅ Minimum Passing Criteria

To be considered for the next round, your submission **must**:

- [ ] Successfully retrieve all sensor data (thermal, battery, memory) on at least ONE platform via MethodChannel
- [ ] Backend API accepts POST requests and stores data persistently
- [ ] Backend survives restart (data is still there)
- [ ] Frontend sends data to backend and displays History screen
- [ ] Include README with setup instructions that work on first try
- [ ] Handle at least one error case gracefully (e.g., API down, platform unavailable)
- [ ] Include `ai_log.md` (even if you didn't use AI)

- [ ] Include `DECISIONS.md` documenting design decisions and handling of ambiguities
- [ ] Include unit tests for backend logic

Auto-Reject Criteria (We will not review further if):

- Setup instructions don't work without much effort from our end
- Used 3rd-party plugin for sensor data (explicitly forbidden)
- No MethodChannel implementation found
- Backend doesn't persist data
- No `ai_log.md` file
- No `DECISIONS.md` file

📝 Evaluation Criteria

We are not looking for a production-ready super-app, but we *are* looking for:

1. **Native Knowledge**: Correct implementation of Platform Channels (MethodChannel) on at least one platform.
2. **Code Quality**: Clean, readable code with proper separation of concerns.
3. **Error Handling**: What happens if the API is down? What if the native sensor is unavailable?
4. **Testing**: Basic unit tests that actually test business logic.
5. **Documentation**: A `README.md` explaining how to run your backend and app.
6. **AI Collaboration**: Transparency and ownership in your `ai_log.md` .
7. **Handling Ambiguity**: Quality of decision-making and reasoning in `DECISIONS.md` . We value candidates who can identify unclear requirements, consider alternatives, and make justified decisions.

🚀 Bonus Points (Pick ONE if you have time)

These are truly optional. Completing the core requirements well is better than half-implementing bonuses.

- **Auto-Logging**: Implement a background service or timer to log vitals every 15 minutes (WorkManager for Android, Background Fetch for iOS).

- **Offline Support**: Save logs locally (SQLite/Hive) if the backend is unreachable, and sync when online.
- **Both Platforms**: Implement MethodChannel for both Android AND iOS.
- **Advanced UI**: Charts/graphs showing thermal trends over time.

📦 Submission

Please push your code (Flutter App + Backend) to a **public GitHub/GitLab repository** and share the link with the recruiter.

Repository Naming Requirement:

Your repository MUST be named using the SHA256 hash of your email address.

How to generate your repository name:

1. Take your email address (the one you used to apply)
2. Generate the SHA256 hash of your email address
3. Use the **first 12 characters** of the hash as your repository name

Example:

If your email is `john.doe@example.com` :

```
# On macOS/Linux:
echo -n "john.doe@example.com" | shasum -a 256

# On Windows (PowerShell):
echo -n "john.doe@example.com" | openssl dgst -sha256

# Output: a1b2c3d4e5f67890abcdef1234567890...  (64 characters)
# Take first 12 characters: a1b2c3d4e5f6
```

Your repository name would be: **`a1b2c3d4e5f6`**

Important notes:

- Use the **exact email address** you used in your application (correct capitalization, no extra spaces)

- The same email will always produce the same hash, ensuring consistency

Your repository must include:

- `README.md` with setup instructions
- `ai_log.md` with your AI collaboration story
- `DECISIONS.md` documenting your design decisions and how you handled ambiguities
- Source code for Flutter app
- Source code for Backend API
- Any tests you wrote

Submission Checklist:

- [ ] Repository is public and accessible
- [ ] README has clear setup steps (dependencies, how to run backend, how to run app)
- [ ] Code runs on first try following README instructions
- [ ] `ai_log.md` is present
- [ ] `DECISIONS.md` is present with thoughtful analysis
- [ ] At least one platform (Android OR iOS) works completely

Good luck! We look forward to reviewing your work.