Biometric Attendance Module – Mobile App + Backend (Detailed Design)

1. High-level Idea

The **Biometric Attendance Module** combines **geofencing** and **face recognition** to ensure that attendance is marked only when:

1. The user is **physically inside a 100 m radius** of the hospital/office (geofence), and
2. The user's **face matches** the face enrolled for that account (biometric).

Both conditions **must** pass. If either fails (outside location or face mismatch), the attendance request is **rejected**.

This dual verification:

- Prevents proxy/buddy punching (one person marking attendance for another).
- Ensures the person is actually at the designated workplace.
- Automatically records attendance with accurate time and location.

---

2. Frontend (Expo Mobile App) – Detailed Workflow

2.1 Login and Token Management

Goal: Establish a secure session and identify the user for subsequent API calls.

Flow:

1. The user opens the mobile app and logs in using:
    - Email + password, or
    - Staff ID + password (as per your HIMS design).
2. The app sends these credentials to the backend's **login API**.
3. On successful authentication, the backend returns:
    - An **auth token** (e.g., JWT or Laravel Sanctum token).
    - Optional: user profile, permissions, role, etc.
4. The app securely stores the token:
    - Preferably in **SecureStore** (or similar secure storage).
    - Fallback: AsyncStorage if needed (with care).
5. For all future requests (check-in/check-out), the app:
    - Adds an `Authorization: Bearer <token>` header to identify the user.

Why this matters:
The token links all attendance actions to a specific authenticated user and prevents anonymous or spoofed requests.

2.2 User Presses "Mark Attendance"

When the user taps the **"Check-In"** button on the app, the frontend triggers a **multi-step flow**:

Step 1: Get GPS Location

Goal: Determine where the user is at the moment of check-in.

1. The app uses **Expo Location** (or similar) to:
    - Request **location permission** (if not already granted).
    - Read the current **latitude** and **longitude**.
2. Once coordinates are obtained (e.g., `user_lat`, `user_lng`), the app sends a request to the backend like:
    - `POST /api/attendance/pre-check`
    - Headers: `Authorization: Bearer <token>`
    - Body: `{ lat: user_lat, lng: user_lng }`
3. At this point:
    - **No face scan** is done yet.
    - This avoids unnecessary biometric prompts if the user is obviously outside the allowed area.

Step 2: Backend Geofence Validation (100 m Rule)

Goal: Check if the user is within the allowed 100 m radius of the hospital/office.

On the backend, for the given site (e.g., the user's assigned hospital/branch):

- The system has stored:
    - `center_lat`
    - `center_lng`
    - `radius` = 100 m (configurable but default 100 m)

The backend:

1. Receives `user_lat`, `user_lng` from the app.
2. Loads the geofence data (`center_lat`, `center_lng`, `radius`).
3. Uses a distance formula, usually **Haversine**, to calculate the distance between:
    - The geofence center, and
    - The user's current location.
4. Compares the result:
    - If `distance > 100 m`:

        - Responds with something like:

```
{
  "status": "geo_failed",
  "message": "You are outside the allowed area"
}
```

        - The attendance flow **stops here** on the frontend (no face scan UI is opened).
    - If `distance <= 100 m`:
        - Responds with:

```
{
  "status": "geo_ok",
  "message": "Proceed to face verification"
}
```

The frontend then proceeds to the biometric step only if `status = "geo_ok"`.

Step 3: Trigger Face Recognition on Success

Once the backend confirms the user is inside the geofence:

1. The app receives `geo_ok`.
2. The app opens:
    - Either the **device biometric prompt** (Face ID / Android face lock) using something like `LocalAuthentication`, or
    - A **camera screen** to capture a face image (if using custom server-side face recognition).
3. The UI might show an instruction:
    - "Align your face in the frame and tap capture."
    - Or the OS-native Face ID popup: "Confirm your identity to continue."

2.3 Face Capture and Verification Patterns

There are **two main implementation patterns**.

Pattern A: Device Biometric (LocalAuthentication / Face ID / Android Face Lock)

- The app calls the OS-level biometric API.
- The mobile OS:
    - Compares the live face with the **face stored on the device** as part of device security.
    - Returns only a success/failure result to the app.
    - Does **not** give raw face images or templates to the app for privacy.

Flow:

1. App invokes device biometric prompt.
2. User's face is scanned and verified **locally** on the device.
3. If verification **fails**:
    - App shows an error: "Biometric authentication failed."
    - Attendance is not attempted.
4. If verification **succeeds**:
    - App calls backend endpoint, e.g., `POST /api/attendance/check-in`:
        - Headers: `Authorization: Bearer <token>`
        - Body: `{ biometric_success: true, lat, lng }` (and maybe metadata like device info).
    - Backend trusts that:
        - The OS has authenticated the device user,
        - And the token identifies the employee.

Pros:

- Simple to implement using Expo/React Native APIs.
- No need to store and manage face templates on server.
- Strong device-level security.

Cons:

- Tied to device's enrolled face.
- If the user logs in on another device, they must enroll that device's biometrics.

Pattern B: Custom Face Recognition (Server-Side)

Here, you implement your own face recognition module on the backend.

Enrollment (one-time per user):

1. During registration or profile setup:
    - The user stands in good light and captures a clear face image via the app camera.
2. The app sends this image to the backend.
3. The backend:
    - Detects the face, extracts a **face embedding** (vector representation).
    - Stores this embedding securely as `face_template_reference` linked to the user in the DB.

Verification (every attendance attempt):

1. After geofence is confirmed, the app:
    - Opens camera, captures a live face image.
    - Sends the image (or compressed/base64) to the backend.
2. The backend:
    - Extracts an embedding from the live image.

- Compares it with the stored embedding for that user.
- Computes a similarity/distance score.
- If the similarity is **above a threshold**, it is considered a match.
- If below threshold → considered mismatch or another person.

3. Backend responds:

   - On match:

     ```
     {
       "status": "face_ok",
       "message": "Face recognized successfully"
     }
     ```

   - On mismatch:

     ```
     {
       "status": "face_failed",
       "message": "Face did not match the enrolled profile"
     }
     ```

Pros:

- Not tied to a specific device.
- Centralized face template; user can use multiple devices.

Cons:

- Requires a more complex backend (ML models, image handling).
- Needs strong security and privacy controls for biometric data.

2.4 Attendance Decision on the Frontend

After both conditions are satisfied:

- **Geofence valid** and
- **Face verification success**

The backend creates the attendance record and returns an overall success response to the app, containing:

- Date and time of check-in.
- Location or branch name.
- Any additional info (e.g., shift, remarks).

The app displays a confirmation like:

> "Checked-in successfully at 09:05 AM."

If either geofence or face fails, the app displays an appropriate error message and does **not** mark attendance.

2.5 Check-out Flow

Check-out is very similar to check-in:

1. User taps **"Check-Out"**.
2. App repeats the **location check**:
    - Optional: require geofence again to make sure they are leaving from the workplace.
3. You can choose to:
    - Require **face verification again**, or
    - Only rely on **token + location** if you want a lighter UX.
4. Backend:
    - Fetches that user's **attendance row for today**.
    - Updates it with `checkout_time`, and optionally `checkout_lat`, `checkout_lng`.
    - Calculates total hours if needed.

---

3. Backend – Core Logic (Laravel Example)

3.1 Data Model / Tables

You will typically need at least these tables:

1. **users**
    - `id`
    - `name`
    - `email` / `staff_id`
    - `role`
    - `face_template_reference` (nullable if device biometrics are used instead)
    - Other fields as per your HIMS
2. **geofences / locations**
    - `id`
    - `name` (e.g., "Main Hospital", "Branch A")
    - `center_lat`
    - `center_lng`
    - `radius` (default 100 m, but configurable per site)
    - `status` (active/inactive)
3. **attendances**
    - `id`
    - `user_id`
    - `date`
    - `checkin_time`
    - `checkout_time`

- `checkin_lat`
- `checkin_lng`
- `checkout_lat` (optional)
- `checkout_lng` (optional)
- `status` (`present`, `late`, `half-day`, etc.)
- `method` (`face+geo`, `manual`, etc.)
- `remarks` (optional)

4. **audit_logs**

- `id`
- `user_id` (nullable if not authenticated)
- `event_type` (e.g., `geo_failed`, `face_failed`)
- `details` (JSON or text: coordinates, error reason, etc.)
- `created_at`

These tables give you traceability and reporting capability.

3.2 Geofence Validation Logic

Input from app: `(user_lat, user_lng)` and implicit `user_id` via token.

Backend steps:

1. Identify which **geofence** applies to the user:
   - E.g., use the user's assigned location, or a general hospital geofence.
2. Load:
   - `center_lat`, `center_lng`, `radius`.
3. Compute the **distance** between `(center_lat, center_lng)` and `(user_lat, user_lng)` using a formula like **Haversine**.
4. If `distance <= radius`:
   - Return success (`geo_ok`).
5. Else:
   - Log to `audit_logs` with `event_type = geo_failed`.
   - Return `geo_failed` response to frontend.

This ensures only users **within 100 m** can move to the next step.

3.3 Face Recognition Logic

Enrollment (One-time Setup)

1. The frontend provides a face image during registration.

2. Backend:
    - Processes the image to extract facial features (embedding).
    - Saves the embedding (or a reference to it) as `face_template_reference` for that user.
3. Optionally store:
    - Quality metrics.
    - Enrollment date/time.

Verification (Every Attendance)

For each attendance attempt:

1. The backend receives either:
    - A flag `biometric_success: true` (for device biometric mode), or
    - A fresh image to be compared (for custom recognition).
2. If using server-side comparison:
    - Extract new embedding from the live image.
    - Compare with stored embedding for that `user_id`.
    - If similarity is above a defined threshold:
        - Consider it a match.
    - Otherwise:
        - Log as `face_failed` in `audit_logs`.
        - Return a failure response.
3. If using device biometrics:
    - Just check that `biometric_success` is true and that:
        - The request is authenticated via a valid token.
        - Optionally verify device ID / other context if needed.

3.4 Combined Decision Engine (End-to-End Flow)

For every **Mark Attendance** request, the backend applies a **logical sequence**:

1. **Token Verification**
    - Validate the auth token (JWT/Sanctum).
    - Identify `user_id`.
    - If invalid token → reject.
2. **User State Checks**
    - Confirm user is active (not blocked).
    - Optionally check shift timing, role, or other business rules.
3. **Geofence Check**
    - Run geofence validation using the last provided location.
    - If it fails:

        - Log `geo_failed`.
        - Return immediately with an error (no face check).

4. **Face Recognition Check**
    - If geofence passes:
        - Perform face recognition based on the chosen pattern.
    - If face fails:
        - Log `face_failed`.
        - Return error.

5. **Attendance Create/Update**
    - If both geofence and face checks pass:
        - If this is a **check-in**:
            - Create a new attendance row for today if none exists.
            - Set `checkin_time`, `checkin_lat`, `checkin_lng`, `status = present`, `method = face+geo`.
        - If this is a **check-out**:
            - Find today's attendance row for the user.
            - Update `checkout_time`, `checkout_lat`, `checkout_lng`.

6. **Response to App**
    - Return a structured success response:
        - `status: success`
        - `message: "Checked-in successfully"`
        - `data: { date, time, location_name, ... }`

---

4. Why Dual Verification is Used

1. **Prevents Buddy Punching**
    - A colleague cannot mark attendance on behalf of someone else.
    - They would need:
        - The person's face, and
        - To be physically at the location.

2. **Ensures Physical Presence**
    - Geofencing confirms the person is actually at/near the hospital/office, not at home or elsewhere.

3. **Automates Accurate Records**
    - Each attendance entry includes:
        - Time, date, user, location coordinates, and method.
    - This reduces manual entry, errors, and disputes.

4. **Improves Security and Auditability**
    - `audit_logs` make it easy to review failed attempts and suspicious behavior.

5. Mapping to Your Stack (Expo + Laravel)

5.1 Expo Mobile App

The Expo app will:

- Use **Location APIs** to:
    - Request location permissions.
    - Fetch current GPS coordinates.
- Use **Biometric APIs / Camera** to:
    - Trigger OS-level Face ID / face lock (device biometrics), or
    - Capture an image of the user's face for server-side recognition.
- Communicate with the backend over **HTTPS** using JSON APIs:
    - `POST /login`
    - `POST /attendance/pre-check` (for geofence validation)
    - `POST /attendance/check-in`
    - `POST /attendance/check-out`
- Store and send the **auth token** for all protected routes.

5.2 Laravel Backend

The Laravel backend will:

- Handle **authentication**:
    - Login, token generation and validation.
- Manage **geofence configuration**:
    - Admin UI/APIs to set `center_lat`, `center_lng`, and `radius` (e.g., 100 m) for each hospital/branch.
- Implement **geofence validation**:
    - Given `user_lat` and `user_lng`, determine if the user is inside the allowed zone.
- Implement or integrate **face recognition logic**:
    - Either trust device biometrics or process images server-side.
- Manage **attendance records**:
    - Create / update entries in `attendances` table.
    - Link attendance to user, time, and location.
- Provide **admin APIs** for:

    - Viewing attendance logs.
    - Filtering by date, user, department, etc.
    - Viewing geofence failures, face failures (from `audit_logs`).

---

6. How You Can Use This .md

This Markdown content can serve as:

- A **module design document** for your HIMS project.
- The base for:
    - SRS/SDD sections for “Biometric Attendance Module”.
    - Implementation guide for frontend and backend developers.
    - Explanation to faculty/clients on how the system ensures secure and accurate attendance.