ZKTeco SenseFace 2A — Full Integration (Node.js + Express + MongoDB)

> A complete, production-ready system design, code samples, database schema, deployment instructions, security and troubleshooting steps to integrate ZKTeco SenseFace 2A devices (ADMS / PUSH) with a Node.js + Express backend and MongoDB.

Table of contents

1. Overview & Goals
2. High-level architecture
3. Components and responsibilities
4. MongoDB schemas (Mongoose models)
5. REST API — endpoints (device, admin, webhooks)
6. Full Node.js + Express example (production-ready)
7. Handling device commands (pushing commands to device)
8. Web UI ideas and endpoints
9. Security, validation and hardening
10. Deployment (Docker, Nginx, SSL, PM2)
11. Scaling & reliability
12. Testing & troubleshooting
13. Appendix: curl examples, sample logs, links

---

1. Overview & Goals

- Accept PUSH (ADMS) events from ZKTeco SenseFace 2A devices.
- Store attendance logs in MongoDB and optionally forward them (webhooks) to other systems.
- Provide admin APIs to manage devices and users (enroll users, upload faces/templates if needed).
- Provide a mechanism to send commands back to devices (restart, sync time, firmware, or user sync) via command queue (the device polls `/iclock/getrequest`).
- Secure the endpoints and make the system ready for production (TLS, WAF, rate-limiting).

---

2. High-level architecture

```
+------------+       HTTP PUSH       +---------------+    +-----------+
| ZKTeco     | --------------------> | Node/Express  | -> | MongoDB   |
| Device(s)  |  (POST /iclock/cdata) | Backend       |    +-----------+
+------------+                       | - endpoints   |
                                     | - validation  |    +------------+
                                     | - queue        | -> | Worker(s)  |
                                     +---------------+    +------------+
                                            |
                                            | webhooks / events
                                            v
                                      3rd party systems
```

Components: - **Device(s)**: ZKTeco SenseFace 2A sending ADMS/PUSH HTTP requests. - **Node/Express Backend**: REST endpoints handling device pings, logs, admin, commands. - **MongoDB**: store devices, users, attendance, commands. - **Worker / Queue**: handle heavy tasks (image processing, forwarding webhooks) — Redis or RabbitMQ. - **Reverse Proxy (Nginx)**: TLS termination, basic auth pass-through, load-balancing.

---

3. Components and responsibilities

Device - Pushes logs to your server: `POST /iclock/cdata`. - Polls for commands: `GET /iclock/getrequest`. - Ping test: `GET /iclock/ping`.

Express App - Device endpoints: `/iclock/ping`, `/iclock/getrequest`, `/iclock/cdata`. - Admin endpoints: register device, add users, enroll faces/cards, list logs. - Webhooks: forward logs to configured external URLs. - Command queue: store commands for devices; `/iclock/getrequest` returns queued commands.

Database - Collections: `devices`, `users`, `attendance`, `commands`, `webhook_logs`, `audit_logs`.

Worker - Consume `commands` and `webhook` jobs and process them asynchronously.

---

4. MongoDB Schemas (Mongoose)

Device model

```
const DeviceSchema = new mongoose.Schema({
  sn: { type: String, required: true, unique: true },
  name: String,
  ip: String,
  port: Number,
  lastSeen: Date,
  public: { type: Boolean, default: false },
  metadata: mongoose.Schema.Types.Mixed,
  createdAt: { type: Date, default: Date.now }
});
```

User model

```
const UserSchema = new mongoose.Schema({
  empId: { type: String, required: true, unique: true },
  name: String,
  cardNo: String,
  faceTemplateId: String, // optional reference when using templates
  devices: [String],
  meta: mongoose.Schema.Types.Mixed,
  createdAt: { type: Date, default: Date.now }
});
```

Attendance model

```
const AttendanceSchema = new mongoose.Schema({
  deviceSN: String,
  empId: String,
  raw: mongoose.Schema.Types.Mixed,
  time: Date,
  status: String,
  verify: String,
  createdAt: { type: Date, default: Date.now }
});
```

Command model (for pushing commands to device)

```
const CommandSchema = new mongoose.Schema({
  deviceSN: String,
  command: String,
  args: mongoose.Schema.Types.Mixed,
  createdAt: { type: Date, default: Date.now },
  processed: { type: Boolean, default: false },
  processedAt: Date
});
```

---

5. REST API — endpoints

Device-facing (required by ZKTeco)

- `GET /iclock/ping` — Reply `OK`.
- `GET /iclock/getrequest?SN=<sn>` — Reply `OK` or a specific command format (we will respond with `OK` and commands if queued).

- `POST /iclock/cdata` — Receive attendance JSON payload.
- `POST /iclock/updateuser` or similar — If devices push user data (not always used).

Admin / App

- `POST /api/devices` — register device (sn, name, ip)
- `GET /api/devices` — list devices
- `POST /api/users` — create user
- `GET /api/attendance` — query logs
- `POST /api/devices/:sn/command` — queue a command to device
- `POST /api/webhooks` — register webhook target for events

---

6. Full Node.js + Express example (production-ready)

> This example uses modern ES modules, Mongoose, and a small service layer. It demonstrates handling device pushes, validating device SN, saving attendance, queuing webhooks, and supporting commands.

```
// index.js
import express from 'express';
import mongoose from 'mongoose';
import bodyParser from 'body-parser';
import Device from './models/device.js';
import Attendance from './models/attendance.js';
import Command from './models/command.js';

const app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// Simple health/ping
app.get('/iclock/ping', (req, res) => res.send('OK'));

// Device asks for commands
app.get('/iclock/getrequest', async (req, res) => {
  try {
    const sn = req.query.SN || req.query.sn || req.get('SN');
    if (!sn) return res.status(400).send('ERR');

    // update lastSeen
    await Device.findOneAndUpdate({ sn }, { $set: { lastSeen: new Date() } });

    // find first unprocessed command
    const cmd = await Command.findOneAndUpdate(
      { deviceSN: sn, processed: false },
      { $set: { processed: true, processedAt: new Date() } }
    );

    if (!cmd) return res.send('OK');

    // format the command for device (ZKTeco expects XML-like or simple OK?)
    // Minimal: reply OK and device will read commands via other mechanism.
    // If device expects a specific command syntax, send according to spec.
    return res.send(`OK`);
  } catch (err) {
    console.error(err);
    return res.status(500).send('ERR');
  }
});

// Attendance push
app.post('/iclock/cdata', async (req, res) => {
  try {
    // req.body can come as form-data or raw JSON depending on device firmware
    const payload = Object.keys(req.body).length ? req.body : {};

    // ZKTeco sometimes sends 'table: ATTLOG' and 'SN' and nested fields
    const sn = payload.SN || payload.sn || req.query.SN;
    if (!sn) return res.status(400).send('ERR');

    // Basic validation: table must be ATTLOG or CHECKIN
    if (payload.table && payload.table !== 'ATTLOG' && payload.table !==
'CHECKIN') {
      // still accept, but log
    }

    const record = {
      deviceSN: sn,
      empId: payload.CardNo || payload.EnrollNumber || payload.UserID ||
payload.User,
      raw: payload,
      time: payload.Time ? new Date(payload.Time) : new Date(),
      status: payload.Status || null,
      verify: payload.Verify || null
    };

    const att = await Attendance.create(record);

    // TODO: enqueue webhook job

    return res.send('OK');
  } catch (err) {
    console.error('cdata err', err);
    return res.status(500).send('ERR');
  }
});

// Start server
await mongoose.connect(process.env.MONGO_URI || 'mongodb://mongo:27017/zkteco');
const port = process.env.PORT || 8090;
app.listen(port, () => console.log(`Listening ${port}`));
```

> Notes: - ZKTeco devices sometimes send `application/x-www-form-urlencoded` or `text/plain`. The body parser above handles common cases. If your device sends raw body, capture `req.rawBody` as well.

---

7. Handling device commands (push back to device)

Flow 1. Admin queues a command (via `POST /api/devices/:sn/command`), e.g. `REBOOT`, `SYNC_TIME`, `GET_USER`, `DOWNLOAD_USER` 2. You save a `commands` document (processed=false). 3. When device calls `GET /iclock/getrequest`, your server returns `OK` plus a way for device to fetch commands (or you can return commands directly if the device supports it).

Command representation example

```
{
  "type": "RESTART"
}
```

Important: ZKTeco devices' command support depends on firmware. Many deployments poll `getrequest` for server responses and then the server must return commands in a specific format (older devices used XML). For SenseFace, minimal approach is queueing and using ZKTeco SDK or ZKBio API to send templates/users.

---

8. Web UI ideas and endpoints

- Dashboard: device status, lastSeen, online/offline
- Attendance viewer: filter by employee, device, date range
- Device management: register SNs, configure IP, port
- User management: import CSV, enroll card numbers, push templates
- Webhook management: configure external URLs per tenant

---

9. Security, validation and hardening

1. **Use TLS**: Put Nginx in front and use LetsEncrypt. Devices commonly support HTTP; place Nginx to accept HTTPS from clients and proxy to internal service.
2. **Whitelist device IPs**: If known static IPs exist, restrict to them in Nginx or firewall.
3. **Validate device SN**: Keep a `devices` collection; only accept `cdata` from registered `SN` values.
4. **Rate limit**: Protect endpoints from floods (express-rate-limit).
5. **Auth for admin APIs**: JWT + RBAC.
6. **CSRF**: Not relevant for device endpoints but for admin UI use standard CSRF protections.
7. **Log everything**: store raw payloads for audit and debugging.
8. **Harden body parsing**: accept only expected sizes and content-type.

---

10. Deployment (Docker + Nginx + PM2)

docker-compose.yml (minimal)

```
version: '3.8'
services:
  app:
    build: .
    restart: always
    environment:
      - MONGO_URI=mongodb://mongo:27017/zkteco
    ports:
      - '8090:8090'
    depends_on:
      - mongo
  mongo:
    image: mongo:6
    volumes:
      - mongo-data:/data/db
volumes:
  mongo-data:
```

Nginx: Terminate TLS and forward `/iclock/*` to `http://app:8090`.

Let's Encrypt: Use certbot + Nginx for certificates.

PM2: Use PM2 in container or systemd on bare VM.

---

11. Scaling & reliability

- Use a message queue (Redis streams, RabbitMQ, or BullMQ) to enqueue webhook forwarding and heavy tasks.
- Use multiple backend replicas behind load balancer (Kubernetes/ECS).
- Stateless app: store sessions in Redis if used.
- Use TTL indexes on attendance if you have retention policy.

---

12. Testing & troubleshooting

- Use `ngrok` when testing devices behind NAT. Configure device server IP to ngrok forwarding URL.
- Example logs to look for:
- `GET /iclock/ping` — device checks connectivity
- `GET /iclock/getrequest?SN=XXX` — device asking for command
- `POST /iclock/cdata` — attendance record
- Common issues:
- **Connection refused**: port blocked by firewall or app not listening on public interface
- **Wrong payload format**: some firmware versions send urlencoded body — log raw body
- **Device time mismatch**: sync device time with server (send command or set in device UI)

---

13. Appendix — useful curl examples & tests

Ping test

```
curl https://YOUR_DOMAIN/iclock/ping
# should return: OK
```

Simulate cdata POST

```
curl -X POST https://YOUR_DOMAIN/iclock/cdata
  -H 'Content-Type: application/x-www-form-urlencoded'
  --data
'SN=123456&table=ATTLOG&CardNo=EMP100&Time=2025-12-03+12:00:00&Status=0&Verify=1'
```

Queue a command (admin)

```
curl -X POST -H 'Authorization: Bearer <token>' -H 'Content-Type: application/
json'
  https://YOUR_DOMAIN/api/devices/123456/command -d '{"command":"RESTART"}'
```

Final notes & next steps

- I included a working Node/Express skeleton and Mongoose schemas that you can drop into a repo.
- If you want, I can:
- generate a full Git repository (Express app + Mongoose models + Dockerfile + docker-compose)
- provide a React admin UI single-file (create with canvas/react)
- produce a postman collection for testing

Tell me which of the above you want next and I will generate the code/repo for you.