📘 System Documentation: Flutter App + Raspberry Pi (BLE) + NestJS Backend

A Complete End-to-End Architecture Overview
Author: *Mohamed Eliwa*

---

1. System Overview

This system is designed to collect sensor readings from an embedded Raspberry Pi device using Bluetooth Low Energy (BLE), store them on a NestJS backend (with a database), and finally display and manage them through a Flutter mobile application.

The system has **three interconnected components**, each with a specific role:

1. **Raspberry Pi (Edge Device)**
    - Reads sensor data (e.g., DHT, ECG, temperature, custom signals, etc.).
    - Sends the data over BLE to a mobile device.
    - Acts as a peripheral, advertising a custom BLE service.
2. **Flutter Application (Mobile Frontend)**
    - Connects to Raspberry Pi via BLE.
    - Reads the live sensor values.
    - Sends these values to the NestJS backend through REST API calls.
    - Displays real-time sensor data history, charts, logs, and device status.
3. **NestJS Backend (Server + Database)**
    - Provides REST endpoints to receive and store sensor values.
    - Stores records in a database (SQL or NoSQL).
    - Allows retrieving current and historical sensor data.
    - Enables future expansion (alerts, analytics, dashboards).

Together, the system implements a full IoT + Mobile + Cloud architecture.

---

2. Raspberry Pi BLE System (Documentation)

2.1 Hardware Responsibilities

The Raspberry Pi acts as the data source and BLE peripheral.

Functions include:

- Reading data from sensors (e.g., DHT22 temperature/humidity).
- Running a BLE advertisement service.
- Packaging the reading into BLE characteristics.
- Responding to read requests from the Flutter app.
- Maintaining low-latency communication for live monitoring.

2.2 BLE Design

The Raspberry Pi exposes a BLE **service** and **characteristic**:

BLE Service

- **Service UUID**: custom UUID for the IoT system
- Responsible for exposing sensor readings.

BLE Characteristic

- **Characteristic UUID**: custom
- Properties:
    - Read
    - Notify (if implemented)

Flutter reads this characteristic to retrieve the latest values.

2.3 Data Format

The Pi sends small JSON-like or comma-separated string values such as:

temp:24.7,hum:48

or

{"temp": 24.7, "humidity": 48}

Flutter parses these values before forwarding them to the backend.

2.4 Raspberry Pi Software Responsibilities

1. Initialize BLE service.
2. Continuously read sensors.
3. Update characteristic values.
4. Manage BLE advertising.
5. Handle multiple connection attempts gracefully.
6. Avoid blocking BLE loop — sensor readings must be buffered.

2.5 Future Extensibility

The Pi side supports expansion to:

- Multiple sensors
- OTA firmware updates
- Maintaining a local caching database
- Automatic retries on BLE disconnection
- Pairing and whitelisting devices

---

3. Flutter Application (Documentation)

3.1 Architecture Overview

Flutter acts as a **central hub**, responsible for:

- Discovering the Raspberry Pi BLE device
- Connecting to it
- Subscribing to sensor characteristic updates
- Parsing and formatting the data
- Displaying the values inside the UI
- Sending the readings to the NestJS backend

3.2 App Functional Modules

The Flutter app includes several key modules:

1. BLE Scanner Module

- Scans for BLE devices
- Filters based on the Raspberry Pi advertising name
- Attempts connection with retry logic

2. BLE Reader Module

- Reads the BLE characteristic containing the sensor data
- Parses the values
- Shows them in the UI

3. API Communication Module

- Sends readings to the NestJS backend through HTTP POST
- Handles network errors and retries
- Stores offline data (optional)

4. UI Presentation Layer

Displays:

- Current readings
- Historical logs
- Connection state
- Device information

3.3 Data Flow

The flow is:

1. BLE characteristic received →
2. Parsed into a Dart model →

3. POST request made to NestJS backend →
4. Backend stores into database →
5. Flutter can fetch history using /readings endpoint.

3.4 Error Handling

- Automatic reconnection when BLE disconnects
- Backend unavailability → temporary local storage
- Incorrect sensor format → fallback parsing

3.5 Onboarding & Device Pairing

The app can optionally include:

- QR pairing
- Saved device list
- BLE permissions flow
- Background scanning

---

4. NestJS Backend (Documentation)

4.1 Role of the Backend

The backend is the **data centralization layer**.

It receives POST requests from the Flutter app and stores every reading in the database. It also exposes endpoints to retrieve the values.

4.2 Backend Flow

Incoming Data

Flutter sends sensor data via POST:

POST /readings

The backend:

1. Validates the payload
2. Stores the reading in the database
3. Returns HTTP 201 (Created)

Retrieving Data

Two useful endpoints exist:

1. Get all readings

GET /readings

Returns full history.

2. Get the latest reading

GET /readings/latest

Useful to show in the Flutter home screen.

4.3 Readings Storage

When Flutter sends readings, the backend saves them in a database table or collection:

Fields include:

- id (primary key)
- temperature
- humidity
- timestamp
- device_id (optional for multiple devices)

Stored in SQL or NoSQL depending on configuration.

4.4 Folder Structure Overview

Typical NestJS structure:

src/

readings/

readings.controller.ts

readings.service.ts

readings.module.ts

dto/

entity/

main.ts

Even without showing code, these files handle:

- Controller: receives REST requests
- Service: business logic
- Entity/Model: represents table in DB

4.5 Running the Backend

The backend is launched using:

npm run start

or in development:

npm run start:dev

NestJS will listen by default on:

http://0.0.0.0:3000

This ensures **the server is accessible from mobile devices**, not only localhost.

4.6 Deployment & Production Notes

When deploying:

- Expose port 3000
- Use PM2 or systemd to run in background

- Use a reverse proxy (Nginx)
- Enable HTTPS
- Backup the database periodically

---

5. End-to-End Data Flow (Full System)

Below is the complete sequence:

1. **Sensor → Raspberry Pi**
    - Pi reads sensors continuously.
2. **Pi → Flutter via BLE**
    - Flutter connects to BLE characteristic.
    - Retrieves live sensor values.
3. **Flutter → NestJS via REST**
    - App parses BLE values.
    - Sends structured reading to backend.
4. **NestJS → Database**
    - Stores reading in DB.
    - Makes it available for querying.
5. **Flutter → Backend (History Retrieval)**
    - Fetches /readings to show past data.
    - Fetches /readings/latest for dashboard.

---

6. Use Cases

1. Real-Time Monitoring

Live values shown on-screen as BLE updates happen.

2. Historical Logging

All sensor activity stored on the backend for analytics.

3. Alerts & Notifications

Backend can send alerts if temperature or humidity exceed thresholds.

4. Device Diagnostics

Flutter shows:

- Connection status
- Last reading timestamp
- Battery level (if implemented)

---

7. Future Improvements

You can extend the system with:

🔷 **Backend**

- User authentication
- Multi-device support
- Device registration system
- GraphQL support
- Long-term storage in time-series DB (InfluxDB)

🔷 **Raspberry Pi**

- BLE pairing
- Multiple sensors
- Power saving
- Local caching

🔷 **Flutter**

- Offline mode
- BLE auto-reconnect
- Graphical charts (temperature over time)
- Notification system