# IoT Dokumentacja Smart Lighting System

January 2026

## Spis treści

# 1 System Inteligentnego Oświetlenia LED z Monitorowaniem Wilgotności

## 1.1 Zespół

| Lp. | Imię i Nazwisko | Email |
|---|---|---|
| 1. | Mikołaj Kołek | mkolek@student.agh.edu.pl |
| 2. | Jan Jędra | jjedra@student.agh.edu.pl |
| 3. | Karol Bystrek | karbystrek@student.agh.edu.pl |
| 4. | Patryk Chamera | pchamera@student.agh.edu.pl |

## 1.2 Cel i założenia projektu

### 1.2.1 Grupa docelowa

System został zaprojektowany dla **magazynów zbiorów w archiwach, bibliotekach i muzeach**, gdzie kontrola warunków środowiskowych jest kluczowa dla zachowania cennych zbiorów.

### 1.2.2 Cel projektu

System zapewnia:

- **Stałą kontrolę wilgotności** - ciągły monitoring parametrów środowiskowych z pełnym zapisem historycznym do audytu
- **Szybką sygnalizację stanu** - natychmiastowy sygnał świetlny przy detekcji ruchu pracownika
- **Pełny zapis historyczny** - wszystkie dane telemetryczne są zapisywane w bazie danych, umożliwiając analizę trendów i audyt
- **Automatyczną reakcję** - system automatycznie dostosowuje kolor sygnalizacji w zależności od poziomu wilgotności, umożliwiając szybką ocenę warunków bez konieczności sprawdzania aplikacji

### 1.2.3 Zastosowanie

Urządzenie montowane w pomieszczeniach magazynowych, gdzie:

- Pracownicy potrzebują szybkiej informacji wizualnej o stanie wilgotności
- Wymagany jest pełny zapis danych do audytu i analizy
- Konieczna jest automatyczna sygnalizacja przy wejściu personelu

## 1.3 Wykorzystany sprzęt i komunikacja

### 1.3.1 3.1 Mikrokontroler ESP32

**Producent:** Espressif Systems
**Link:** ESP32 na Temu ESP32 wycofane z Temu na którym kupiliśmy (w momencie pisania dokumentacji widzimy, że link nie aktywny)

**Funkcja w systemie:**

- Obsługuje całą logikę systemu (odczyt czujników, sterowanie LED, komunikacja sieciowa)
- Komunikacja bezprzewodowa:
  - **WiFi:** 802.11 b/g/n (2.4 GHz) - komunikacja z backendem przez MQTT
  - **Bluetooth Low Energy (BLE):**
    - Tryb serwera: provisioning WiFi (konfiguracja przez aplikację)
    - Tryb klienta: odczyt danych z czujnika Xiaomi

**Interfejsy wykorzystane:**

- **GPIO:** Sterowanie LED, odczyt czujników
- **I2C:** Komunikacja z BMP280 (GPIO21-SDA, GPIO22-SCL)
- **ADC:** Odczyt fotorezystora (GPIO34, ADC1_CHANNEL_6)
- **RMT:** Sterowanie paskiem LED WS2812B (GPIO4)

#### 3.1.1 Tabela interfejsów komunikacyjnych

| Interfejs | Typ | Komponent | Funkcja |
|---|---|---|---|
| **ADC** | Analogowe wejście | Fotorezystor GL5616 | Odczyt natężenia światła do regulacji jasności LED |
| **I2C** | Cyfrowa magistrala szeregowa | BMP280 | Odczyt ciśnienia atmosferycznego |
| **GPIO (cyfrowe)** | Cyfrowe we/wy | HC-SR04 (Trig) | Wyzwalanie pomiaru odległości |
| **GPIO (cyfrowe)** | Cyfrowe wejście | HC-SR04 (Echo) | Odbieranie sygnału echa |
| **GPIO (cyfrowe)** | Cyfrowe wejście | Przycisk BOOT | Wejście w tryb provisioning |
| **RMT** | Cyfrowy protokół | WS2812B LED | Sterowanie paskiem LED (protokół jednoprzewodowy) |
| **BLE (Peripheral)** | Bezprzewodowy | ESP32 → Aplikacja | Provisioning WiFi (serwer GATT) |
| **BLE (Central)** | Bezprzewodowy | ESP32 → Xiaomi | Odczyt temperatury i wilgotności (klient GATT) |

| Interfejs | Typ | Komponent | Funkcja |
|---|---|---|---|
| **WiFi** | Bezprzewodowy | ESP32 → Backend | Komunikacja z serwerem przez MQTT |
| **MQTT** | Protokół aplikacyjny | ESP32 ↔ Broker | Publikacja telemetrii, odbieranie komend |

### 1.3.2 3.2 Czujnik ciśnienia atmosferycznego BMP280

**Producent:** Bosch Sensortec
**Link:** BMP280 na Temu

**Funkcja w systemie:**

- **Używany TYLKO do odczytu ciśnienia atmosferycznego** (nie wykorzystujemy temperatury z tego czujnika)
- Dane ciśnienia są zapisywane w telemetrii i mogą być wykorzystane do analizy warunków atmosferycznych

**Komunikacja:**

- **Protokół:** I2C (Inter-Integrated Circuit)
- **Częstotliwość:** 100 kHz (standard)
- **Piny połączenia:**
  - VCC → 3.3V ESP32
  - GND → GND ESP32
  - SDA → GPIO21 (ESP32)
  - SCL → GPIO22 (ESP32)

**Parametry:**

- Zakres ciśnienia: 300-1100 hPa
- Dokładność: ±1 hPa
- Odczyt: Co 30 sekund (w tasku `sensor_reading_task`)

### 1.3.3 3.3 Czujnik odległości/ruchu HC-SR04

**Producent:** SparkFun
**Link:** HC-SR04 na Botland

**Funkcja w systemie:**

- **Służy do detekcji ruchu pracownika** w pomieszczeniu
- Gdy odległość spada poniżej progu konfigurowalnego (domyślnie 50 cm), system interpretuje to jako wykrycie ruchu i aktywuje oświetlenie LED

**Zasada działania:**

- Emisja fal ultradźwiękowych (40 kHz) i pomiar czasu powrotu echa
- Zakres pomiarowy: 2-400 cm
- Dokładność: 3 mm

**Piny połączenia:**

- VCC → 5V ESP32
- GND → GND ESP32
- Trig (Trigger) → GPIO5 (ESP32) - wyjście cyfrowe
- Echo → GPIO18 (ESP32) - wejście cyfrowe

**Częstotliwość pomiarów:**

- **Tryb aktywny:** Co 50 ms (20 pomiarów na sekundę) - gdy wykryto ruch w ciągu ostatnich 5 sekund
- **Tryb uśpienia:** Co 500 ms (2 pomiary na sekundę) - gdy brak ruchu przez 5+ sekund

### 1.3.4 3.4 Czujnik temperatury i wilgotności Xiaomi Mi Temperature and Humidity Monitor 2

**Producent:** Xiaomi
**Link:** Xiaomi Mi Temperature and Humidity Monitor 2

**Funkcja w systemie:**

- Główny czujnik do pomiaru **wilgotności powietrza** (używany do określenia koloru LED)
- Pomiar temperatury

**Komunikacja:**

- **Protokół:** Bluetooth Low Energy (BLE)
- **Tryb:** ESP32 działa jako **klient GATT** (nie odczytujemy advertising frames, tylko łączymy się i odczytujemy charakterystyki)
- **Proces odczytu:**
    1. ESP32 skanuje sieć BLE w poszukiwaniu urządzenia
    2. Po znalezieniu, ESP32 łączy się jako klient GATT
    3. Odkrywa serwisy i charakterystyki:
        - **Environmental Service** (UUID: 0x181A):
            - Temperature Characteristic (UUID: 0x2A6E) - odczyt temperatury
            - Humidity Characteristic (UUID: 0x2A6F) - odczyt wilgotności
        - **Battery Service** (UUID: 0x180F):
            - Battery Level Characteristic (UUID: 0x2A19) - poziom baterii
    4. Odczytuje wartości
    5. Rozłącza się po odczycie
- **Częstotliwość odczytu:** Co 30 sekund (w tasku `sensor_reading_task`)

**Format danych:**

- Temperatura: int16_t (raw)$/100.0 =$ wartość w °C
- Wilgotność: uint16_t (raw)$/100.0 =$ wartość w %

### 1.3.5 3.5 Czujnik natężenia światła - Fotorezystor

**Producent:** GL5616
**Link:** Fotorezystor 5-10kΩ GL5616 na Botland

**Funkcja w systemie:**

- Pomiar natężenia światła otoczenia w celu automatycznej regulacji jasności diod LED
- Wysokie natężenie światła (słoneczny dzień) → zmniejsza jasność LED
- Niskie natężenie (ciemność) → zwiększa jasność LED
- Możliwe jest również użycie ręcznej regulacji jasności z pominięciem fotorezystora

**Komunikacja:**

- **Interfejs:** ADC (Analog-to-Digital Converter)
- **Pin:** GPIO34 (ADC1_CHANNEL_6)

**Logika jasności:**

- **Auto-brightness włączone:**
  - Ciemno (0-30%): 100% jasności LED
  - Średnio (30-70%): 50% jasności LED
  - Jasno (70-100%): 20% jasności LED
- **Auto-brightness wyłączone:** Używa ręcznie ustawionej wartości (0-100%)

### 1.3.6 3.6 Sygnalizacja - Diody LED WS2812B

**Link:** Pasek LED WS2812B na AliExpress

**Funkcja w systemie:**

- **Wmontowane w obudowę lampy zamiast oryginalnych diod**
- Główny element wyjściowy systemu - wyświetla kolory RGB określone na podstawie wilgotności powietrza
- Jasność regulowana przez fotorezystor lub ręcznie
- LED włączają się automatycznie po wykryciu ruchu przez HC-SR04

**Parametry:**

- **Typ:** Pasek diod LED RGB adresowalnych indywidualnie (NeoPixel)
- **Liczba diod:** 24 diody (konfigurowalna liczba aktywnych, u nas używamy 5)
- **Komunikacja:** Protokół jednoprzewodowy (single-wire) z sygnałem zegarowym
- **Pin połączenia:** GPIO4 (D4) - sterowanie przez RMT (Remote Control)

**Sterowanie:**

- Moduł RMT ESP32 generuje precyzyjne sygnały czasowe wymagane przez protokół WS2812B
- Każda dioda może być sterowana indywidualnie (adresowalne)

### 1.3.7 3.7 Zasilanie

**3.7.1 Akumulator Li-ion 18650** **Link:** Akumulator Li-ion 18650 1200mAh
**Pojemność:** 1200 mAh
**Funkcja:** Główne źródło zasilania urządzenia

**3.7.2 Panel solarny**

- Ładowanie akumulatora w ciągu dnia
- Zintegrowany z obudową lampy
- Współpracuje z modułem ładowania JC-21461-USB

**3.7.3 Moduł ładowania JC-21461-USB**

- **Wbudowany w lampę**
- Obsługuje ładowanie z:
  - **USB-C** - możliwość ładowania z zewnętrznego źródła
  - **Panelu solarnego** - automatyczne ładowanie w ciągu dnia

**Zalety:**

- Autonomiczne działanie dzięki panelowi solarnemu
- Możliwość szybkiego ładowania przez USB-C
- Ochrona akumulatora przed uszkodzeniem

## 1.4 Opis funkcjonalności i logiki działania

**Platforma chmurowa:** System wykorzystuje **AWS IoT Core** jako broker MQTT w chmurze. Komunikacja MQTT między urządzeniami ESP32 a backendem odbywa się przez AWS IoT Core, zapewniając skalowalną i bezpieczną infrastrukturę komunikacyjną.

### 1.4.1 4.1 Logika LED

**4.1.1 Określanie koloru na podstawie wilgotności** **Źródło danych:** Czujnik Xiaomi Mi Temperature and Humidity Monitor 2 (odczyt przez BLE)
**Algorytm:**

1. System odczytuje aktualną wilgotność z czujnika Xiaomi (co 30 sekund).
2. Wartość wilgotności jest porównywana z **konfigurowalnymi progami** (4 wartości progowe):
   - Próg 0: Minimum (domyślnie 0%)
   - Próg 1: Granica niska-średnia (domyślnie 30%)
   - Próg 2: Granica średnia-wysoka (domyślnie 70%)
   - Próg 3: Maximum (domyślnie 100%)
3. Na podstawie progu, system wybiera jeden z **3 konfigurowalnych kolorów RGB**:
   - **Kolor 0:** Dla wilgotności $<$ próg 1 (domyślnie: czerwony - 255,0,0)
   - **Kolor 1:** Dla wilgotności między progiem 1 a 2 (domyślnie: niebieski - 0,0,255)
   - **Kolor 2:** Dla wilgotności $\geq$ próg 2 (domyślnie: zielony - 0,255,0)

#### 4.1.2 Określanie jasności Dwa tryby pracy:

1. **Auto-brightness (automatyczna):**
    - Jasność zależy od natężenia światła z fotorezystora.
    - Logika:
        - Ciemno (0-30% natężenia): 100% jasności LED (255/255)
        - Średnio (30-70% natężenia): 50% jasności LED (128/255)
        - Jasno (70-100% natężenia): 20% jasności LED (51/255)
2. **Manual brightness (ręczna):**
    - Jasność jest ustawiona na stałe w konfiguracji (0-100%).
    - Fotorezystor jest ignorowany.

**Konfiguracja:** Przez komendę MQTT `SET_CONFIG` (pole `autobrightness: true/false`, `brightnessPct: 0-100`).

#### 4.1.3 Aktywacja LED po wykryciu ruchu Warunki aktywacji:

- LED zapalają się **tylko po wykryciu ruchu** przez HC-SR04
- Odległość $<$ próg konfigurowalny (50 cm)
- LED pozostają włączone podczas ciągłego wykrywania ruchu

**Auto-off LED:**

- **Timeout braku ruchu:** Po braku wykrycia ruchu przez konfigurowalny czas (domyślnie 15 sekund) $\rightarrow$ LED wyłączają się
- **Maksymalny czas świecenia:** Maksymalny czas włączenia LED (domyślnie 5 minut) $\rightarrow$ LED wyłączają się automatycznie

### 1.4.2 4.2 Provisioning (Wi-Fi)

#### 4.2.1 Uruchomienie trybu Provisioning Sposoby wejścia w tryb provisioning:

1. **Pierwsze uruchomienie urządzenia:**
    - Jeśli urządzenie nie ma zapisanych danych WiFi w NVS (Non-Volatile Storage)
    - Automatyczne wejście w tryb provisioning
2. **Przytrzymanie przycisku BOOT:**
    - Przytrzymanie przycisku BOOT przez **3 sekundy**
    - Urządzenie wchodzi w tryb provisioning (nawet jeśli ma zapisane dane WiFi)

**Stan urządzenia w trybie provisioning:**

- Zatrzymuje połączenie WiFi (jeśli aktywne)
- Uruchamia advertising BLE
- Udostępnia serwis GATT z charakterystykami provisioning

#### 4.2.2 Proces parowania przez aplikację Kroki:

1. **Użytkownik otwiera aplikację webową/mobilną** i przechodzi do ekranu "Setup WiFi"
2. **Skanowanie urządzeń BLE:**
    - Aplikacja skanuje urządzenia BLE w poszukiwaniu serwisu provisioning (UUID serwisu).
    - Użytkownik wybiera urządzenie z listy.
3. **Połączenie i odczyt danych urządzenia:**
    - Aplikacja łączy się z urządzeniem przez BLE GATT.
    - Odczyt charakterystyki **MAC Address** (read-only, 6 bajtów) - wyświetlany użytkownikowi.
    - Odczyt charakterystyki **Proof of Possession (PoP)** (read-only, 32 bajty hex) - wyświetlany użytkownikowi.
    - **MAC i PoP są wymagane do "claimowania" urządzenia w backendzie.**
4. **Wprowadzenie danych:**
    - Użytkownik wprowadza:
        - **SSID** sieci WiFi
        - **Hasło** WiFi
        - **Lokalizację**
        - **Nazwę urządzenia**
5. **Wysłanie konfiguracji:**
    - Aplikacja zapisuje dane do charakterystyk GATT:
        - `CHR_SSID`: SSID
        - `CHR_PASS`: Hasło
        - `CHR_IDS`: csv z customerId, locationId, deviceId
        - `CHR_CMD`: bajt 0x01 (SAVE) - uruchamia proces połączenia
6. **Przetworzenie w ESP32:**
    - `ble_provisioning_gatt_write_cb()` odbiera zapis danych WiFi.
    - Zapisuje SSID i hasło do NVS (`wifi_config_save()`).
    - Po otrzymaniu komendy "CONNECT": restartuje system.
7. **Połączenie z WiFi:**
    - Po restarcie, ESP32:
        - Odczytuje dane WiFi z NVS
        - Łączy się z siecią WiFi
        - Po połączeniu: łączy się z brokerem MQTT

---

#### 4.2.3 Claiming urządzenia w backendzie Proces:

1. Użytkownik w aplikacji webowej wprowadza:
    - MAC address (odczytany wcześniej przez BLE)

- Proof of Possession (PoP) (odczytany wcześniej przez BLE)

2. Wysyła żądanie: `POST /api/devices/claim`
3. Backend:
    - Weryfikuje PoP (porównuje z zapisanym w bazie danych lub tworzy nowy rekord)
    - Przypisuje urządzenie do użytkownika
    - Ustawia status urządzenia na "CLAIMED"
4. Urządzenie jest teraz widoczne w dashboardzie użytkownika

**Bezpieczeństwo:**

- Proof of Possession (PoP) - 32-bajtowy losowy ciąg znaków hex generowany przy pierwszym uruchomieniu.
- Zapisany w NVS urządzenia i wymagany do claimowania.
- Zapobiega nieautoryzowanemu przypisaniu urządzenia do innego użytkownika.

### 1.4.3 4.3 Tryb Uśpienia

#### 4.3.1 Zasada działania Warunek wejścia w tryb uśpienia:

- Jeśli przez **5 sekund nie wykryto ruchu** (odległość $>$ próg) $\rightarrow$ Tryb "uśpienia"

**Zachowanie w trybie uśpienia:**

- **Detekcja ruchu:** Odbywa się tylko **2 razy na sekundę** (co 500 ms), zamiast standardowego próbkowania (co 50 ms = 20 razy na sekundę).
- **Komunikacja sieciowa/serwerowa:** Jest **podtrzymywana cały czas** (nie jest usypiana).
    - Połączenie WiFi pozostaje aktywne
    - Połączenie MQTT pozostaje aktywne
    - Publikacja danych telemetrycznych kontynuowana (co 30 sekund)
    - Odbieranie komend MQTT działa normalnie

**Warunek wyjścia z trybu uśpienia:**

- Wykrycie ruchu (odległość $<$ próg) $\rightarrow$ Powrót do trybu aktywnego (próbkowanie co 50 ms).

### 1.4.4 4.4 OTA (Over-The-Air Updates)

#### 4.4.1 Proces aktualizacji Krok 1: Sprawdzanie wersji

- System periodycznie sprawdza dostępność nowej wersji firmware (domyślnie co 60 sekund - `OTA_CHECK_INTERVAL_MS`).
- Żądanie HTTP GET: `/api/firmware/check?mac={MAC_ADDRESS}&version={CURRENT_VERSION}`
- Przykład: `/api/firmware/check?mac=AA:BB:CC:DD:EE:FF&version=29.0`

- Wersja jest zdefiniowana w pliku `main_esp/main/ota_update.h` jako `FIRMWARE_VERSION "29.0"`.

**Krok 2: Porównanie wersji**

- Backend porównuje wersję z żądania z najnowszą dostępną wersją.
- Format wersji: `X.Y.Z` (np. `29.0`, `24.0`).
- Porównanie semantyczne (major.minor.patch).

**Krok 3: Odpowiedź backendu**

- **Status 204 (No Content):** Brak nowszej wersji → aktualizacja pomijana.
- **Status 200 (OK):** Nowa wersja dostępna → JSON z danymi:

```
{
  "version": "30.0",
  "url": "https://backend.example.com/api/firmware/download/30.0"
}
```

**Krok 4: Weryfikacja wersji (ESP32)**

- ESP32 porównuje wersję z odpowiedzi z aktualną wersją (`FIRMWARE_VERSION` w `ota_update.h`).
- Jeśli nowa wersja jest faktycznie nowsza (semantycznie) → kontynuacja.
- Jeśli nie → aktualizacja pomijana (ochrona przed rollbackiem).

**Krok 5: Pobranie firmware**

- ESP32 pobiera plik firmware z URL podanego w odpowiedzi.
- Protokół: HTTPS (ESP-IDF `esp_https_ota`).
- Pobieranie na **osobną partycję OTA** (nie nadpisuje aktualnie działającej partycji).
- Po pomyślnym pobraniu: partycja jest oznaczana jako aktywna do następnego boota.

**Krok 6: Reboot i uruchomienie nowej partycji**

- Po pomyślnym pobraniu, ESP32:
  - Oznacza nową partycję jako aktywną
  - Restartuje system
  - Uruchamia firmware z nowej partycji
  - Po pierwszym uruchomieniu: `ota_mark_boot_valid()` - oznacza boot jako poprawny (ochrona przed rollbackiem)

**Krok 7: Ochrona przed rollbackiem**

- Jeśli nowa wersja nie uruchomi się poprawnie, ESP32 może wrócić do poprzedniej partycji.
- Po pomyślnym uruchomieniu: boot jest oznaczany jako “valid”, co blokuje automatyczny rollback.

#### 4.4.2 Konfiguracja wersji **Plik:** `main_esp/main/ota_update.h`

```
#define FIRMWARE_VERSION "29.0"
```

### 1.4.5 ESP32 Flash Encryption

W systemie wdrożono sprzętowe szyfrowanie pamięci Flash Encryption (AES-256). Cały firmware (Bootloader, Tablica partycji oraz Aplikacja) został zaszyfrowany unikalnym kluczem prywatnym zapisanym w bezpiecznikach eFuse procesora.

#### 4.5.1 Stan zabezpieczeń urządzenia:

1. **Klucz prywatny**: Zaszyty na stałe w `BLOCK1` eFuse.
2. **Ochrona Odczytu (RD_DIS)**: Aktywna, klucz jest niewidoczny dla oprogramowania, dostęp ma wyłącznie sprzętowy silnik AES.
3. **Integralność**: Każda próba wgrania niezaszyfrowanego kodu lub kodu zaszyfrowanego innym kluczem skutkuje blokadą startu procesora.

## 1.5 Komunikacja i Aplikacje

### 1.5.1 5.1 BLE (Bluetooth Low Energy)

**5.1.1 Skanowanie urządzeń BLE** **Kontekst:** Odczyt danych z czujnika Xiaomi Mi Temperature and Humidity Monitor 2

**Proces skanowania:**

1. ESP32 uruchamia skanowanie BLE (`ble_gap_disc_start()`) w zadaniu `sensor_reading_task`.
2. Filtrowanie po nazwie urządzenia: "ATC_8E4B89" (zdefiniowane jako `kTargetName_Sensor` w `sensor_task.cpp`).
3. Po znalezieniu urządzenia w advertising frames: anulowanie skanowania (`ble_gap_disc_cancel()`) i inicjacja połączenia GATT.
4. Timeout skanowania: 30 sekund (jeśli nie znaleziono urządzenia, próba jest powtarzana w następnym cyklu).

**5.1.2 Parsowanie danych z czujnika Xiaomi** System nie parsuje advertising frames. Zamiast tego, łączy się jako klient GATT i odczytuje charakterystyki.

**Proces odczytu:**

1. **Połączenie GATT:**
   - ESP32 łączy się z urządzeniem jako klient GATT (`ble_gap_connect()`).
   - Timeout połączenia: 30 sekund (semafor `s_ble_sem` z timeoutem 30000 ms).
   - Połączenie jest zamykane natychmiast po odczycie wszystkich wartości.
2. **Odkrywanie serwisów:**
   - `ble_gattc_disc_all_svcs()` - odkrywa wszystkie serwisy.

- Szukane serwisy:
  - **Environmental Service** (UUID: 0x181A)
  - **Battery Service** (UUID: 0x180F)

3. **Odkrywanie charakterystyk:**
   - `ble_gattc_disc_all_chrs()` - odkrywa charakterystyki w każdym serwisie.
   - Zapisuje "handles" (identyfikatory) charakterystyk:
     - Temperature (UUID: 0x2A6E)
     - Humidity (UUID: 0x2A6F)
     - Battery (UUID: 0x2A19)
4. **Odczyt wartości:**
   - `ble_gattc_read()` - odczytuje wartości z charakterystyk sekwencyjnie (callback chain).
   - Kolejność odczytu: Temperatura $\rightarrow$ Wilgotność $\rightarrow$ Bateria.
   - **Format danych:**
     - Temperatura: 2 bajty (`int16_t`, little-endian) / 100.0 $\rightarrow$°C
     - Wilgotność: 2 bajty (`uint16_t`, little-endian) / 100.0 $\rightarrow$ %
     - Bateria: 1 bajt (`uint8_t`) $\rightarrow$ % (0-100)
5. **Rozłączenie i aktualizacja:**
   - Po odczycie wszystkich wartości, ESP32 rozłącza się (`ble_gap_terminate()`).
   - Dane są aktualizowane w cache `LatestSensorData::update()` (thread-safe singleton).
   - Cache jest wykorzystywany przez `distance_sensor_task` do określenia koloru LED.
   - Dane są również dodawane do kolejki `SensorManager` do wysłania przez MQTT.

**Częstotliwość:** Co 30 sekund (zdefiniowane jako `SENSOR_READ_INTERVAL_MS = 30000` w `sensor_task.cpp`).

- Odczyt jest wykonywany w zadaniu FreeRTOS `sensor_reading_task`.
- Jeśli provisioning jest aktywny, odczyt jest pomijany (opóźnienie 5 sekund).

### 1.5.2 5.2 MQTT

**5.2.1 Struktura tematów (Topics)** **Format:** `smart-led/device/{MAC_ADDRESS}/{TYPE}`
**Przykłady:**

- `smart-led/device/AA:BB:CC:DD:EE:FF/telemetry`
- `smart-led/device/AA:BB:CC:DD:EE:FF/status`
- `smart-led/device/AA:BB:CC:DD:EE:FF/cmd`

**Typy tematów:**

| Typ | Suffix | Kierunek | Opis |
|---|---|---|---|
| **Telemetry** | `/telemetry` | Device → Backend | Dane telemetryczne (batch) |
| **Status** | `/status` | Device → Backend | Status połączenia (`online`/`offline`) |
| **Command** | `/cmd` | Backend → Device | Komendy sterujące |
| **Response** | `/resp` | Device → Backend | Odpowiedzi na komendy |

**MAC Address format:** `AA:BB:CC:DD:EE:FF` (wielkie litery, dwukropek jako separator)

### 5.2.2 Format komunikatów JSON

**Telemetry (Device → Backend)** **Topic:** `smart-led/device/{MAC}/telemetry`
**QoS:** 1
**Retained:** false
**Format:** JSON Array (batch do 50 rekordów)

```
[
  {
    "timestamp": 1702464000,
    "temperature": 22.5,
    "humidity": 45.0,
    "pressure": 1013.2,
    "personCount": 1
  },
  {
    "timestamp": 1702464030,
    "temperature": 22.6,
    "humidity": 45.2,
    "pressure": 1013.3,
    "personCount": 0
  }
]
```

**Pola:**

- `timestamp` (int64): Unix timestamp (sekundy)
- `temperature` (float): Temperatura w °C
- `humidity` (float): Wilgotność w %
- `pressure` (float): Ciśnienie atmosferyczne w hPa
- `personCount` (int): Liczba wykrytych osób od ostatniego odczytu

**Częstotliwość publikacji:**

- Co 30 sekund (domyślnie) - zadanie `mqtt_publishing_task` sprawdza kolejkę co 30 sekund
- Lub gdy kolejka osiągnie 50 rekordów (batch processing) - dane są wysyłane natychmiast w partiach

**Status (Device → Backend)** **Topic:** `smart-led/device/{MAC}/status`
**QoS:** 1
**Retained:** true (wiadomość jest zachowana dla nowych subskrybentów)
**Format:** JSON Object

```
{
  "state": "online"
}
```

**Wartości `state`:**

- `"online"` - Urządzenie jest połączone i aktywne
- `"offline"` - Urządzenie jest rozłączone (Last Will and Testament)

**Last Will and Testament (LWT):**

- Przy rozłączeniu ESP32, broker MQTT automatycznie publikuje `{"state": "offline"}` na temat `/status`.
- Backend aktualizuje status urządzenia w bazie danych.

**Publikacja:**

- Przy połączeniu MQTT: automatyczna publikacja `{"state": "online"}`
- Przy rozłączeniu: LWT publikuje `{"state": "offline"}`

---

**Command (Backend → Device)** **Topic:** `smart-led/device/{MAC}/cmd`
**QoS:** 1
**Retained:** false
**Format:** JSON Object
**Komenda SET_CONFIG:**

```
{
  "type": "SET_CONFIG",
  "payload": {
    "humidityThresholds": [0.0, 30.0, 70.0, 100.0],
    "colors": ["FF0000", "00FF00", "0000FF"],
    "brightnessPct": 80,
    "autobrightness": true,
    "numLeds": 6,
    "noMotionTimeoutSec": 15,
    "maxOnDurationSec": 300,
    "distanceThresholdCm": 50.0
  }
}
```

**Komenda SET_COLOR:**

```
{
  "type": "SET_COLOR",
  "payload": {
    "red": 255,
    "green": 0,
    "blue": 0
  }
}
```

**Komenda SET_BRIGHTNESS:**

```
{
  "type": "SET_BRIGHTNESS",
  "payload": {
    "brightness": 128
  }
}
```

**Komenda GET_CONFIG:**

```
{
  "type": "GET_CONFIG"
}
```

**Odpowiedź na GET_CONFIG (Device → Backend):**

- Publikowana na temat `smart-led/device/{MAC}/config` (QoS 1)
- Format JSON z aktualną konfiguracją urządzenia

**5.2.3 Konfiguracja MQTT** **Broker:** AWS IoT Core (produkcja) / Eclipse Mosquitto 2.0 (development)
**Port:** 8883 (TLS - AWS IoT Core), 1883 (TCP - Mosquitto)
**Protokół:** MQTT 3.1.1
**Bezpieczeństwo:** TLS/SSL z certyfikatami X.509 (AWS IoT Core), autoryzacja przez AWS IAM

**Konfiguracja klienta ESP32:**

- URI: Konfigurowalne (domyślnie: `mqtt://192.168.1.1:1883`)
- Client ID: MAC address urządzenia
- Keep Alive: 60 sekund
- Last Will and Testament: `{"state": "offline"}` na temat `/status`

**Subskrypcje:**

- **ESP32 subskrybuje:**
  - `smart-led/device/{MAC}/cmd` (QoS 1)
- **Backend subskrybuje:**
  - `smart-led/device/+/telemetry` (QoS 1) - dane telemetryczne
  - `smart-led/device/+/status` (QoS 1) - status urządzeń

### 1.5.3 Aplikacja Serwerowa (Backend)

**Technologie:**

- **Framework:** Spring Boot 4.0.1 (Java 21)
- **Baza danych:** PostgreSQL 17
- **ORM:** Hibernate/JPA
- **Bezpieczeństwo:** Spring Security z JWT
- **Dokumentacja API:** Swagger/OpenAPI 3.0

#### 5.3.1 Kluczowe funkcjonalności

**Odbieranie danych telemetrycznych** **Endpoint:** Automatyczne przez MQTT (nie REST)
**Proces:**

1. Backend subskrybuje tematy `smart-led/device/+/telemetry`.
2. `MqttIngestionService.handleTelemetry()` przetwarza wiadomości.
3. Parsuje JSON array do listy `TelemetryDto`.
4. Dla każdego rekordu: `TelemetryService.saveTelemetry()` zapisuje do bazy danych.
5. Dane są przypisane do urządzenia po MAC address.

**Endpoint REST do odczytu:**

- `GET /api/devices/{deviceId}/telemetry` - historia telemetrii (z filtrowaniem po dacie)
- `GET /api/devices/{deviceId}/telemetry/latest` - najnowsze dane

**Claiming urządzeń** **Endpoint:** `POST /api/devices/claim`
**Request Body:**

```
{
  "macAddress": "AA:BB:CC:DD:EE:FF",
  "proofOfPossession": "abc123...",
  "name": "Urządzenie 1",
  "locationId": 1
}
```

**Proces:**

1. Weryfikacja PoP (Proof of Possession).
2. Sprawdzenie, czy urządzenie nie jest już przypisane.
3. Utworzenie rekordu `Device` w bazie danych.
4. Przypisanie do użytkownika (z tokenu JWT) i lokalizacji.
5. Zwrócenie danych urządzenia.

**Bezpieczeństwo:**

- Wymagany token JWT (autoryzacja).
- PoP musi być zgodny z zapisanym w bazie (lub nowy rekord jest tworzony).

**Historia/audyt Funkcjonalności:**

- **Zapis wszystkich danych telemetrycznych** - każdy odczyt jest zapisywany w bazie danych z timestampem.
- **Historia zmian statusu** - logowanie zmian statusu urządzenia (ONLINE/OFFLINE).
- **Filtrowanie po dacie** - możliwość pobrania danych z określonego zakresu czasowego.
- **Agregacja danych** - możliwość analizy trendów (średnie, min, max w określonym okresie).

**Endpointy:**

- `GET /api/devices/{deviceId}/telemetry ?from={timestamp}&to={timestamp}` - historia z filtrowaniem
- `GET /api/devices/{deviceId}/telemetry/latest` - najnowsze dane

**Struktura bazy danych:**

- Tabela `telemetry`: `id`, `device_id`, `timestamp`, `temperature`, `humidity`, `pressure`, `person_count`
- Relacja: `Device` → wiele `Telemetry` (one-to-many)

**Wysyłanie komend do urządzeń Endpoint:** `POST /api/devices/{deviceId}/control`
**Request Body:**

```
{
  "type": "SET_CONFIG",
  "payload": {
    "humidityThresholds": [0.0, 30.0, 70.0, 100.0],
    "colors": ["FF0000", "00FF00", "0000FF"]
  }
}
```

**Proces:**

1. Weryfikacja tokenu JWT i uprawnień (czy urządzenie należy do użytkownika).
2. Pobranie urządzenia z bazy danych (MAC address).
3. `CommandService.sendCommand()` tworzy JSON payload.
4. `MqttPublisher.publish()` publikuje na temat `smart-led/device/{MAC}/cmd`.
5. ESP32 odbiera komendę i wykonuje akcję.

**Dostępne komendy:**

- `SET_CONFIG` - aktualizacja konfiguracji LED
- `SET_COLOR` - bezpośrednie ustawienie koloru LED
- `SET_BRIGHTNESS` - ustawienie jasności LED
- `GET_CONFIG` - pobranie aktualnej konfiguracji

#### 5.3.2 Architektura Warstwy:

- **Controller** (`controller/`) - endpointy REST API
- **Service** (`service/`) - logika biznesowa
- **Repository** (`repository/`) - dostęp do bazy danych (JPA)
- **Model** (`model/`) - encje bazy danych
- **MQTT** (`mqtt/`) - integracja z brokerem MQTT
- **Security** (`config/`, `jwt/`) - autoryzacja i bezpieczeństwo

### 1.5.4 Aplikacja Webowa (Frontend)

**Technologie:**

- **Framework:** React 18.2.0
- **Build Tool:** Vite 5.1.4
- **Stylowanie:** Tailwind CSS 3.4.1
- **Routing:** React Router DOM 6.22.0
- **Ikony:** Lucide React

#### 5.4.1 Funkcjonalności aplikacji

**Autoryzacja użytkownika Rejestracja i logowanie:**

- Formularz rejestracji nowego użytkownika
- Formularz logowania z autoryzacją JWT
- Token JWT przechowywany w localStorage
- Automatyczne przekierowanie do Dashboard po zalogowaniu

**Dashboard - zarządzanie urządzeniami Główny ekran aplikacji:**

- **Wyświetlanie listy urządzeń** - wszystkie urządzenia przypisane do użytkownika
- **Grupowanie według lokalizacji** - urządzenia pogrupowane według pomieszczeń
- **Status online/offline** - wizualny wskaźnik (zielona/szara kropka) pokazujący stan połączenia urządzenia
- **Tworzenie lokalizacji** - możliwość dodania nowych pomieszczeń (np. “Magazyn A”, “Sala B”)
- **Claimowanie urządzeń** - ręczne przypisanie urządzenia przez wprowadzenie MAC address i PoP
- **Usuwanie urządzeń i lokalizacji** - zarządzanie zasobami z potwierdzeniem akcji

**Funkcje:**

- Odświeżanie listy urządzeń
- Nawigacja do szczegółów urządzenia (kliknięcie na karcie)
- Filtrowanie urządzeń według lokalizacji

**Sterowanie urządzeniem  Wyświetlanie danych telemetrycznych:**

- **Temperatura** - aktualna wartość w °C
- **Wilgotność** - aktualna wartość w %
- **Ciśnienie** - aktualna wartość w hPa
- **Wykrycie ruchu** - status (YES/NO)
- **Timestamp ostatniej aktualizacji** - czas ostatniego odczytu
- Automatyczne odświeżanie danych co 5 sekund (polling)

**Konfiguracja LED:**

- **Kolory RGB** - wybór 3 kolorów dla różnych progów wilgotności (picker kolorów)
- **Progi wilgotności** - 4 wartości progowe (0%, 30%, 70%, 100%)
- **Liczba aktywnych LED** - ustawienie liczby diod (1-5)
- **Próg wykrywania ruchu** - odległość w cm (10-400 cm)
- **Auto-brightness** - włączenie/wyłączenie automatycznej regulacji jasności
- **Jasność ręczna** - suwak do ustawienia jasności (0-100%) gdy auto-brightness wyłączone
- **Wysyłanie konfiguracji** - przycisk "Update Settings" wysyła konfigurację przez MQTT

**Aktualizacja firmware (OTA):**

- **Upload pliku firmware** - możliwość przesłania pliku `.bin` z nową wersją
- **Wersja firmware** - wprowadzenie numeru wersji (np. "1.0.1")
- **Automatyczna aktualizacja** - po przesłaniu, urządzenie automatycznie pobiera i instaluje nową wersję

**Provisioning urządzeń  Konfiguracja WiFi przez Web Bluetooth:**

1. **Skanowanie urządzeń BLE** - wyszukiwanie urządzeń w trybie provisioning
2. **Połączenie BLE** - łączenie się z urządzeniem przez Web Bluetooth API
3. **Odczyt danych urządzenia:**
    - MAC Address (6 bajtów) - wyświetlany użytkownikowi
    - Proof of Possession (PoP) - 32 bajty hex, wyświetlany użytkownikowi
4. **Wprowadzenie danych WiFi:**
    - SSID sieci WiFi
    - Hasło WiFi
    - Wybór lokalizacji (z listy istniejących)
    - Nazwa urządzenia
5. **Wysłanie konfiguracji** - zapis danych przez charakterystyki GATT
6. **Automatyczne claimowanie** - po wysłaniu konfiguracji, urządzenie jest automatycznie claimowane w backendzie
7. **Status operacji** - wyświetlanie aktualnego stanu procesu (Scanning, Connected, Config sent, etc.)

### 1.5.5 5.5 Aplikacja Mobilna

**Technologie:**

- **Framework:** React Native 0.81.5
- **Platforma:** Expo SDK ~54.0.31
- **Komunikacja BLE:** `react-native-ble-plx`
- **Platformy:** Android, iOS

#### 5.5.1 Funkcjonalności aplikacji

**Dashboard - zarządzanie urządzeniami  Wyświetlanie listy urządzeń:**

- Wszystkie urządzenia przypisane do użytkownika
- Filtrowanie według lokalizacji (pomieszczeń)
- Status online/offline każdego urządzenia (zielona/szara kropka)
- Odświeżanie listy przez pull-to-refresh

**Tworzenie i zarządzanie lokalizacjami:**

- Dodawanie nowych pomieszczeń (np. “Magazyn A”, “Sala B”)
- Usuwanie lokalizacji (long-press na pili lokalizacji)
- Filtrowanie urządzeń według wybranej lokalizacji

**Szczegóły urządzenia:**

- Wyświetlanie danych telemetrycznych w czasie rzeczywistym:
  - Temperatura (°C)
  - Wilgotność (%)
  - Ciśnienie (hPa)
  - Wykrycie ruchu (Detected/None)
- Automatyczne odświeżanie danych co 2 sekundy
- Nawigacja do ekranu konfiguracji urządzenia
- Usuwanie urządzenia z potwierdzeniem

**Provisioning urządzeń  Konfiguracja WiFi przez BLE:**

- **Skanowanie urządzeń BLE** - wyszukiwanie urządzeń w trybie provisioning
- **Połączenie z urządzeniem** - wybór urządzenia z listy znalezionych
- **Automatyczne odczytanie MAC Address** - urządzenie automatycznie odczytuje MAC z ESP32 przez BLE
- **Wprowadzenie danych:**
  - SSID sieci WiFi
  - Hasło WiFi
  - Nazwa urządzenia (opcjonalna)
  - Wybór lokalizacji (z listy istniejących lub utworzenie nowej)

- **Automatyczne claimowanie** - po odczycie MAC, urządzenie jest automatycznie rejestrowane w backendzie
- **Wysłanie konfiguracji** - zapis danych WiFi przez charakterystyki GATT BLE
- **Automatyczny restart** - po wysłaniu konfiguracji, ESP32 automatycznie restartuje się i łączy z WiFi

**Proces:**

1. Użytkownik wybiera "Add Device" z Dashboard.
2. Aplikacja skanuje urządzenia BLE (10 sekund).
3. Użytkownik wybiera urządzenie z listy.
4. Aplikacja łączy się i odczytuje MAC Address.
5. Użytkownik wprowadza dane WiFi i wybiera lokalizację.
6. Aplikacja automatycznie rejestruje urządzenie w backendzie (claim).
7. Aplikacja wysyła konfigurację WiFi przez BLE.
8. ESP32 restartuje się i łączy z siecią WiFi.

**Konfiguracja urządzenia  Konfiguracja LED:**

- **Kolory RGB** - wybór 3 kolorów dla różnych progów wilgotności (presety lub ręczne hex)
- **Progi wilgotności** - 4 wartości progowe (domyślnie: 30%, 50%, 70%, 90%, konfigurowalne)
- **Liczba aktywnych LED** - ustawienie liczby diod (1-5)

**Konfiguracja jasności:**

- **Auto-brightness** - włączenie/wyłączenie automatycznej regulacji jasności (na podstawie fotorezystora)
- **Jasność ręczna** - suwak do ustawienia jasności (0-100%) gdy auto-brightness wyłączone

**Konfiguracja czujników:**

- **Próg wykrywania ruchu** - odległość w cm (10-400 cm)

**Wysyłanie konfiguracji:**

- Przycisk "Save Configuration" wysyła konfigurację przez REST API backendu
- Backend przesyła komendę `SET_CONFIG` przez MQTT do ESP32
- ESP32 aktualizuje konfigurację w NVS i stosuje nowe ustawienia

### 5.5.2 Połączenie z systemem IoT  Komunikacja z backendem:

- Aplikacja mobilna komunikuje się z backendem przez REST API
- Endpointy identyczne jak w aplikacji webowej (zarządzanie urządzeniami, telemetria, lokalizacje)
- Backend host konfigurowalny (domyślnie: `http://192.168.0.186:8080`)

**Komunikacja z ESP32:**

- **Provisioning:** Bezpośrednia komunikacja BLE (GATT) - skanowanie, połączenie, odczyt/zapis charakterystyk
- **Konfiguracja:** Pośrednia przez backend - Aplikacja → REST API → MQTT → ESP32
- **Telemetria:** Pośrednia przez backend - ESP32 → MQTT → Backend → REST API → Aplikacja mobilna

**Uprawnienia:**

- **Android:** Wymagane uprawnienia Bluetooth i lokalizacji (dla skanowania BLE)
- **iOS:** Wymagane uprawnienia Bluetooth (automatycznie żądane przy pierwszym użyciu)

## 1.6 Podsumowanie

System Inteligentnego Oświetlenia LED z Monitorowaniem Wilgotności to kompleksowe rozwiązanie IoT przeznaczone dla magazynów zbiorów w archiwach, bibliotekach i muzeach. System zapewnia stałą kontrolę wilgotności, szybką sygnalizację stanu przy wejściu pracownika oraz pełny zapis historyczny do audytu.

**Kluczowe cechy:**

- Automatyczna sygnalizacja wizualna na podstawie wilgotności
- Wykrywanie ruchu z adaptacyjnym oświetleniem LED
- Komunikacja bezprzewodowa (WiFi, MQTT, BLE)
- Zasilanie autonomiczne (akumulator + panel solarny)
- Oszczędzanie energii (tryb uśpienia przy braku ruchu)
- Aktualizacje OTA (Over-The-Air)
- Pełny zapis danych do audytu

**Architektura:**

- **Warstwa urządzeń:** ESP32 z czujnikami i LED
- **Warstwa komunikacyjna:** MQTT (AWS IoT Core w chmurze)
- **Warstwa serwerowa:** Spring Boot (Java) + PostgreSQL
- **Warstwa kliencka:** Aplikacja webowa (React PWA) + Aplikacja mobilna (React Native)