# Politechnika Wrocławska

**Wydział Informatyki i Telekomunikacji**


---

Kierunek: **Informatyka Techniczna (ITE)**

Specjalność: **Inżynieria Systemów Informatycznych (INS)**

# PRACA DYPLOMOWA INŻYNIERSKA

## Inteligentny system wspomagający opiekę nad osobą starszą lub niepełnosprawną z użyciem technologii IoT i Raspberry Pi


Ivan Hancharyk

Opiekun pracy: **dr inż. Dominik Żelazny**



Słowa kluczowe: IoT, MQTT, ESP8266, czujnik ruchu PIR, Raspberry Pi


---

**WROCŁAW 2025**

# STRESZCZENIE

Celem pracy inżynierskiej było zaprojektowanie i implementacja inteligentnego systemu wspomagającego zdalną opiekę nad osobą starszą lub niepełnosprawną w środowisku domowym. System integrował czujniki ruchu PIR z mikrokontrolerem ESP8266 oraz lokalnym serwerem opartym na platformie Raspberry Pi, na którym uruchomiono brokera MQTT *Mosquitto*, backend w technologii *Flask* oraz bazę danych *SQLite*.

Aplikacja webowa, opracowana w środowisku *React*, umożliwiała przegląd i analizę zarejestrowanych danych o aktywności użytkownika, prezentując historię zdarzeń z podziałem na pomieszczenia oraz czas ich wystąpienia. System nie działał w trybie czasu rzeczywistego, lecz wykorzystywał dane zapisane w lokalnej bazie *SQLite*, które były okresowo aktualizowane przez usługę serwera *Flask*. Takie podejście do przetwarzania danych pozwoliło na odtworzenie wzorców zachowań oraz wykrywanie nietypowych sytuacji, takich jak długotrwały brak ruchu w określonym pomieszczeniu lub aktywność poza typowymi godzinami.

Zrealizowane rozwiązanie charakteryzowało się modularną budową i możliwością łatwego rozszerzania o dodatkowe czujniki oraz pomieszczenia. System stanowił praktyczny przykład zastosowania technologii Internetu Rzeczy (IoT) do wsparcia opieki nad osobami starszymi lub niepełnosprawnymi. Łączył elementy inżynierii systemowej, analizy danych oraz projektowania interfejsów użytkownika, tworząc spójne i możliwe do wdrożenia narzędzie służące do monitorowania codziennej aktywności domowej.

# ABSTRACT

The objective of this engineering thesis was to design and implement an intelligent home assistance system that supported remote monitoring and care for elderly or disabled people in a domestic environment. The system integrated a PIR motion sensor connected to an ESP8266 microcontroller and a local Raspberry Pi server running an MQTT broker (*Mosquitto*), a *Flask*-based backend, and an *SQLite* database.

The web application, developed in the *React* environment, enabled browsing and analysis of recorded user activity data, presenting a chronological history of events divided by rooms and timestamps of their occurrence. The system did not operate in real time; instead, it relied on data stored in a local *SQLite* database, which were periodically updated by a *Flask*-based server service. This processing approach allowed the reconstruction of behavioral patterns and the detection of unusual situations, such as prolonged inactivity in a given room or activity outside typical daily hours.

The implemented solution featured a modular architecture and could be easily extended with additional sensors and rooms. The system represented a practical example of applying Internet of Things (IoT) technologies to support the care of elderly or disabled individuals. It combined aspects of system engineering, data analysis, and user interface design to create an integrated, deployable tool for monitoring everyday domestic activity.

# Spis treści

# Wstęp

Ograniczona dostępność stałej opieki nad osobami starszymi oraz wysokie koszty usług opiekuńczych spowodowały potrzebę opracowania rozwiązań umożliwiających bezpieczne i zdalne wsparcie w warunkach domowych. Zastosowanie technologii informatycznych pozwoliło na tworzenie systemów, które nie zastępują człowieka, lecz wspierają opiekunów poprzez dostarczanie wiarygodnych informacji o zachowaniu użytkownika oraz wczesne wykrywanie potencjalnie niebezpiecznych sytuacji, takich jak długotrwały brak ruchu w pomieszczeniu czy aktywność w nietypowych godzinach nocnych. Dzięki takim rozwiązaniom osoby sprawujące opiekę mogą skupić się na własnych obowiązkach, zachowując pewność, że system automatycznie poinformuje je o niepokojących zdarzeniach.

Rozwój koncepcji Internetu Rzeczy (ang. *Internet of Things*, IoT) umożliwił tworzenie niedrogich, energooszczędnych i łatwych w instalacji systemów czujnikowych, które mogą funkcjonować bez konieczności ciągłej obsługi przez użytkownika. Wykorzystanie mikrokontrolerów z łącznością Wi-Fi, protokołu komunikacyjnego MQTT oraz lokalnych baz danych pozwoliło na budowę skalowalnych i elastycznych systemów przetwarzania danych, łączących prostotę sprzętu z możliwościami analizy informacji po stronie serwera. Dzięki temu technologia IoT zaczęła odgrywać ważną rolę w podnoszeniu jakości życia osób starszych i niepełnosprawnych, umożliwiając tworzenie inteligentnych, samodzielnych systemów monitorowania aktywności domowej.

## Cel i zakres pracy

Celem pracy było zaprojektowanie, wykonanie i przetestowanie inteligentnego systemu Internetu Rzeczy, wspomagającego zdalny nadzór nad aktywnością domową osoby starszej lub niepełnosprawnej. System umożliwia niezawodne pozyskiwanie danych z czujnika ruchu PIR, ich zapis w lokalnej bazie danych, analizę z wykorzystaniem reguł oraz prezentację wyników za pośrednictwem interfejsu REST API i panelu webowego.

Zaprojektowane rozwiązanie obejmuje kompletny łańcuch przetwarzania danych — od mikrokontrolera ESP8266 współpracującego z czujnikiem PIR, przez komunikację z brokerem MQTT, aż do serwera aplikacyjnego *Flask* i interfejsu użytkownika opartego na *React*. System został uruchomiony na komputerze Raspberry Pi i działa w sieci lokalnej, zapewniając pełną autonomię oraz prywatność danych użytkownika.

Zakres pracy obejmował projekt, implementację oraz analizę działania systemu, którego strukturę przedstawiono w siedmiu rozdziałach.

W rozdziale 1 zaprezentowano podstawy teoretyczne oraz przegląd istniejących rozwiązań z zakresu Internetu Rzeczy i technologii monitorowania aktywności. Opisano architekturę typowego systemu IoT, zasadę działania czujników PIR oraz kwestie prywatności w monitoringu domowym.

Rozdział 2 przedstawia koncepcję projektu systemu — jego architekturę logiczną, model danych, zastosowane reguły detekcji oraz założenia dotyczące interfejsu użytkownika.

W rozdziale 3 opisano proces implementacji poszczególnych komponentów, obejmujący oprogramowanie mikrokontrolera ESP8266, moduł loggera MQTT, silnik reguł, warstwę serwerową *Flask* oraz interfejs webowy opracowany w React.

Rozdział 4 dotyczy wdrożenia i utrzymania systemu na platformie Raspberry Pi. Omówiono instalację, konfigurację usług systemowych oraz metody monitorowania i konserwacji działania systemu.

W rozdziale 5 opisano testy funkcjonalne i symulacyjne, analizę stabilności systemu, wyniki testów w warunkach rzeczywistych oraz wnioski z ich realizacji.

Rozdział 6 porusza zagadnienia bezpieczeństwa i ochrony danych, w tym autoryzację komunikacji MQTT, zabezpieczenia serwera *Flask* oraz aspekty etyczne i prawne dotyczące przetwarzania informacji.

Rozdział 7 zawiera analizę użyteczności systemu, ocenę ergonomii interfejsu webowego, dostępność rozwiązania dla osób starszych oraz kierunki dalszego rozwoju projektu. Całość kończy podsumowanie, w którym przedstawiono wnioski końcowe i ocenę przydatności zaprojektowanego rozwiązania.

# Rozdział 1

# Podstawy i przegląd rozwiązań

Rozdział ten przedstawia podstawy teoretyczne oraz przegląd wybranych rozwiązań z zakresu Internetu Rzeczy (IoT, ang. *Internet of Things*) i technologii monitorowania aktywności domowej. Omówiono w nim architekturę typowego systemu IoT, metody detekcji ruchu, protokół komunikacyjny MQTT oraz zagadnienia prywatności w kontekście gromadzenia i przetwarzania danych. Opisane zagadnienia stanowiły punkt wyjścia do opracowania koncepcji oraz implementacji autorskiego systemu zdalnego nadzoru aktywności.

## 1.1 Architektura IoT

Internet Rzeczy (IoT) stanowi jedno z kluczowych podejść w projektowaniu nowoczesnych systemów automatyzacji i monitorowania. Architektura typowego systemu IoT opiera się na trzech zasadniczych warstwach: sensorycznej, komunikacyjnej i aplikacyjnej [1]. Warstwa sensoryczna obejmuje urządzenia brzegowe, które zbierały dane z otoczenia — najczęściej przy użyciu czujników środowiskowych, ruchu lub stanu. Warstwa komunikacyjna zapewniała przesył danych z wykorzystaniem lekkich protokołów sieciowych (np. MQTT, CoAP), a warstwa aplikacyjna odpowiadała za ich analizę, prezentację i interakcję z użytkownikiem.

W zaprojektowanym systemie rolę urządzenia brzegowego pełnił mikrokontroler ESP8266 połączony z czujnikiem ruchu PIR. Moduł ten, wyposażony w łączność Wi-Fi, wysyłał dane do lokalnego serwera uruchomionego na komputerze jednopłytkowym Raspberry Pi. Na serwerze działał broker komunikatów *Mosquitto* [2], aplikacja serwerowa w technologii *Flask* oraz lokalna baza danych *SQLite*. Taka architektura zapewniała pełną autonomię działania bez potrzeby korzystania z usług chmurowych. Dane nie opuszczały sieci lokalnej, co podnosiło poziom prywatności i niezależności systemu, tworząc kompletny model typu *end-to-end*.

## 1.2 Metody monitorowania aktywności

Systemy nadzoru domowego i wspomagające opiekę wykorzystywały wiele typów czujników. Najprostsze rozwiązania bazowały na detekcji ruchu, otwarcia drzwi, nacisku lub obecności. Bardziej zaawansowane systemy stosowały kamery wizyjne, radary mikrofalowe lub analizę dźwięku, jednak takie metody wymagały dużych zasobów obliczeniowych i często naruszały prywatność użytkownika.

Czujniki PIR (ang. *Passive Infrared Sensor*) działały na zasadzie pomiaru promieniowania podczerwonego emitowanego przez obiekty cieplejsze od otoczenia. Detektor składał się z elementu piroelektrycznego oraz soczewki Fresnela, która segmentowała pole widzenia na kilka stref. Zmiana rozkładu promieniowania w czasie (np. podczas przejścia człowieka przez jedną ze stref) powodowała chwilowy wzrost napięcia na wyjściu czujnika. Sygnał ten był następnie wzmacniany i filtrowany, a jego poziom można było regulować za pomocą potencjometrów czułości i czasu trwania impulsu.

W opracowanym systemie zastosowano popularny moduł HC-SR501 współpracujący z mikrokontrolerem ESP8266 [3]. Dane o stanie ruchu (*motion = true/false*) publikowano cyklicznie za pomocą protokołu MQTT, co umożliwiało ich analizę i klasyfikację przez silnik reguł. Takie rozwiązanie okazało się tanie i energooszczędne.

## 1.3 MQTT i broker komunikatów

MQTT (ang. *Message Queuing Telemetry Transport*) jest lekkim, dwukierunkowym protokołem komunikacyjnym zaprojektowanym do przesyłania niewielkich pakietów danych w środowiskach o ograniczonych zasobach. Wykorzystuje model *publish–subscribe*, w którym komunikacja między nadawcami a odbiorcami odbywa się za pośrednictwem centralnego brokera. Urządzenia publikują dane w określonych tematach (*topics*), a odbiorcy subskrybują je, aby otrzymywać aktualizacje w czasie rzeczywistym [2].

Każda wiadomość w MQTT posiada poziom jakości dostarczania (*Quality of Service, QoS*), który określa sposób potwierdzania transmisji. Wyróżnia się trzy poziomy: – **QoS 0** — wiadomość była wysyłana bez potwierdzenia, – **QoS 1** — nadawca oczekiwał potwierdzenia od brokera, – **QoS 2** — pełna dwustronna wymiana potwierdzeń.

W zaprojektowanym systemie zastosowano poziom QoS 1, który zapewniał równowagę między niezawodnością a szybkością transmisji. Dodatkowo broker MQTT obsługiwał tzw. *retained messages* — ostatnią znaną wartość dla każdego tematu, co pozwalało odtworzyć aktualny stan po restarcie systemu. Zastosowano również mechanizm *keep-alive*, który umożliwiał wykrywanie utraty łączności z urządzeniem (reguła *NO_HEARTBEAT*).

## 1.4 Prywatność w monitoringu domowym

Ochrona prywatności stanowi jedno z najważniejszych wyzwań w projektowaniu systemów monitoringu. Nadmierna szczegółowość danych mogła prowadzić do ujawnienia wzorców zachowań lub informacji osobistych. Z tego względu przyjęto minimalny zakres gromadzonych informacji — jedynie zdarzenia binarne (ruch/brak ruchu) oraz dane techniczne urządzeń.

Dane były przechowywane lokalnie w bazie *SQLite* na serwerze Raspberry Pi, bez przesyłania do zewnętrznych usług. Dostęp do API systemu *Flask* ograniczono kluczem autoryzacyjnym, a broker *Mosquitto* wykorzystywał podstawową autoryzację użytkowników. Brak integracji z chmurą eliminował ryzyko nieautoryzowanego dostępu z Internetu. System został zaprojektowany jako zamknięty, działający w sieci domowej (LAN), z naciskiem na prostotę, bezpieczeństwo i pełną kontrolę nad danymi użytkownika.

# Rozdział 2

# Projekt systemu

Rozdział ten przedstawia założenia projektowe, architekturę oraz główne komponenty opracowanego systemu Internetu Rzeczy (IoT) wspomagającego zdalny nadzór nad aktywnością domową osoby starszej lub niepełnosprawnej. Omówiono w nim wymagania funkcjonalne i niefunkcjonalne, strukturę architektury logicznej, model danych, reguły detekcji oraz koncepcję interfejsu użytkownika. Opisane rozwiązania stanowiły podstawę implementacji i testów przedstawionych w kolejnych rozdziałach.

## 2.1 Wymagania

Podstawowym celem projektowanego systemu było umożliwienie zdalnego monitorowania aktywności osoby starszej lub niepełnosprawnej w środowisku domowym przy zachowaniu niskich kosztów wdrożenia i prostoty obsługi. System miał działać w sposób ciągły, automatyczny i nieinwazyjny, bez potrzeby stałej interwencji użytkownika.

Na podstawie analizy potrzeb użytkowników i warunków środowiskowych określono wymagania funkcjonalne, czyli te cechy systemu, które decydowały o jego działaniu i przydatności. System rejestrował zdarzenia ruchu z czujników PIR oraz wysyłał okresowe sygnały aktywności urządzeń (*heartbeat*), pozwalające na monitorowanie ich stanu. Dane przesyłano do lokalnego serwera za pośrednictwem protokołu MQTT, gdzie były zapisywane w bazie danych *SQLite*. Silnik reguł analizował zapisane informacje i wykrywał stany alarmowe, takie jak *INACTIVITY*, *DWELL_CRITICAL* oraz *NO_HEARTBEAT*. Wyniki przetwarzania udostępniano poprzez interfejs REST API, a użytkownik mógł je obserwować w aplikacji webowej w formie czytelnych wizualizacji. W systemie zastosowano także funkcję sygnalizacji pre-alert, realizowaną diodą LED, ostrzegającą o zbliżającym się stanie krytycznym.

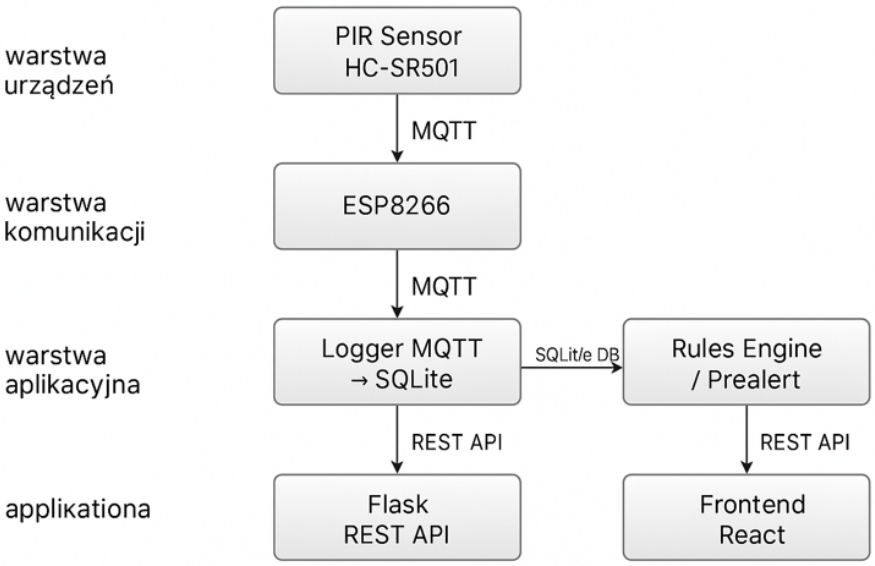
Wymagania niefunkcjonalne odnosiły się do sposobu działania, niezawodności i bezpieczeństwa systemu. Całość działała w sieci lokalnej (LAN) bez połączenia z chmurą publiczną, co zapewniało pełną prywatność. Komunikacja pomiędzy modułami była odporna na krótkotrwałe przerwy w transmisji, a wszystkie komponenty bazowały na technologiach open-source. Dane użytkownika nie były przesyłane poza urządzenie lokalne, a interfejs webowy został zaprojektowany tak, aby był prosty, intuicyjny i dostępny z poziomu przeglądarki. Architektura umożliwiała również łatwe dodawanie kolejnych czujników i pomieszczeń. Serwer pracował stabilnie w trybie ciągłym na komputerze Raspberry Pi.

Zdefiniowane wymagania stanowiły podstawę dalszego projektowania architektury, modelu danych oraz logiki przetwarzania zdarzeń, przedstawionych w kolejnych podrozdziałach.

## 2.2 Architektura i komponenty

Zaprojektowany system składał się z pięciu logicznych komponentów obejmujących cały cykl przetwarzania danych: urządzenia brzegowego (ESP8266 z czujnikiem PIR), brokera komunikatów (*Mosquitto*), loggera MQTT w języku Python, silnika reguł oraz interfejsu użytkownika opartego na *Flask* i *React*. Wszystkie moduły tworzyły spójną architekturę typu *end-to-end*, w której dane przepływały od czujnika ruchu do aplikacji webowej w sposób ciągły i kontrolowany.

Na Rys. 1 przedstawiono blokowy diagram architektury systemu IoT. Schemat ilustrował przepływ informacji pomiędzy urządzeniem brzegowym, brokerem MQTT, serwerem, bazą danych i warstwą prezentacji.




Rysunek 1: Diagram blokowy architektury systemu IoT.

Broker MQTT działał na porcie 1883 i wykorzystywał prostą autentykację (login/hasło). Komunikaty publikowano w trzech głównych tematach:

`iot/{room}/motion` zdarzenia detekcji ruchu

`iot/{room}/heartbeat` sygnały stanu urządzenia

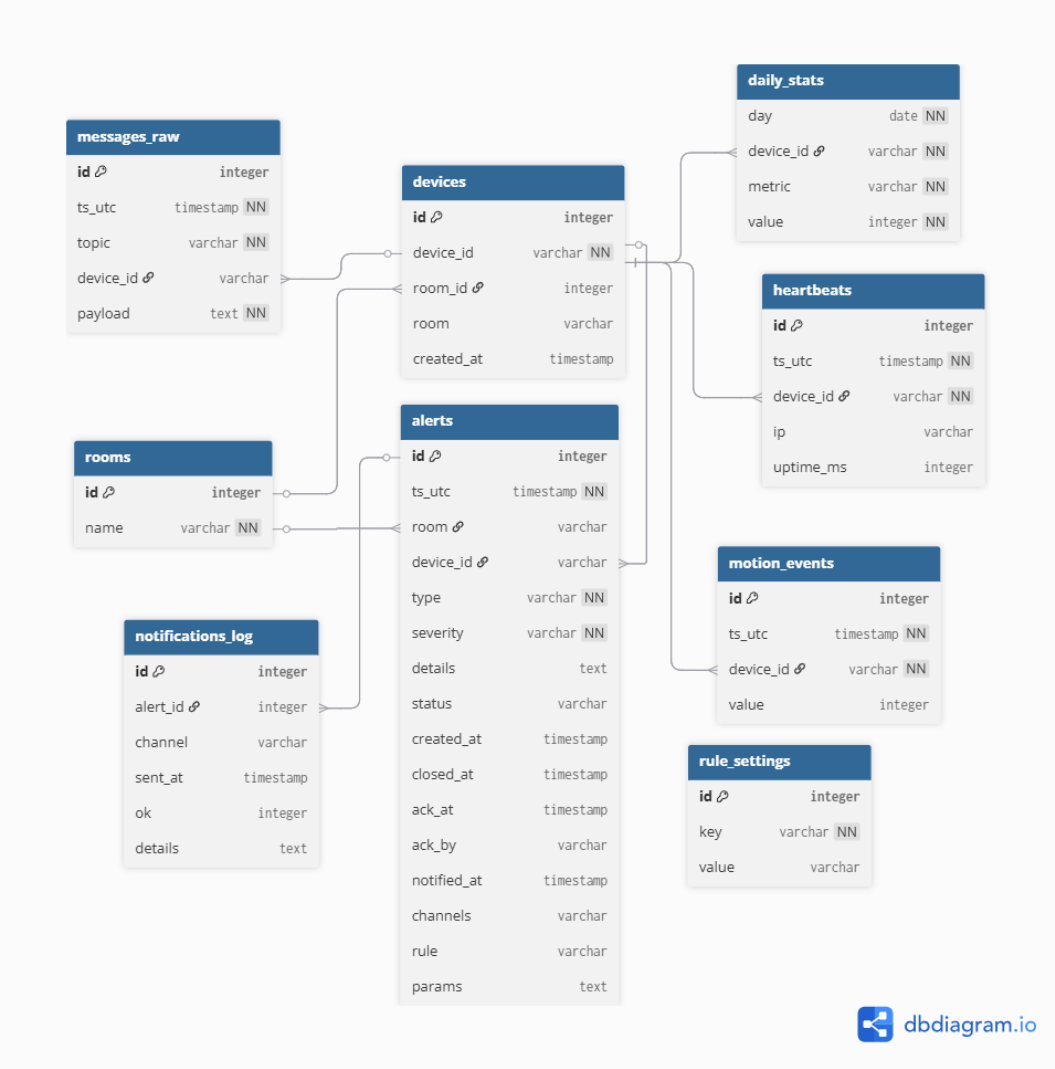
`iot/{room}/command` komendy zwrotne (np. sterowanie diodą LED)

Serwer *Flask* udostępniał interfejs REST API, z którego korzysta frontend React. Dzięki temu możliwe jest przeglądanie listy alertów, historii ruchu oraz stanu urządzeń w czasie rzeczywistym. System nie wymaga połączenia z Internetem — wszystkie komponenty działają w ramach jednej sieci lokalnej.

## 2.3 Model danych

Dane przechowywano w lokalnej bazie *SQLite*, ze względu na jej prostotę, niskie wymagania i niezawodność. Struktura bazy została zaprojektowana tak, aby umożliwiała rozbudowę o kolejne czujniki, pomieszczenia i reguły. W modelu danych ujęto tabele rejestrujące urządzenia, pomieszczenia, zdarzenia ruchu, sygnały heartbeat, alerty, ustawienia reguł oraz dziennik powiadomień. Każdy wpis w tabeli `alerts` zawierał informacje o rodzaju wykrytej anomalii, czasie jej wystąpienia, statusie (aktywna/zamknięta) oraz powiązaniu z urządzeniem i pomieszczeniem, co umożliwiało efektywne raportowanie i filtrację wyników w interfejsie webowym.

Na Rys. 2 przedstawiono schemat relacyjny bazy danych systemu IoT, pokazujący zależności pomiędzy poszczególnymi tabelami i ich atrybutami.




Rysunek 2: Schemat relacyjny bazy danych systemu IoT.

## 2.4 Reguły detekcji

Silnik reguł (*rules_ engine.py*) działa cyklicznie w tle i analizuje zarejestrowane dane w celu wykrywania nietypowych zachowań. Zaimplementowano trzy podstawowe reguły: *INACTIVITY*, *DWELL_ CRITICAL* oraz *NO_ HEARTBEAT*. Zestawienie ich przykładowych progów czasowych przedstawiono w Tabeli 1.

**Reguła INACTIVITY** — identyfikuje brak ruchu w pomieszczeniu przez określony czas. Jeśli w zadanym przedziale (np. 30 minut w dzień, 60 minut w nocy) nie odnotowano zdarzeń ruchu, system generuje alert o potencjalnej bezczynności.

**Reguła DWELL_CRITICAL** — wykrywa nadmiernie długą obecność w jednym pomieszczeniu. Służy do identyfikacji sytuacji potencjalnie niebezpiecznych, takich jak utrata przytomności w łazience. Próg czasowy można definiować osobno dla dnia i nocy.

**Reguła NO_HEARTBEAT** — reaguje na brak sygnałów *heartbeat* z urządzenia. Jeśli w ciągu określonego czasu (np. 5 minut) nie nadejdzie żaden sygnał, system zakłada utratę łączności z czujnikiem i tworzy alert techniczny.

Tabela 1: Przykładowe progi czasowe wykorzystywane przez silnik reguł.

| **Reguła** | **Opis** | **Próg dzienny** | **Próg nocny** |
|---|---|---|---|
| `INACTIVITY` | Brak ruchu w pomieszczeniu | 60 min | – |
| `DWELL_CRITICAL` | Długie przebywanie w strefie krytycznej | 60–90 min | 20–45 min |
| `NO_HEARTBEAT` | Brak sygnału od urządzenia | 15 min | 15 min |

Parametry reguł są definiowane dynamicznie w tabeli `rule_settings`, co umożliwia modyfikację progów i okien czasowych bez ingerencji w kod źródłowy. Każda reguła ma przypisany zestaw akcji (np. wysłanie powiadomienia e-mail, zapalenie diody LED lub zapis wpisu w logu systemowym), co zapewnia elastyczność i rozszerzalność systemu.

## 2.5 Interfejs użytkownika

Warstwa frontendowa została opracowana w technologii *React*, z wykorzystaniem biblioteki *TailwindCSS* do stylizacji oraz narzędzia *React Query* do odświeżania danych. Panel użytkownika obejmował sześć głównych widoków. Widok **Dashboard** prezentował podsumowanie bieżących statystyk systemu, takich jak liczba aktywnych urządzeń, alertów oraz ostatnie zarejestrowane zdarzenia. Sekcja **Alerts** umożliwiała przeglądanie aktywnych i archiwalnych powiadomień z możliwością ich filtrowania. W zakładce **Devices** znajdował się rejestr czujników i ich statusów, obejmujący ostatni sygnał *heartbeat*, stan diody LED oraz przypisanie do pomieszczeń. Widok **Rooms/Map** przedstawiał plan pomieszczeń z rozmieszczeniem czujników i aktualnym stanem aktywności w formie graficznej mapy mieszkania. Sekcja **History** pozwalała analizować zarejestrowane komunikaty MQTT oraz sygnały *heartbeat* w porządku chronologicznym, a także eksportować dane do pliku CSV. Ostatni widok, **Settings**, służył do konfiguracji parametrów reguł oraz interwałów odświeżania danych.

Widok **Rooms/Map** został zaprojektowany jako element interaktywny, pozwalający na wizualne przedstawienie rozmieszczenia czujników w pomieszczeniach. Na obecnym etapie pełni on funkcję demonstracyjną, stanowiąc podstawę do dalszego rozwoju w kierunku bardziej zaawansowanej wizualizacji przestrzennej oraz integracji z rzeczywistym planem mieszkania.

Panel React komunikuje się z serwerem Flask poprzez żądania REST API. Dane są okresowo odświeżane w trybie *polling* co kilka sekund, co zapewnia aktualność przy minimalnym obciążeniu serwera. Wszystkie widoki mają ciemny i jasny motyw, zaprojektowany z myślą o czytelności i prostocie.

Zastosowane rozwiązania projektowe zapewniają wysoką elastyczność interfejsu oraz możliwość jego rozbudowy w przyszłości. Architektura umożliwia wprowadzanie nowych funkcji bez zmian w strukturze danych czy logice komunikacji, co czyni system skalowalnym i przystosowanym do dalszego rozwoju.

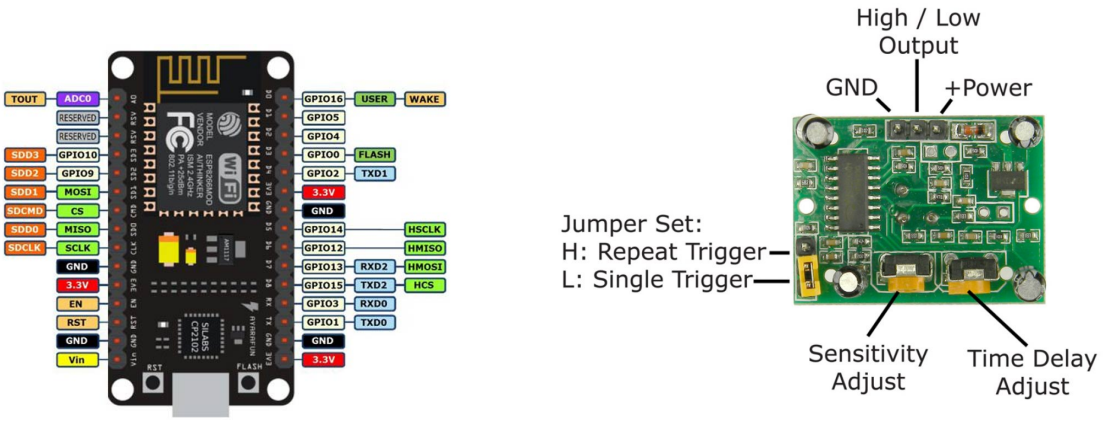
# Rozdział 3

# Implementacja

Rozdział ten przedstawia sposób realizacji zaprojektowanego systemu, obejmujący jego warstwę sprzętową, oprogramowanie mikrokontrolera, komponenty serwerowe oraz interfejs użytkownika. Opisano również mechanizm rejestracji danych, analizę reguł i symulację działania systemu w środowisku testowym.

## 3.1 Hardware i firmware ESP8266

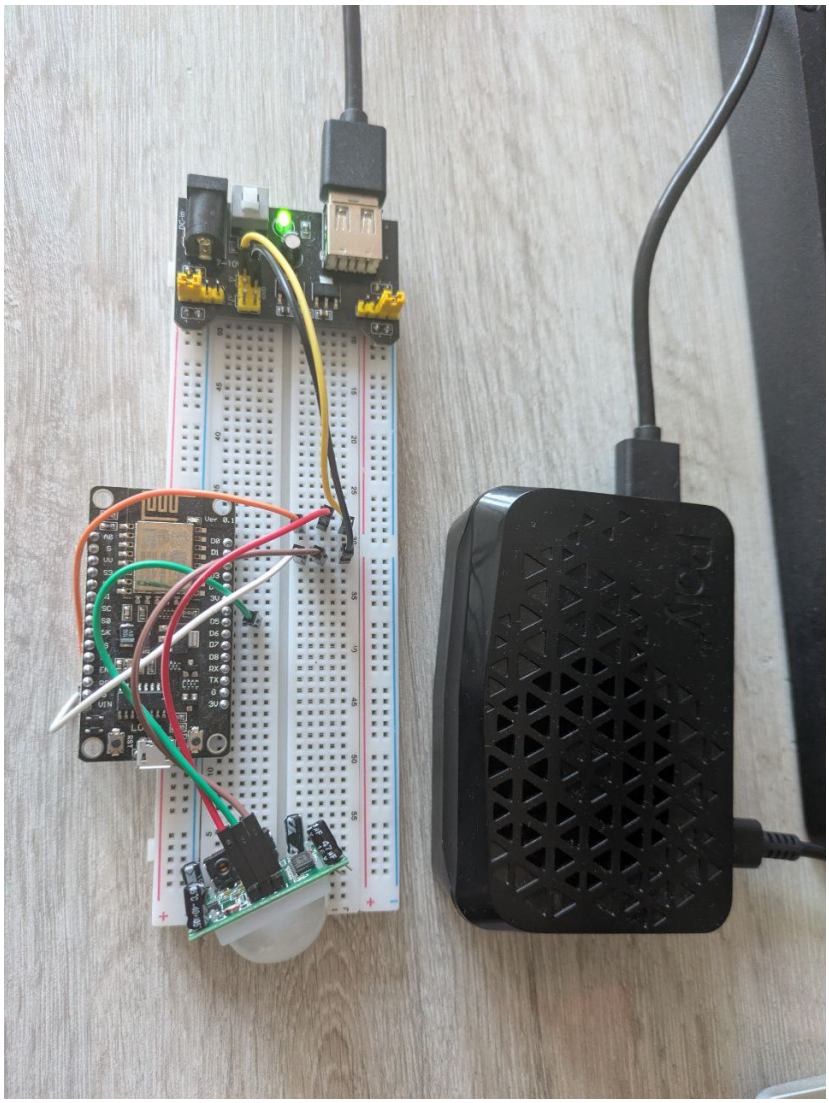
Część sprzętowa systemu obejmowała dwa podstawowe elementy: moduł czujnika ruchu zbudowany na mikrokontrolerze *ESP8266 NodeMCU* oraz jednostkę centralną *Raspberry Pi 4*, pełniącą funkcję lokalnego serwera i pośrednika komunikacyjnego. Moduł czujnika został wyposażony w detektor *HC-SR501 PIR* połączony z mikrokontrolerem ESP8266 (Rys. 3). Układ zasilano napięciem 5 V, a sygnał wyjściowy z czujnika podłączono do pinu cyfrowego D5. Diodę LED sterowano z poziomu pinu D6, co umożliwiało uruchomienie funkcji *pre-alert*, czyli sygnału ostrzegawczego poprzedzającego wygenerowanie alertu. Zasilanie dostarczał moduł *breadboard power supply* z wyjściami 5 V / 3.3 V.




(a) Mikrokontroler ESP8266 NodeMCU (b) Czujnik ruchu HC-SR501

Rysunek 3: Schematy komponentów systemu IoT: (a) mikrokontroler ESP8266 NodeMCU, (b) czujnik ruchu HC-SR501. https://components101.com/..., https://www.ariat-tech.com/....

Zmontowany prototyp urządzenia zaprezentowano na Rys. 4. Układ zamontowano na płytce stykowej, a jego konstrukcja miała charakter eksperymentalny. W przyszłości całość mogłaby zostać zamknięta w dedykowanej obudowie wydrukowanej metodą 3D, co zwiększyłoby trwałość i estetykę urządzenia.



Rysunek 4: Zmontowany prototyp modułu czujnika ruchu na płytce stykowej.

Jednostka centralna systemu – *Raspberry Pi 4* – została umieszczona w obudowie chłodzonej aktywnie za pomocą miniaturowego wentylatora, który uruchamiał się automatycznie po przekroczeniu temperatury 55 °C (wartość progu można było regulować programowo). Zastosowana obudowa to model *Argon Poly Vented* z oferty Botland[1], łączący dobrą wentylację z estetycznym wyglądem i ochroną elementów elektronicznych.

Oprogramowanie mikrokontrolera opracowano w języku C++ z wykorzystaniem środowiska *PlatformIO*. Firmware realizował połączenie Wi-Fi, obsługę protokołu MQTT oraz odczyt danych z czujnika PIR. Główna pętla programu publikowała komunikaty JSON w formacie przedstawionym w Listing 1.

[1] https://botland.com.pl/obudowy-do-raspberry-pi-4b/17276-obudowa-do-raspberry-pi-4b-argon-poly-vented-z-wentylatorem-mini-fan-czarna-5904422327262.html

```
{"motion": true, "device": "esp8266_test", "timestamp":
   "2025-03-14T12:45:32"}
```

Listing 1: Przykładowy komunikat JSON publikowany przez mikrokontroler

W przypadku braku ruchu czujnik wysyłał wartość `false`, a okresowo również wiadomość typu `"heartbeat"` w celu potwierdzenia swojej aktywności. Firmware działał w sposób autonomiczny, uruchamiając się automatycznie po zasileniu urządzenia i utrzymując stabilne połączenie z brokerem MQTT w sieci lokalnej. Zaprojektowany system działał w sieci lokalnej i nie był przeznaczony do zastosowań medycznych ani alarmowych klasy przemysłowej. Nie rejestrował on dźwięku ani obrazu – gromadził wyłącznie dane techniczne i zdarzenia ruchu. Zakres pracy nie obejmował integracji z chmurą, uczenia maszynowego ani certyfikacji bezpieczeństwa, skupiając się na praktycznej weryfikacji koncepcji systemu IoT w warunkach domowych.

## 3.2 Logger i utrzymanie bazy danych

Na komputerze Raspberry Pi uruchomiono brokera *Mosquitto* oraz usługę Pythonową `mqtt_logger.py`, która odbierała komunikaty MQTT i zapisywała je w lokalnej bazie danych *SQLite*. Logger subskrybował tematy `iot/#`, rejestrując wszystkie wiadomości publikowane przez czujniki. Odebrane dane były następnie parsowane i umieszczane w odpowiednich tabelach: `messages_raw` (pełna treść komunikatu), `motion_events` (zdarzenia ruchu) oraz `heartbeats` (sygnały potwierdzające aktywność urządzeń).

Baza danych `events.db` była inicjalizowana automatycznie przy pierwszym uruchomieniu systemu. Zastosowano prosty mechanizm archiwizacji danych — starsze wpisy usuwano po przekroczeniu określonego wieku, co pozwalało utrzymać niską objętość bazy i zapewniało szybki odczyt podczas generowania raportów. Logger działał w trybie ciągłym, dzięki czemu system mógł funkcjonować bez nadzoru użytkownika.

## 3.3 Silnik reguł i analiza danych

Moduł `rules_engine.py` odpowiadał za analizę zgromadzonych danych i generowanie alertów. Silnik pracował cyklicznie co 15 sekund, odczytywał najnowsze wpisy z tabel `motion_events` oraz `heartbeats` i porównywał je z konfiguracją progów czasowych. Zaimplementowane reguły objęły trzy podstawowe przypadki: brak ruchu w pomieszczeniu (*INACTIVITY*), długie przebywanie w jednym miejscu (*DWELL_CRITICAL*) oraz utratę sygnału z urządzenia (*NO_HEARTBEAT*). Parametry działania, w tym progi dzienne i nocne oraz okno wstępnego ostrzeżenia, zostały pobrane z konfiguracji pokoju i umożliwiły dostrojenie zachowania systemu do specyfiki mieszkania.

Po przekroczeniu wyznaczonego progu nieaktywności silnik inicjował etap *pre-alert*: wysyłał polecenie do mikrokontrolera w celu uruchomienia krótkiej sygnalizacji świetlnej, która ostrzegała o zbliżającym się alercie. Jeżeli w czasie obowiązywania *pre-alertu* system odnotował ruch, ostrzeżenie zostało anulowane i nie dochodziło do eskalacji. Gdy brak aktywności utrzymywał się nadal, silnik otwierał właściwy alert w bazie danych i oznaczał go statusem *open*; po powrocie do stanu normalnego zamykał wpis, nadając mu status *closed*. Analogiczny mechanizm zadziałał dla zbyt długiego przebywania w strefie krytycznej oraz dla braku sygnału *heartbeat*, co pozwoliło wykrywać zarówno zdarzenia potencjalnie niebezpieczne dla użytkownika, jak i problemy techniczne z łącznością.

Silnik rozróżniał pory dnia i nocy, dzięki czemu reagował adekwatnie do kontekstu dobowego. Warunki nocne ograniczały przypadkowe wzbudzenia oraz skracały czas detekcji w pomieszczeniach o podwyższonym ryzyku. Wszystkie decyzje diagnostyczne zapisywano w dzienniku działania modułu, a otwarte alerty były niezwłocznie przekazywane do komponentu `notifier.py`, który wysyłał powiadomienia e-mail do opiekuna. Zastosowane podejście zapewniło przewidywalną reakcję systemu, pełną ścieżkę audytową oraz możliwość późniejszej analizy historii zdarzeń bez konieczności prezentowania kodu źródłowego w treści pracy.

## 3.4 Interfejs API (Flask)

Warstwa backendowa systemu została oparta na frameworku *Flask*, który zapewnił lekki i elastyczny serwer HTTP obsługujący komunikację pomiędzy bazą danych *SQLite* a aplikacją webową. Serwer uruchamiał się na porcie 5000 i udostępniał dane w formacie JSON, umożliwiając ich łatwe przetwarzanie po stronie interfejsu React. Dzięki architekturze REST możliwa była realizacja pełnego zestawu operacji — od pobierania i filtrowania danych, aż po ich modyfikację i administrację.

W celu zabezpieczenia dostępu do danych wprowadzono prosty mechanizm autoryzacji za pomocą klucza przesyłanego w nagłówku `X-API-Key`. Każde żądanie bez poprawnego klucza było automatycznie odrzucane z kodem błędu 401, co zapobiegało nieautoryzowanemu odczytowi danych. Serwer obsługiwał metody `GET`, `POST`, `PUT`, `DELETE` oraz `OPTIONS` (preflight), umożliwiając komunikację z frontendem poprzez zapytania asynchroniczne realizowane w bibliotekach *fetch* lub *Axios*. Włączono również mechanizm CORS, który pozwolił na bezpieczną wymianę danych między domenami.

Tabela 2 przedstawia zestawienie dostępnych endpointów REST API wraz z ich funkcjami i parametrami. Każdy z nich został zaprojektowany w sposób modularny, co umożliwiło łatwe rozszerzanie interfejsu o nowe funkcje bez ingerencji w istniejący kod.

Tabela 2: Zestawienie publicznych endpointów interfejsu REST API.

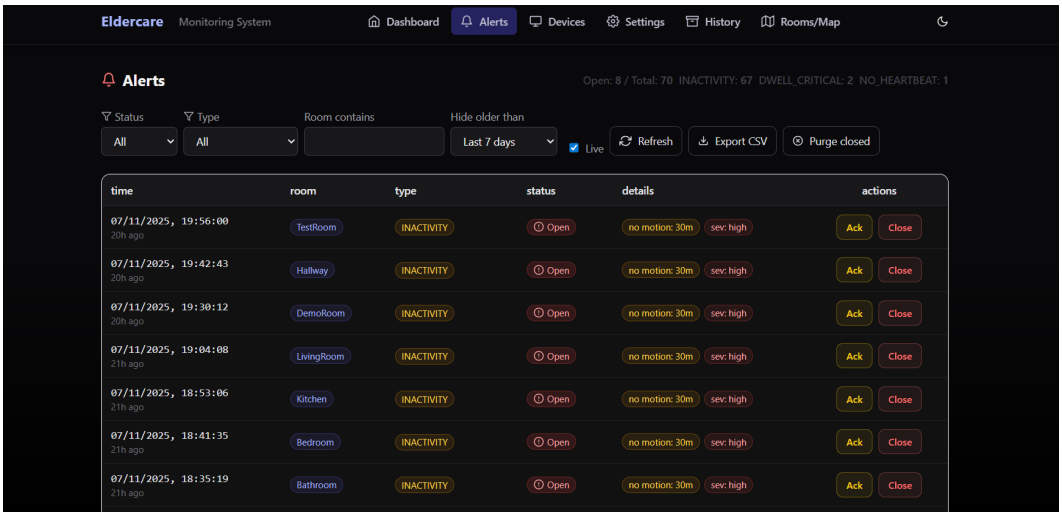
| **Endpoint** | **Metody** | **Opis / najważniejsze parametry** |
|---|---|---|
| `/` | GET | Prosty *health ping* API – potwierdzenie działania serwera. |
| `/api/messages` | GET | Pobiera ostatnie surowe wiadomości MQTT z tabeli `messages_raw`; parametr: `?limit`. |
| `/api/health/latest` | GET | Zwraca zestawienie ostatnich sygnałów *heartbeat* dla każdego urządzenia. |
| `/api/devices` | GET | Udostępnia listę urządzeń wraz z przypisanym pokojem i czasem ostatniej aktywności. |
| `/api/devices/register` | POST | Rejestrował nowe urządzenie lub aktualizował jego przypisanie do pokoju. |
| `/api/devices/{id}/unregister` | POST | Wypisywał urządzenie, usuwając jego przypisanie z bazy danych. |
| `/api/alerts` | GET | Wyświetlał listę alertów z możliwością filtrowania po statusie, pokoju, typie i dacie. |
| `/api/alerts/{id}/ack` | POST | Oznaczał alert jako potwierdzony przez opiekuna. |
| `/api/alerts/{id}/close` | POST | Zamykał wskazany alert po usunięciu przyczyny. |
| `/api/alerts/close-bulk` | POST | Umożliwiał zbiorcze zamykanie alertów starszych niż określony czas. |
| `/api/rule-settings` | GET, PUT, POST | Odczytywał i aktualizował konfigurację reguł działania systemu. |
| `/api/rooms` | GET | Udostępniał listę pokoi z informacjami o ostatnim ruchu i aktywnych alertach. |
| `/api/rooms/{room}/settings` | GET, POST | Pobierał lub zapisywał konfigurację pokoju (czasy progowe, tryb nocny, blokady). |
| `/api/rooms/{room}/led` | POST | Wysyłał polecenie MQTT do urządzenia w celu uruchomienia lub wyłączenia sygnału *pre-alert*. |
| `/api/events/recent` | GET | Zwracał zgrupowany przegląd ostatnich zdarzeń ruchu w systemie. |

Każdy z powyższych punktów końcowych zwracał dane w formacie JSON, co umożliwiło ich bezpośrednie wykorzystanie w aplikacji React. W przypadku błędnych żądań serwer zwracał odpowiedni kod HTTP oraz komunikat opisowy, co ułatwiało diagnostykę i utrzymanie systemu. Zaprojektowany interfejs stanowił kluczowy element komunikacji między warstwą analityczną a wizualizacyjną, gwarantując stabilność działania oraz możliwość dalszej rozbudowy o funkcje raportowania i zdalnego dostępu.

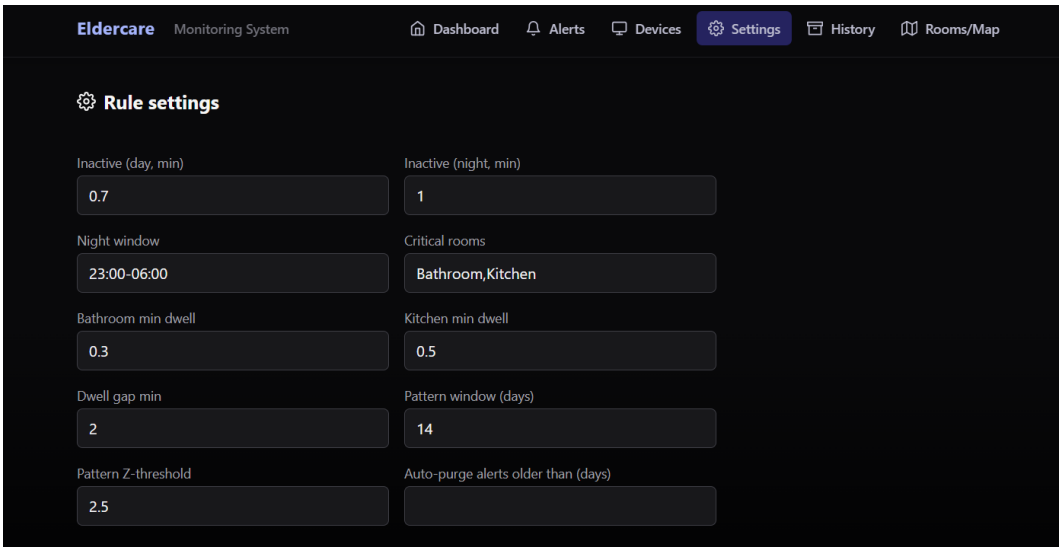
## 3.5 Frontend (React)

Aplikacja webowa została opracowana w technologii *React* z wykorzystaniem stylów *TailwindCSS*. Projekt ma charakter *Single Page Application (SPA)*, co oznacza, że przełączanie między widokami odbywa się bez przeładowywania strony.

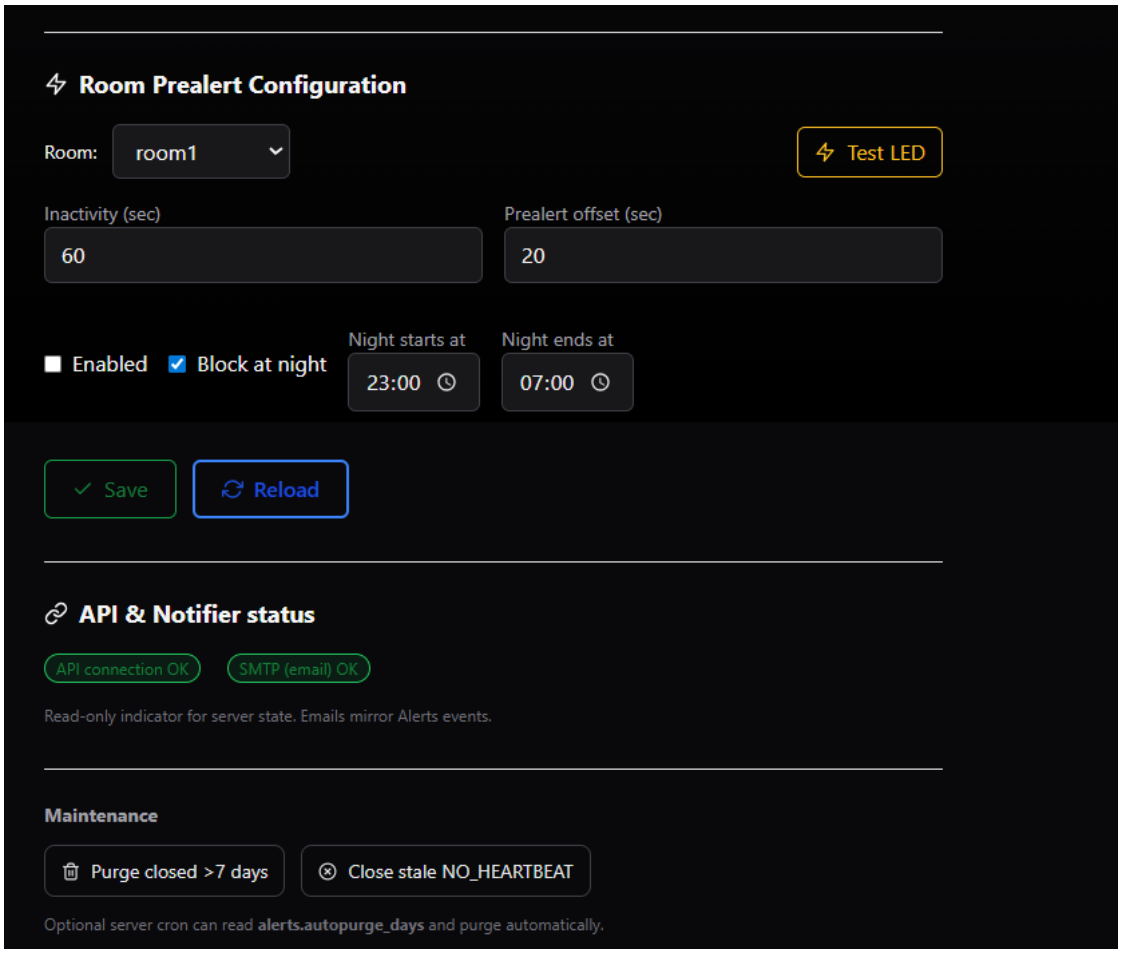
Do wizualizacji wykorzystano biblioteki *Recharts* (wykresy) oraz *ShadCN/UI* (komponenty interfejsu), co zapewnia spójny wygląd i responsywność. Frontend komunikuje się z backendem poprzez bibliotekę *Axios* i wykorzystuje *React Query* do cyklicznego odświeżania danych (*polling*). Na Rys. 5–7 przedstawiono najważniejsze ekrany aplikacji, obejmujące listę alertów oraz panel konfiguracji systemu.




Rysunek 5: Widok *Alerts* – lista aktywnych powiadomień systemowych.




Rysunek 6: Widok *Settings* – konfiguracja progów czasowych i reguł analizy.




Rysunek 7: Widok *Settings* – ustawienia LED, status

Interfejs umożliwiał podgląd bieżących zdarzeń, historii ruchu oraz zmianę parametrów reguł. Zastosowano jasny i ciemny motyw kolorystyczny, co poprawiało komfort użytkowania w różnych warunkach oświetleniowych. Aplikacja była w pełni responsywna i działała na dowolnym urządzeniu w sieci lokalnej.

## 3.6 Symulacja aktywności

W celu przetestowania działania całego systemu opracowano dwa skrypty testowe: `publish_sim.py` oraz `simulator_motion.py`. Wykorzystują one bibliotekę *paho-mqtt* do publikowania komunikatów w tematach `iot/eldercare/{room}/motion/state`, symulując zachowanie czujników PIR. Skrypty generują przykładowe zdarzenia ruchu oraz okresy braku aktywności, co pozwala testować reakcję silnika reguł i loggera na różne scenariusze.

Symulator umożliwia wybór profili aktywności (`morning`, `day`, `night`, `bathroom_long`, `kitchen_cooking`, `random`) oraz tryb ręczny, w którym można ręcznie wysyłać zdarzenia `on/off`. Każdy profil opisuje inny harmonogram ruchu, co pozwala analizować zachowanie systemu w warunkach zbliżonych do rzeczywistych.

Symulacja pozwoliła przetestować stabilność działania loggera, API oraz interfejsu webowego bez konieczności przeprowadzania dużej liczby testów fizycznych, co pozwoliło zaoszczędzić czas. Test w środowisku rzeczywistym również został wykonany i opisano go w kolejnym rozdziale dokumentacji. Skrypty uruchamiane są niezależnie od głównych usług systemowych na Raspberry Pi, dzięki czemu nie blokują pracy pozostałych komponentów systemu. Rules Engine i analiza danych

# Rozdział 4

# Wdrożenie i utrzymanie systemu

Rozdział ten przedstawia proces wdrożenia zaprojektowanego systemu Internetu Rzeczy (IoT) na platformie Raspberry Pi, konfigurację usług i środowiska uruchomieniowego, a także rozwiązania związane z jego utrzymaniem i monitoringiem. Opisano kolejne etapy instalacji oprogramowania, uruchamiania komponentów jako usług systemowych oraz procedury konserwacji, które zapewniły stabilne działanie systemu w trybie ciągłym.

## 4.1 Instalacja i konfiguracja na Raspberry Pi

Proces wdrożenia został przeprowadzony na komputerze jednopłytkowym Raspberry Pi 4B z systemem operacyjnym Raspberry Pi OS Lite (64-bit). Środowisko to zapewniało odpowiednią wydajność oraz niskie zużycie energii, dzięki czemu stanowiło optymalną platformę dla aplikacji klasy IoT.

Pierwszym etapem wdrożenia była konfiguracja sieci lokalnej oraz instalacja niezbędnych pakietów: `mosquitto`, `python3`, `pip`, `sqlite3`, `git` i `systemd`. Po pobraniu kodu źródłowego projektu z repozytorium zainstalowano wszystkie wymagane biblioteki Pythonowe, co umożliwiło poprawne działanie komponentów serwera i loggera danych. Pakiety niezbędne do uruchomienia systemu przedstawiono w Listing 2.

```
sudo apt update && sudo apt install mosquitto mosquitto-clients
   sqlite3 python3-pip
pip install paho-mqtt flask schedule
```

Listing 2: Instalacja pakietów i bibliotek wymaganych przez system

Następnie skonfigurowano brokera MQTT (*Mosquitto*) poprzez edycję pliku `/etc/mosquitto/mosquitto.conf`. Do konfiguracji dodano sekcję ograniczającą dostęp do lokalnych adresów IP oraz włączono autoryzację użytkownika. Dzięki temu system działał w pełni w sieci lokalnej, bez możliwości nieautoryzowanego połączenia z zewnątrz. Fragment konfiguracji brokera Mosquitto pokazano w Listing 3.

```
listener 1883 localhost
allow_anonymous false
password_file /etc/mosquitto/passwd
```

Listing 3: Podstawowa konfiguracja brokera Mosquitto

Utworzenie pliku haseł zrealizowano poleceniem z Listing 4:

```
sudo mosquitto_passwd -c /etc/mosquitto/passwd iot
```

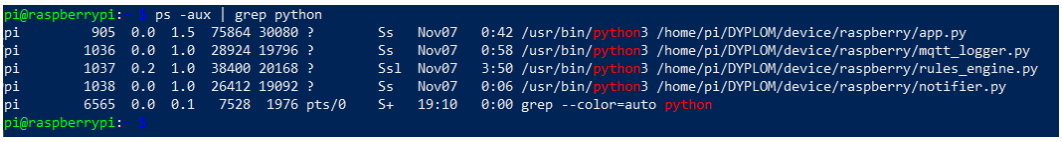
Listing 4: Polecenie tworzące plik haseł dla użytkownika MQTT

Baza danych `events.db` została zainicjalizowana zgodnie z definicją w pliku `schema.sql`, a następnie wypełniona przykładowymi danymi testowymi. Skrypty Pythona — `mqtt_logger.py`, `rules_engine.py`, `notifier.py` oraz `app.py` — umieszczono w katalogu `/home/pi/DYPLOM/device/raspberry/`. Dzięki modularnej architekturze systemu, proces wdrożenia mógł być powtórzony również na innych platformach linuksowych przy zachowaniu identycznej konfiguracji.

## 4.2 Uruchamianie usług systemowych

Wszystkie kluczowe komponenty systemu zostały uruchomione jako niezależne usługi systemowe *systemd*, co zapewniało ich automatyczne wznawianie po restarcie urządzenia i separację procesów. Takie rozwiązanie pozwoliło utrzymać wysoką niezawodność oraz łatwość diagnostyki w przypadku błędów.

Na Rys. 8 przedstawiono widok aktywnych usług systemowych po wdrożeniu aplikacji na Raspberry Pi. Usługi te odpowiadały kolejno za rejestrację danych, analizę zdarzeń, wysyłanie powiadomień oraz obsługę interfejsu webowego.




Rysunek 8: Widok aktywnych usług systemowych po wdrożeniu systemu na Raspberry Pi.

Przykładowy plik jednostki `/etc/systemd/system/mqtt-logger.service` przedstawiono poniżej. Określał on sposób uruchamiania loggera danych po starcie systemu. Przykładową definicję usługi systemowej opisano w Listing 5.

```
[Unit]
Description=MQTT Logger Service
After=network.target mosquitto.service

[Service]
ExecStart=/usr/bin/python3 /home/pi/DYPLOM/device/raspberry/
   mqtt_logger.py
WorkingDirectory=/home/pi/DYPLOM/device/raspberry
Restart=always
User=pi

[Install]
WantedBy=multi-user.target
```

Listing 5: Przykładowa jednostka systemowa systemd dla loggera MQTT

Zdefiniowano też jednostki usług dla komponentów `rules-engine.service`, `notifier.service` oraz `flask-api.service`. Po utworzeniu plików konfiguracyjnych przeładowano menedżer i aktywowano usług. Proces aktywacji usług przedstawiono w Listing 6:

```
sudo systemctl daemon-reload
sudo systemctl enable mqtt-logger.service
sudo systemctl start mqtt-logger.service
```

Listing 6: Aktywacja i uruchamianie usług systemowych

Weryfikację działania usługi pokazano w Listing 7:

```
sudo systemctl status rules-engine.service
```

Listing 7: Sprawdzanie stanu działania komponentów systemowych

Wszystkie komponenty systemowe uruchamiały się automatycznie po restarcie urządzenia, a w przypadku awarii były wznawiane przez *systemd* po kilku sekundach. Taki sposób wdrożenia gwarantował ciągłość działania nawet w przypadku utraty zasilania lub chwilowych błędów.

Tabela 3: Lista usług systemowych wdrożonych na Raspberry Pi.

| **Usługa** | **Skrypt** | **Opis funkcji** |
|---|---|---|
| `mqtt-logger.service` | `mqtt_logger.py` | Rejestrowała komunikaty z czujników oraz zapisywała dane w bazie `SQLite`. |
| `rules-engine.service` | `rules_engine.py` | Analizowała zdarzenia ruchu i sygnały heartbeat, generując alerty i pre-alarmy. |
| `notifier.service` | `notifier.py` | Wysyłała powiadomienia e-mail i sterowała diodami LED w urządzeniach. |
| `flask-api.service` | `app.py` | Udostępniała interfejs REST API i komunikowała się z aplikacją webową React. |

Podział usług na niezależne procesy poprawił stabilność systemu i uprościł diagnostykę. Każdy komponent mógł być rozwijany lub aktualizowany niezależnie, co zapewniało elastyczność i łatwość utrzymania.

## 4.3 Monitoring i konserwacja

Utrzymanie systemu wymagało stałego nadzoru nad procesami, stanem połączenia z czujnikami oraz integralnością danych w bazie. Na potrzeby monitoringu opracowano zestaw skryptów i narzędzi automatyzujących podstawowe czynności administracyjne.

Do analizy logów wykorzystywano standardowe narzędzia systemowe, takie jak `journalctl`, `mosquitto_sub` oraz `tail -f /var/log/syslog`. Pozwalały one obserwować komunikaty usług w czasie rzeczywistym i szybko diagnozować błędy transmisji MQTT. Integralność bazy danych `events.db` kontrolowano poleceniem przedstawionym w Listing 8:

```
sqlite3 /home/pi/DYPLOM/device/raspberry/events.db "PRAGMA
   integrity_check;"
```

Listing 8: Sprawdzenie integralności bazy danych SQLite

Do automatycznego usuwania starych rekordów zastosowano skrypt `cleanup.py`, uruchamiany okresowo (np. raz dziennie) przy pomocy *cron* lub biblioteki *schedule*. Skrypt usuwał dane starsze niż określony limit, co pozwalało utrzymać bazę danych w niewielkim rozmiarze i zapewniało szybki dostęp do aktualnych informacji.

W celu zapewnienia bezpieczeństwa i stabilności działania systemu, administrator wykonywał regularne aktualizacje bibliotek Pythonowych i systemu operacyjnego, okresowe czyszczenie logów oraz tworzenie kopii zapasowych katalogu `/home/pi/DYPLOM/device/raspberry`. Stan wszystkich usług był kontrolowany poleceniem `systemctl status`, które umożliwiało szybką reakcję na ewentualne nieprawidłowości.

Zastosowane rozwiązania pozwoliły osiągnąć wysoką niezawodność i bezobsługowe działanie systemu. Ewentualne błędy były automatycznie rejestrowane w logach i natychmiast widoczne dla administratora. W efekcie system mógł pracować w trybie ciągłym (*24/7*) przy minimalnej interwencji użytkownika, co potwierdziło skuteczność przyjętej architektury wdrożeniowej.

# Rozdział 5

# Testy i analiza działania systemu

Rozdział ten przedstawia proces testowania opracowanego systemu, obejmujący zarówno środowisko symulacyjne, jak i rzeczywiste warunki domowe. Opisano cel i przebieg testów, użyte narzędzia, analizę wyników oraz końcowe wnioski dotyczące poprawności działania i stabilności systemu.

## 5.1 Cel testów

Celem przeprowadzonych testów było potwierdzenie poprawnego działania zaprojektowanego systemu monitorowania aktywności domowej osób starszych, obejmującego warstwę urządzeń IoT (czujniki ruchu ESP8266), warstwę komunikacji MQTT, serwer rejestrujący dane (Raspberry Pi z bazą SQLite) oraz silnik reguł analizujący nieaktywność i brak sygnału z urządzeń.

Podczas testów weryfikowano stabilność komunikacji i poprawność zapisu zdarzeń w bazie danych, sprawdzano reakcję systemu na różne scenariusze ruchu i bezruchu w pomieszczeniach oraz potwierdzano poprawne generowanie stanów *prealert* i ewentualnych alertów. Dodatkowo oceniano zgodność działania z założeniami funkcjonalnymi opisanymi w rozdziale 3.

## 5.2 Środowisko testowe

Testy przeprowadzono w środowisku symulacyjnym uruchomionym na Raspberry Pi 4 Model B. Konfiguracja obejmowała lokalnie działający broker MQTT (*Mosquitto*) na porcie 1883 z uwierzytelnianiem użytkownika `iot/iot`, skrypt `mqtt_logger.py` logujący wiadomości MQTT do bazy *SQLite*, silnik reguł `rules_engine.py` analizujący stany *INACTIVITY* i *NO_HEARTBEAT*, serwer API (*Flask*) udostępniający dane testowe oraz skrypt `simulator_motion.py` generujący symulacje ruchu dla poszczególnych pomieszczeń.

## 5.3 Procedura testowa

Przed rozpoczęciem testów zainicjalizowano środowisko odpowiednimi zmiennymi środowiskowymi, przedstawionymi w Listing 9, oraz parametrami połączenia z brokerem MQTT i bazą SQLite:

```
export API="http://127.0.0.1:5000"
export KEY="iotkey"
export MQTT_HOST="192.168.0.48"
export MQTT_USER="iot"
export MQTT_PASS="iot"
export DB="/home/pi/DYPLOM/device/raspberry/events.db"
```

Listing 9: Zmienne środowiskowe użyte podczas konfiguracji środowiska testowego

W środowisku testowym utworzono trzy wirtualne pomieszczenia: `kitchen_sim`, `bathroom_sim` oraz `living_sim`. Ich ustawienia przedstawiono w Listing 10:

```
/api/rooms/kitchen_sim/settings          inactivity_sec=90,
   prealert_offset_sec=30
/api/rooms/bathroom_sim/settings         inactivity_sec=180,
   prealert_offset_sec=60
/api/rooms/living_sim/settings           inactivity_sec=120,
   prealert_offset_sec=30
```

Listing 10: Ustawienia reguł dla pomieszczeń symulacyjnych użytych w testach

Po ponownym uruchomieniu silnika reguł przeprowadzono trzy równoległe symulacje odpowiadające różnym scenariuszom aktywności. Każdy proces generował dane ruchu (`motion/state`) oraz sygnały kontrolne (`motion/health`), które były przesyłane do brokera MQTT i rejestrowane w bazie `events.db`. Wyniki zapisano także w pliku `mqtt.log`. Po zakończeniu symulacji dane zostały przeanalizowane przy użyciu skryptu `validate_results.py` oraz raportu zbiorczego `analyze_latest.sh`.

## 5.4 Przebieg testów i obserwacje

W trakcie testów obserwowano w czasie rzeczywistym komunikaty MQTT, które potwierdzały poprawne działanie mechanizmu prealertu. Przykładowy fragment logów przedstawiono w Listing 11:

```
iot/eldercare/kitchen_sim/cmd/prealert {"action": "start", "
   reason": "INACTIVITY", "ttl_sec": 20}
iot/eldercare/kitchen_sim/cmd/prealert {"action": "stop",  "
   reason": "INACTIVITY"}
```

Listing 11: Fragment logu MQTT prezentujący aktywację oraz zatrzymanie prealertu

System prawidłowo wykrywał okresy braku ruchu i uruchamiał diodę ostrzegawczą na urządzeniu ESP8266, a po ponownym wykryciu ruchu anulował prealert. W trakcie symulacji zarejestrowano dziesiątki zdarzeń ruchu oraz regularne sygnały „heartbeat", które potwierdzały ciągłą łączność urządzeń z serwerem. Nie odnotowano fałszywych alertów ani utraty komunikacji, a wszystkie dane zostały zapisane poprawnie w bazie SQLite.

## 5.5 Wyniki analizy danych

Automatyczna analiza danych została uruchomiona poleceniem przedstawionym w Listing 12:

```
/home/pi/DYPLOM/device/raspberry/analyze_latest.sh
```

Listing 12: Uruchomienie skryptu analizującego najnowsze dane testowe

Wyniki zapisano w pliku `/home/pi/DYPLOM/tests/20251108-201159/analysis.csv`. Na Tabeli 4 przedstawiono podsumowanie wyników testów symulacyjnych, obejmujące liczbę wykrytych ruchów oraz statusy alertów dla każdego z pomieszczeń.

Tabela 4: Podsumowanie wyników testów symulacyjnych.

| **Pokój** | **Liczba ruchów** | **INACTIVITY** | | **NO_HEARTBEAT** | |
|---|---|---|---|---|---|
| | | **open** | **closed** | **open** | **closed** |
| bathroom_sim | 3 | 0 | 0 | 0 | 0 |
| kitchen | 1 | 0 | 0 | 0 | 0 |
| kitchen_sim | 28 | 0 | 0 | 0 | 0 |
| living_sim | 10 | 0 | 0 | 0 | 0 |
| room1 | 10 | 0 | 0 | 0 | 0 |

Wyniki analizy potwierdziły, że wszystkie pomieszczenia aktywne przesyłały dane i zostały prawidłowo rozpoznane przez silnik reguł. Nie wygenerowano żadnych trwałych alertów o nieaktywności ani braku sygnału, co świadczyło o stabilnym i spójnym działaniu systemu.

## 5.6 Test rzeczywisty w warunkach domowych

Oprócz testów symulacyjnych przeprowadzono również eksperyment w rzeczywistych warunkach domowych. Celem była ocena zachowania systemu podczas długotrwałej pracy z fizycznym czujnikiem ruchu. W tym celu wykorzystano układ ESP8266 NodeMCU z czujnikiem PIR HC-SR501, zainstalowany w pomieszczeniu mieszkalnym o powierzchni około 12 $m^2$. Czujnik zamontowano na wysokości 2,1 m, tak aby jego pole detekcji obejmowało cały obszar aktywności użytkownika. Zasilanie czujnika zrealizowano poprzez port USB Raspberry Pi 4, który pełnił również rolę brokera MQTT, loggera danych i serwera API.

Test trwał około 13 godzin — od godzin porannych do późnego wieczora. W tym czasie urządzenie rejestrowało naturalne zdarzenia ruchu: przemieszczanie się użytkownika, krótkie wyjścia z pokoju oraz dłuższe okresy bezczynności. Ustawienia reguł pozostawały identyczne jak w testach symulacyjnych (`inactivity_sec`=600 s, `prealert_offset_sec`=60 s). Fragment działania prealertu w trakcie realnego testu przedstawiono w Listing 13:

```
{"action": "start", "reason": "INACTIVITY", "ttl_sec": 60}
{"action": "stop",  "reason": "INACTIVITY"}
```

Listing 13: Przykład komunikatów prealertu podczas testów w warunkach domowych

Po około dziewięciu minutach bezruchu system automatycznie uruchamiał prealert — dioda LED na płytce migała, sygnalizując zbliżający się próg nieaktywności. Jeśli użytkownik poruszył się w tym czasie, ostrzeżenie zostawało anulowane. Dłuższe przerwy (np. sen lub opuszczenie pokoju) powodowały wygenerowanie alertu *INACTIVITY*, zapisanego w bazie `events.db` i widocznego w interfejsie webowym.

Podczas całodziennego testu czujnik zarejestrował ponad 150 zdarzeń ruchu, a silnik reguł `rules_engine.py` pracował stabilnie. Wszystkie alerty były zgodne z rzeczywistym zachowaniem użytkownika, a w bazie danych widoczne były regularne zapis wiadomości `motion/state` oraz `motion/health`, co potwierdzało ciągłą łączność z urządzeniem. Nie odnotowano fałszywych wywołań ani nadmiernej czułości czujnika, a średni czas reakcji (system → baza danych) wynosił poniżej 1 s.

## 5.7 Wnioski z testów

Przeprowadzone testy — zarówno symulacyjne, jak i rzeczywiste — potwierdziły, że zaprojektowany system spełniał wszystkie założenia funkcjonalne i techniczne określone na etapie projektu. Połączenie mikrokontrolerów ESP8266, protokołu MQTT, lokalnego serwera Raspberry Pi oraz silnika reguł napisanego w Pythonie pozwoliło stworzyć kompletny i stabilny system IoT. System skutecznie wykrywał stany nieaktywności, reagował na brak sygnału (`NO_HEARTBEAT`) i informował użytkownika o potencjalnych zagrożeniach.

Zarówno testy symulacyjne, jak i całodniowa obserwacja w środowisku domowym, potwierdziły odporność systemu na przerwy w komunikacji, brak fałszywych alarmów oraz poprawne działanie algorytmu prealert. Dzięki modularnej architekturze rozwiązanie można było łatwo rozbudować o kolejne czujniki, integrację z chmurą lub interfejsy mobilne.

Podsumowując, system okazał się stabilny i wiarygodny w działaniu, a uzyskane rezultaty potwierdziły skuteczność przyjętych założeń projektowych. Opracowane rozwiązanie stanowi solidną podstawę dla dalszych badań i wdrożeń w zakresie systemów IoT wspierających opiekę nad osobami starszymi i niepełnosprawnymi.

# Rozdział 6

# Bezpieczeństwo i prywatność

Rozdział ten przedstawia działania podjęte w celu zapewnienia bezpieczeństwa komunikacji, ochrony danych oraz poszanowania prywatności użytkowników w opracowanym systemie IoT. Opisano wdrożone mechanizmy zabezpieczeń, sposób przetwarzania informacji oraz aspekty etyczne i prawne związane z zastosowaniem tego typu rozwiązań.

## 6.1 Bezpieczna komunikacja IoT

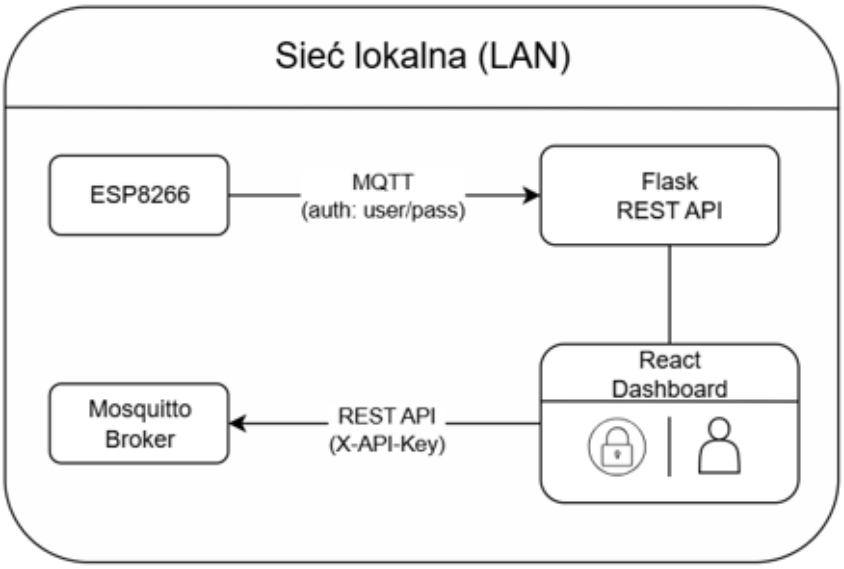
Systemy Internetu Rzeczy były szczególnie narażone na zagrożenia wynikające z otwartej natury komunikacji sieciowej, ograniczonych zasobów sprzętowych urządzeń brzegowych oraz braku zaawansowanych mechanizmów szyfrowania. W opracowanym rozwiązaniu założono, że cała komunikacja przebiegała w obrębie zaufanej sieci lokalnej (LAN), jednak mimo to wdrożono środki bezpieczeństwa, które chroniły dane i zapobiegały nieautoryzowanemu dostępowi.

Urządzenia ESP8266 korzystały z protokołu MQTT, który został zabezpieczony prostą autoryzacją z wykorzystaniem nazwy użytkownika i hasła, wymuszaną przez brokera *Mosquitto*. Konfiguracja obejmowała ograniczenie dostępu do portu 1883 wyłącznie dla adresów IP z sieci lokalnej oraz filtrowanie klientów publikujących wiadomości. W środowisku produkcyjnym zalecano wdrożenie szyfrowania TLS (port 8883) z certyfikatami X.509, co umożliwiałoby bezpieczne przesyłanie komunikatów pomiędzy modułami.

Komunikacja między frontendem React a serwerem *Flask* była realizowana poprzez interfejs REST API, zabezpieczony kluczem autoryzacyjnym przesyłanym w nagłówku `X-API-Key`. Każde żądanie bez poprawnego klucza było automatycznie odrzucane, co uniemożliwiało nieautoryzowany dostęp do danych. Wszystkie wywołania API były rejestrowane w logach serwera wraz z adresem IP i czasem wykonania.

W warstwie aplikacyjnej zastosowano separację odpowiedzialności. Każda usługa działała jako niezależny proces systemowy, co zapobiegało propagowaniu błędów między komponentami. W przypadku awarii jednego modułu, pozostałe kontynuowały pracę. Serwer Flask i broker Mosquitto uruchamiały się jako jednostki *systemd*, co zapewniało ich automatyczne wznawianie po błędzie i stały nadzór przez system operacyjny Raspberry Pi.

Na Rys. 9 przedstawiono schemat komunikacji i autoryzacji pomiędzy poszczególnymi komponentami systemu. Diagram ilustrował przepływ danych oraz zastosowane punkty kontrolne zabezpieczeń.




Rysunek 9: Schemat komunikacji i autoryzacji między komponentami systemu IoT.

Bezpieczeństwo zwiększono także poprzez ograniczenie fizycznego dostępu do urządzenia Raspberry Pi, regularne aktualizacje systemu operacyjnego oraz bibliotek Pythonowych. Wszystkie dane były przechowywane lokalnie, bez łączenia z usługami chmurowymi, co znacząco zmniejszało ryzyko wycieku informacji. Dodatkowo prowadzono stały monitoring logów usług *systemd* i brokera *Mosquitto*, co pozwalało na wczesne wykrywanie prób nieautoryzowanego logowania lub błędów transmisji. Takie rozwiązanie umożliwiało bieżące reagowanie na potencjalne incydenty bezpieczeństwa i utrzymywało wysoką dostępność systemu.

## 6.2 Ochrona danych osobowych

System został zaprojektowany w zgodzie z zasadą minimalizacji danych. Gromadził wyłącznie informacje techniczne i zdarzenia logiczne związane z ruchem, bez danych osobowych użytkowników. Nie rejestrowano obrazu wideo, dźwięku ani innych informacji, które mogłyby prowadzić do identyfikacji osoby.

Dane zapisywane w bazie *SQLite* obejmowały datę i czas wystąpienia zdarzenia, nazwę urządzenia i pomieszczenia, typ alertu (np. *INACTIVITY*, *NO_HEARTBEAT*) oraz parametry techniczne czujników, takie jak czas trwania czy aktualny stan. W żadnym momencie nie przetwarzano danych osobowych w rozumieniu RODO, ponieważ zapisane informacje nie umożliwiały powiązania ich z konkretną osobą fizyczną.

Projekt zachowywał jednak ducha przepisów o ochronie prywatności. Zapewniono pełną transparentność działania, umożliwiając użytkownikowi wgląd w zarejestrowane dane. Zaimplementowano także mechanizmy ograniczające retencję danych — starsze rekordy mogły być automatycznie usuwane za pomocą skryptu `cleanup.py`. System nie przekazywał żadnych informacji do podmiotów trzecich, a baza danych mogła zostać w dowolnym momencie wyczyszczona i zainicjalizowana ponownie przez użytkownika.

W przypadku potencjalnego rozszerzenia systemu o zdalny dostęp przez Internet, należałoby wdrożyć dodatkowe zabezpieczenia: szyfrowanie transmisji HTTPS, dwustopniową autoryzację użytkowników oraz szyfrowanie bazy danych przy użyciu lokalnego klucza. Takie podejście gwarantowałoby zachowanie prywatności również w środowiskach rozproszonych.

## 6.3 Aspekty etyczne i prawne

Opracowany system miał charakter wspomagający, a nie nadzorczy. Jego celem było zwiększenie bezpieczeństwa osób pozostających bez stałej opieki, z jednoczesnym poszanowaniem ich prywatności i autonomii. System nie śledził szczegółowej aktywności użytkownika, a jedynie informował o odstępstwach od normalnych wzorców zachowań, które mogły oznaczać sytuację potencjalnie niebezpieczną.

W ujęciu etycznym istotne było zachowanie równowagi między bezpieczeństwem a prywatnością. Nadmierne monitorowanie mogłoby prowadzić do ingerencji w życie prywatne użytkownika, dlatego w projekcie celowo zrezygnowano z kamer, mikrofonów i innych urządzeń gromadzących dane wrażliwe. Zastosowano jedynie czujniki ruchu rejestrujące proste sygnały binarne, co zapewniało odpowiedni kompromis między funkcjonalnością a ochroną prywatności.

Odpowiedzialność za sposób wykorzystania systemu spoczywała na użytkowniku lub instytucji wdrażającej rozwiązanie. Autor projektu nie przewidywał jego użycia w celach inwigilacyjnych, kontroli pracowników ani do automatycznego podejmowania decyzji dotyczących zdrowia użytkowników.

Z punktu widzenia prawa autorskiego wszystkie elementy systemu oparto na technologiach open-source (*Python*, *Flask*, *Mosquitto*, *React*, *TailwindCSS*), udostępnianych na licencjach MIT, BSD i EPL, co umożliwiało ich swobodne wykorzystanie w projektach akademickich i niekomercyjnych.

Podsumowując, opracowany system potwierdził, że bezpieczne i etyczne rozwiązania IoT nie muszą być złożone ani kosztowne. Kluczowe znaczenie miało odpowiednie zaprojektowanie architektury, ograniczenie zakresu gromadzonych danych oraz świadomość użytkowników w zakresie ich praw i odpowiedzialności.

# Rozdział 7

# Użyteczność i dostępność systemu

Rozdział ten przedstawia ocenę ergonomii interfejsu użytkownika, poziom dostępności rozwiązania dla osób starszych oraz możliwości dalszego rozwoju i integracji systemu. Opisano zastosowane zasady projektowania wizualnego, zachowania responsywności oraz sposób dostosowania aplikacji do potrzeb osób o ograniczonych możliwościach technicznych. Na końcu zaprezentowano kierunki rozbudowy i przyszłe zastosowania.

## 7.1 Ocena ergonomii interfejsu

Interfejs użytkownika aplikacji webowej został opracowany z zachowaniem zasad ergonomii, spójności wizualnej oraz czytelności przekazu. Układ graficzny przyjął formę minimalistyczną, pozbawioną elementów rozpraszających uwagę, a jednocześnie zachowującą pełną funkcjonalność. Widoki panelu — *Dashboard*, *Alerts*, *Devices*, *History* i *Settings* — utrzymano w jednolitym stylu kolorystycznym, z wyraźnym kontrastem pomiędzy tłem a elementami aktywnymi. Pozwoliło to użytkownikowi łatwo identyfikować najważniejsze informacje i szybko reagować na alerty systemowe.

Projekt graficzny powstał z użyciem frameworka *TailwindCSS*, który umożliwił tworzenie nowoczesnych, responsywnych układów dostosowujących się do rozdzielczości ekranu. Interfejs zachowywał czytelność zarówno na ekranach komputerów stacjonarnych, jak i na urządzeniach mobilnych. Wprowadzono dwa motywy graficzne — ciemny i jasny — które użytkownik mógł dowolnie przełączać w zależności od warunków oświetleniowych. Rozwiązanie to zwiększało komfort pracy i ograniczało zmęczenie wzroku podczas dłuższego korzystania z aplikacji.

Komponenty interfejsu, takie jak `StatusPill`, `MiniChart` oraz `Toast`, zostały zaprojektowane z myślą o natychmiastowym przekazie informacji. Kolorystyka komponentów była spójna semantycznie: zielony oznaczał stan normalny, żółty sygnalizował ostrzeżenie, natomiast czerwony wskazywał alert krytyczny. Wszystkie elementy interaktywne posiadały podświetlenie przy najechaniu kursorem (efekt *hover*) oraz wizualne zaznaczenie aktywnego stanu, co zwiększało intuicyjność obsługi. Taki sposób projektowania zapewniał ergonomiczny interfejs zgodny z zasadami *Human-Centered Design* i znacząco ułatwiał codzienne użytkowanie.

## 7.2 Dostępność dla osób starszych

System został opracowany z myślą o osobach starszych oraz nieposiadających doświadczenia technicznego. Jego głównym założeniem było całkowite wyeliminowanie konieczności wykonywania czynności konfiguracyjnych przez użytkownika końcowego. Po uruchomieniu urządzenia wszystkie procesy startowały automatycznie, a komunikacja między czujnikami a serwerem przebiegała w tle, bez potrzeby ingerencji człowieka. Tym samym system działał w pełni samodzielnie, zapewniając ciągłe monitorowanie aktywności.

Dane prezentowano w prostym, kontrastowym panelu webowym, którego interfejs charakteryzował się dużymi elementami graficznymi, czytelnymi etykietami i czcionką bezszeryfową (*sans-serif*). Kolorystyka oraz proporcje komponentów zostały dobrane zgodnie z wytycznymi WCAG (Web Content Accessibility Guidelines), co gwarantowało wysoki kontrast i czytelność. Interfejs nie zawierał nadmiarowych animacji ani efektów wizualnych, które mogłyby utrudniać korzystanie z aplikacji osobom starszym.

Z punktu widzenia użytkownika system nie wymagał żadnej obsługi — osoba monitorowana nie musiała podejmować żadnych działań. Całość funkcjonowania nadzorował opiekun lub członek rodziny, który mógł śledzić status czujników i alerty z poziomu panelu. Do podstawowej obsługi wystarczała elementarna znajomość pracy z przeglądarką internetową i siecią lokalną. Instalacja oprogramowania na Raspberry Pi ograniczała się do kilku poleceń terminala, co umożliwiało szybkie uruchomienie systemu również przez użytkowników niebędących programistami.

## 7.3 Możliwości rozwoju i integracji

System zaprojektowano w sposób modularny, co umożliwiało jego dalszą rozbudowę bez konieczności modyfikacji istniejącej architektury. Każdy z głównych modułów — logger, silnik reguł, interfejs API, aplikacja webowa oraz moduł notifier — działał jako niezależna usługa systemowa *systemd*, dzięki czemu można je było oddzielnie aktualizować i testować. Taka struktura zwiększała elastyczność i niezawodność systemu.

W aktualnej wersji system obsługiwał powiadomienia e-mail kierowane do opiekuna w sytuacjach krytycznych, takich jak długotrwały brak ruchu lub utrata łączności z czujnikiem. Funkcjonalność ta została zaimplementowana w module `notifier.py`, który pobierał otwarte alerty z bazy `SQLite` i przesyłał powiadomienia za pomocą zewnętrznego serwera SMTP. W przyszłości mechanizm ten można było rozszerzyć o wysyłkę wiadomości SMS, integrację z komunikatorami internetowymi lub wprowadzenie systemu wieloetapowych przypomnień, jeśli użytkownik nie zareagował w określonym czasie.

W ramach dalszego rozwoju przewidziano kilka kierunków udoskonaleń. Jednym z nich było rozszerzenie zestawu czujników o tanie moduły środowiskowe, takie jak detektory temperatury i wilgotności (DHT11/DHT22) lub czujniki otwarcia drzwi. Dane z tych urządzeń mogłyby być publikowane w tym samym brokerze MQTT i zapisywane w rozszerzonych tabelach bazy danych.

Kolejnym krokiem miało być wdrożenie zdalnego dostępu do panelu webowego poprzez aktywację HTTPS i wprowadzenie prostego systemu kont użytkowników, co umożliwiłoby bezpieczne logowanie spoza sieci lokalnej, np. przez tunel VPN lub dedykowaną domenę `.local`. Rozważano również integrację z usługą synchronizacji danych, umożliwiającą okresowy eksport statystyk do plików CSV i wysyłanie ich na adres e-mail opiekuna bądź zapisywanie kopii zapasowych w sieci NAS.

Ostatnim z rozpatrywanych kierunków rozwoju był tryb awaryjny offline, który pozwalałby systemowi działać nawet w przypadku utraty połączenia z brokerem MQTT lub bazą danych. Mechanizm buforowania lokalnego umożliwiałby zapis zdarzeń w pamięci urządzenia i ich automatyczne przesłanie po przywróceniu łączności, co znacznie zwiększałoby niezawodność całego rozwiązania.

Podsumowując, opracowany system okazał się nie tylko funkcjonalny i czytelny w obsłudze, ale również elastyczny w kontekście przyszłych rozszerzeń. Zastosowana architektura modułowa oraz nacisk na dostępność sprawiły, że rozwiązanie to może stanowić solidną podstawę dla rozwoju inteligentnych systemów wsparcia osób starszych i niepełnosprawnych w warunkach domowych.

# Rozdział 8

# Podsumowanie

Rozdział ten podsumowuje wyniki prac nad opracowaniem systemu Internetu Rzeczy (ang. *Internet of Things*, IoT) wspomagającego zdalny nadzór nad aktywnością osób starszych lub niepełnosprawnych w środowisku domowym. Przedstawione poniżej wnioski obejmują najważniejsze osiągnięcia, rozwiązane problemy oraz możliwe kierunki dalszego rozwoju projektu.

Celem pracy było stworzenie kompletnego i w pełni działającego systemu IoT, który umożliwia monitorowanie aktywności użytkownika z wykorzystaniem prostych czujników ruchu oraz lokalnego przetwarzania danych. Opracowane rozwiązanie obejmuje wszystkie warstwy funkcjonalne systemu — od sprzętu (moduł ESP8266 z czujnikiem PIR), poprzez komunikację i analizę danych, aż po prezentację wyników w formie interaktywnego panelu webowego.

W trakcie realizacji osiągnięto wszystkie zakładane cele. Urządzenie poprawnie rejestrowało zdarzenia ruchu, a dane były przesyłane i zapisywane w lokalnej bazie *SQLite*. Silnik reguł skutecznie analizował historię aktywności i generował alerty dotyczące nieaktywności, zbyt długiego przebywania w jednym pomieszczeniu oraz braku sygnału od urządzenia (INACTIVITY, DWELL_CRITICAL, NO_HEARTBEAT). Wykorzystanie interfejsu *REST API* oraz panelu w technologii *React* umożliwiło intuicyjny dostęp do danych i zapewniło rozszerzalność całego systemu. Całość działała w trybie ciągłym na komputerze Raspberry Pi, bez konieczności używania usług chmurowych, co zwiększało niezależność i bezpieczeństwo danych.

Podczas implementacji napotkano kilka trudności praktycznych, związanych głównie z konfiguracją połączeń MQTT, stabilnością komunikacji Wi-Fi oraz synchronizacją czasu pomiędzy urządzeniami. Problemy te rozwiązano poprzez zastosowanie mechanizmów ponawiania transmisji, buforowania danych oraz okresowego monitorowania stanu łączności. Przeprowadzone testy — zarówno symulacyjne, jak i rzeczywiste — potwierdziły stabilność działania systemu, poprawność detekcji zdarzeń oraz niezawodność komunikacji pomiędzy wszystkimi komponentami.

Największą zaletą opracowanego rozwiązania okazała się jego prostota, niski koszt wdrożenia oraz pełna niezależność od zewnętrznych usług sieciowych. System może zostać zainstalowany w dowolnym domu z dostępem do sieci Wi-Fi, bez konieczności posiadania specjalistycznej wiedzy technicznej. Dzięki modularnej architekturze poszczególne elementy można rozwijać niezależnie, co umożliwia stopniowe rozszerzanie funkcjonalności o kolejne czujniki, pomieszczenia i reguły przetwarzania danych. Potwierdziło to elastyczność i skalowalność projektu.

W dalszej perspektywie system może być rozwijany w kilku kierunkach. Pierwszym z nich jest integracja z dodatkowymi czujnikami środowiskowymi, takimi jak sensory temperatury, wilgotności czy otwarcia drzwi, co pozwoliłoby rozszerzyć zakres analizowanych informacji o warunki otoczenia. Kolejnym kierunkiem może być wdrożenie zdalnego dostępu do systemu z wykorzystaniem bezpiecznego połączenia HTTPS, autoryzacji użytkowników oraz dwustopniowego uwierzytelniania. Naturalnym etapem rozwoju pozostaje również opracowanie mobilnej wersji panelu użytkownika, umożliwiającej nadzór z poziomu smartfona lub tabletu, co dodatkowo zwiększy dostępność systemu dla opiekunów i członków rodzin.

Podsumowując, opracowany system stanowi w pełni funkcjonalny prototyp, który może zostać realnie wykorzystany jako narzędzie wspierające opiekę domową nad osobami starszymi lub niepełnosprawnymi. Projekt potwierdził, że przy użyciu ogólnodostępnych komponentów oraz podstawowych umiejętności technicznych można stworzyć niezawodne, bezpieczne i użyteczne rozwiązanie, łączące nowoczesną technologię z realną wartością społeczną i potencjałem do dalszego rozwoju.

# Bibliografia


[1] C. Pfister, *Internet Rzeczy. Budowa sieci i aplikacji*. Gliwice: Helion, 2017. Available at: https://helion.pl/ksiazki/internet-rzeczy-budowa-sieci-i-aplikacji-christian-pfister [Accessed: 2025-10-15].

[2] E. Foundation, "Mosquitto mqtt broker documentation." https://mosquitto.org/, 2025. [Accessed: 2025-10-15].

[3] E. Systems, "Esp8266 official documentation." https://docs.espressif.com/, 2025. [Accessed: 2025-10-15].

[4] E. Foundation, "Eclipse paho mqtt client library for python." https://www.eclipse.org/paho/, 2025. [Accessed: 2025-10-15].

[5] P. Projects, "Flask web framework documentation." https://flask.palletsprojects.com/, 2025. [Accessed: 2025-10-15].

[6] R. P. Foundation, "Raspberry pi documentation." https://www.raspberrypi.com/documentation/, 2025. [Accessed: 2025-10-15].

[7] M. P. Inc., "React – a javascript library for building user interfaces." https://react.dev/, 2025. [Accessed: 2025-10-15].

[8] S. Consortium, "Sqlite database documentation." https://sqlite.org/docs.html, 2025. [Accessed: 2025-10-15].

[9] T. Labs, "Tailwind css framework documentation." https://tailwindcss.com/, 2025. [Accessed: 2025-10-15].

# Spis rysunków

# Spis tabel

# Spis listingów