UMCS

UNIWERSYTET MARII CURIE—SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **informatyka**

Kamil Goławski
nr albumu: 312304

Projektowanie oraz implementacja wielowarstwowego Rest API w Spring Boot — od CRUDa do modularnego systemu zarządzania danymi pogodowymi

Designing and implementing
a multi—layer Rest API in Spring Boot
— from CRUD to a modular
weather data management system

Praca licencjacka
napisana w Katedrze oprogramowania systemów informatycznych
Instytutu Informatyki UMCS
pod kierunkiem **dr hab. Beaty Byliny**

Lublin 2022

Spis treści

Wstęp

W dobie postępu technologicznego, inżynieria oprogramowania zmierza ku tworzeniu co raz bardzej elastycznych i modularnych aplikacji. Niniejsza praca przedstawia proces projektowania i implementacji wielowarstwowego systemu REST API w technologii Spring Boot na przykładzie warstwy backendowej aplikacji Climatly, stanowiącej część złożonego ekosystemu do zarządzania danymi pogodowymi z sieci czujników meteorologicznych.

Głównym celem pracy jest zaprezentowanie procesu powstawania warstwy serwerowej systemu od prostego interfejsu CRUD do modularnego oraz skalowalnego rozwiązania, które realizuje złożone scenariusze biznesowe związane z szerokopojetym zarządzaniem odczytami środowiskowymi. Szczególną uwagę poświęcono modularyzacji i separacji odpowiedzialności poszczególnych komponentów backendu, co stanowi kluczowy aspekt projektowania skalowalnych aplikacji serwerowych.

W pracy przedstawione zostaną decyzje architektoniczne oraz uzasadnienie wyboru konkretnych rozwiązań technologicznych warstwy backendowej, jak Spring Boot, bazy danych PostgreSQL i InfluxDB. Zaprezentowane zostaną również implementacje modułów funkcjonalnych backendu, takich jak zarządzanie czujnikami, integracja z bazą szeregów czasowych, komunikacja w czasie rzeczywistym, automatyzacja zadań oraz zarządzanie konfiguracją systemu.

W kolejnych rozdziałach przedstawione zostaną szczegółowe aspekty projektu warstwy serwerowej, od koncepcji systemu, poprzez wykorzystane technologie, aż po analizę modularności i przepływu danych. Na zakończenie, zostaną podsumowane doświadczenia projektowe w kontekście rozwoju backendu, napotkane wyzwania oraz potencjalne kierunki rozwoju warstwy serwerowej systemu Climatly, który stanowi jedną z kluczowych usług całego ekosystemu zarządzania danymi meteorologicznymi.

Rozdział 1

Wprowadzenie do systemu zarządzania siecią pogodową czujników

System Climatly to kompleksowe rozwiązanie umożliwiające monitoring parametrów środowiskowych poprzez sieć rozproszonych czujników meteorologicznych. Celem systemu jest gromadzenie, przetwarzanie i udostępnianie danych pogodowych w formie przyjaznej dla użytkownika końcowego. Architektura systemu oparta jest na modelu mikroserwisów, gdzie warstwa backendowa — będąca przedmiotem niniejszej pracy — stanowi kluczowy komponent integrujący, przetwarzający i udostępniający dane.

1.1 Koncepcja systemu Climatly oraz jego komponenty zewnętrzne

System Climatly to kompleksowe rozwiązanie umożliwiające monitoring parametrów środowiskowych poprzez sieć rozproszonych czujników meteorologicznych. Celem systemu jest gromadzenie, przetwarzanie i udostępnianie danych pogodowych w formie przyjaznej dla użytkownika końcowego.

Ekosystem Climatly składa się z następujących komponentów:

- **Sieć czujników fizycznych** — rozproszone urządzenia pomiarowe dokonujące cyklicznych odczytów parametrów środowiskowych (temperatura, wilgotność, ciśnienie atmosferyczne). Czujniki te stanowią warstwę sprzętową systemu.
- **Gateway (brama)** — urządzenie pośredniczące zbierające dane z fizycznych czujników i przekształcające je do ustandaryzowanego formatu. Gateway pełni funkcję mostu między warstwą sprzętową a warstwą przetwarzania danych.

- **Broker MQTT** — komponent odpowiedzialny za zarządzanie kolejką wiadomości, implementujący protokół MQTT (Message Queuing Telemetry Transport). Za jego pośrednictwem dane z Gateway'a trafiają do bazy danych szeregów czasowych.

- **InfluxDB** — wyspecjalizowana baza danych przystosowana do efektywnego przechowywania i przetwarzania danych szeregów czasowych. InfluxDB przechowuje surowe odczyty z czujników wraz z dokładnymi znacznikami czasowymi.

- **Warstwa serwerowa** — centralna część systemu odpowiedzialna za pobieranie danych z InfluxDB, ich przetwarzanie i udostępnianie innym komponentom. Szczegółowy opis przepływu danych oraz architektura tej warstwy zostanie przedstawiona w rozdziale 3.

- **Panel administracyjny** — aplikacja umożliwiająca zarządzanie konfiguracją systemu. Komunikuje się z warstwą serwerową, wysyłając polecenia zmiany statusu czujników, modyfikacji częstotliwości odświeżania oraz innych parametrów operacyjnych.

- **Aplikacja kliencka** — interfejs użytkownika końcowego, prezentujący dane w formie interaktywnych wykresów, map i zestawień. Konsumuje dane udostępniane przez warstwę serwerową.

1.2 Rola oraz założenia projektowe warstwy backendowej

Warstwa backendowa, będąca głównym przedmiotem niniejszej pracy, stanowi kluczowy element integrujący różne komponenty systemu Climatly. Projektując tę warstwę, nadałem jej rolę scentralizowanego węzła, który nie tylko pobiera i przetwarza dane pomiarowe z bazy odczytów pogodowych, ale również wzbogaca je o niezbędny kontekst przestrzenny i semantyczny. Backend realizuje główną logikę biznesową systemu, zapewniając mechanizmy odpowiedzialne za niezawodność oraz skalowalność całego rozwiązania.

Dzięki dwukierunkowej komunikacji z pozostałymi komponentami, warstwa serwerowa nie tylko udostępnia przetworzone dane aplikacji klienckiej, ale również przyjmuje i realizuje polecenia konfiguracyjne z panelu administracyjnego. Umożliwia to dynamiczne dostosowanie zachowania systemu poprzez wpływanie na parametry czujników, ich statyczne dane oraz cykl życia samego backendu, jak i dedykowanej, niezależnej od wcześniej wspomnianej bazy odczytów pogodowych, bazy metadanych.

Założenia projektowe

Projektując warstwę backendową Climatly, przyjąłem szereg kluczowych założeń projektowych:

1. **Wysoka wydajność** — System powinien minimalizować opóźnienia między momentem pozyskania danych a ich udostępnieniem użytkownikowi końcowemu, zapewniając płynne i responsywne doświadczenie użytkownika. Jest to szczególnie istotne w dzisiejszych czasach bardzo wymagających użytkowników.

2. **Niezależność procesów** — Aktualizacja danych z pojedynczego czujnika nie powinna wpływać na dostępność i aktualizację danych z pozostałych czujników. Należy zadbać, aby procesy nie blokowały się, nie walczyły o zasoby i aby pojedynczy błąd nie hamował działania całego systemu.

3. **Monitorowanie niesprawności** — System powinien śledzić wszelkie nieprawidłowości w działaniu czujników oraz innych komponentów, umożliwiając szybką diagnozę i reakcję na problemy. Pomoże to znacznie w rozwoju systemu, zbierając wszelakie problemy, które pojawiają się w trakcie jego funkcjonowania na każdej warstwie całego ekosystemu Climatly.

4. **Elastyczność konfiguracji** — Backend powinien umożliwiać łatwe dodawanie nowych czujników, usuwanie lub czasowe wyłączanie istniejących, bez wpływu na funkcjonowanie pozostałych elementów systemu. Projektując system należy antycypować przyszłą ewolucję infrastruktury sprzętowej, aby warstwa serwerowa stanowiła elastyczną platformę zdolną do absorpcji zmian technologicznych, a nie barierę ograniczającą potencjał rozwojowy całego ekosystemu.

5. **Skalowalność** — Architektura systemu powinna obsługiwać wzrost liczby czujników, objętości danych oraz liczby użytkowników bez znaczącej degradacji wydajności. System powinien umożliwiać rozszerzenie o nowe lokalizacje i budynki bez ingerencji w istniejącą infrastrukturę, zapewniając efektywne działanie przy zwiększonym obciążeniu.

6. **Modularność** — System powinien być podzielony na niezależne moduły funkcjonalne o jasno zdefiniowanych granicach odpowiedzialności. Takie podejście ułatwia równoległą pracę nad różnymi funkcjonalnościami, izolację błędów oraz stopniowe rozszerzanie systemu bez destabilizacji istniejących komponentów. Pozostawia to szeroką perspektywę rozwoju o nowe usługi i serwisy nie tylko w warstwie serwerowej, ale i poza nią.

1.3 Wpływ zakładanych funkcjonalności na architekturę systemu

Projektując warstwę serwerową, rozpocząłem od szczegółowej analizy wymaganych funkcjonalności, które system miał oferować. Był to naturalny krok następujący bezpośrednio po określeniu założeń projektowych — w moim procesie projektowym założenia i funkcjonalności stanowiły dwa ściśle powiązane elementy, które wzajemnie się uzupełniały i kształtowały. To właśnie zidentyfikowane funkcjonalności, w połączeniu z przyjętymi wcześniej założeniami, stały się fundamentem dla późniejszych decyzji architektonicznych i naturalnym źródłem modularnego podziału całej warstwy serwerowej.

Na podstawie analizy wymagań określiłem następujące kluczowe funkcjonalności:

1. **Zarządzanie danymi czujników** — Przechowywanie i zarządzanie metadanymi czujników, takimi jak lokalizacja, przypisanie do budynku, piętro, nazwa oraz status operacyjny. Funkcjonalność ta była niezbędna, ponieważ baza danych szeregów czasowych przechowuje jedynie surowe odczyty, nie oferując możliwości zarządzania danymi statycznymi. Aby system mógł poprawnie funkcjonować i nadawać głębszy sens zbieranym danym, warstwa serwerowa musiała przejąć odpowiedzialność za zarządzanie tymi metadanymi.

2. **Pobieranie i przetwarzanie odczytów** — Cykliczne pobieranie danych pomiarowych z bazy szeregów czasowych, ich walidacja oraz agregacja w różnych przekrojach czasowych. Jest to absolutnie podstawowa funkcjonalność systemu, która w cyklu życia całego ekosystemu powinna być zawsze dostępna, niezależnie od sytuacji. Stanowi ona najbardziej newralgiczną część, ponieważ na jej podstawie budowane są kluczowe dane w docelowej bazie danych warstwy serwerowej.

3. **Udostępnianie danych historycznych** — Dostarczanie interfejsu do przeszukiwania i filtrowania historycznych danych pomiarowych, z możliwością generowania statystyk i wykresów. Jest to jeden z głównych przypadków użycia dla docelowych użytkowników systemu, dla których przeglądanie danych w ramach określonych przedziałów czasowych, np. jednego dnia czy całego miesiąca, powinno być intuicyjne i efektywne.

4. **Dostarczanie aktualnych odczytów w czasie rzeczywistym** — Zapewnienie mechanizmu strumieniowania najnowszych odczytów do aplikacji klienckiej bez konieczności ciągłego odpytywania serwera. Funkcjonalność ta pozwala użytkownikom, oprócz dostępu do danych historycznych, na zapoznanie się z bieżącymi odczytami oraz obserwowanie dynamiki ich zmian, szczególnie istotne

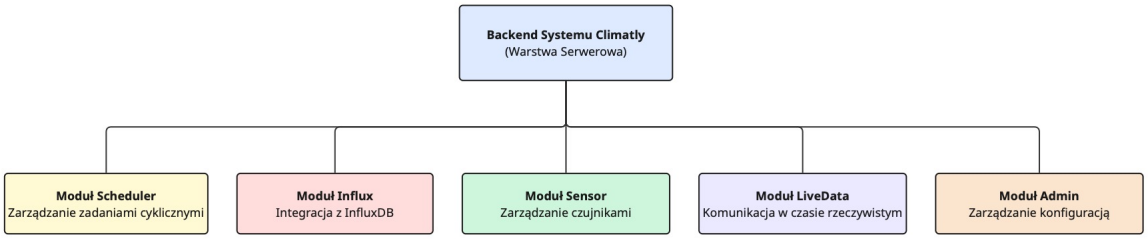
podczas skrajnych warunków pogodowych czy nagłych zmian parametrów środowiskowych.

5. **Zarządzanie konfiguracją systemu** — Umożliwienie zdalnej konfiguracji parametrów pracy, takich jak częstotliwość odświeżania danych czy obecny status czujników. Funkcjonalność ta stanowi naturalne przedłużenie panelu administracyjnego, pozwalając osobom moderującym system na wprowadzanie zmian w czasie rzeczywistym, bez konieczności fizycznej ingerencji w infrastrukturę.

6. **Automatyzacja zadań cyklicznych** — Planowanie i wykonywanie zadań cyklicznych związanych z pobieraniem, przetwarzaniem i archiwizacją danych. Ta funkcjonalność zapewnia niezawodność systemu poprzez eliminację zależności od czynnika ludzkiego w rutynowych operacjach. Dzięki automatyzacji, system samodzielnie zarządza harmonogramem zadań, gwarantując ciągłość działania oraz spójność danych bez konieczności ręcznej interwencji czy poświęcania dodatkowego czasu przez administratorów.

To właśnie zidentyfikowane wcześniej funkcjonalności bezpośrednio zdeterminowały strukturę systemu, stając się głównym czynnikiem przemawiającym za przyjęciem architektury modularnej. W myśl zasady pojedynczej odpowiedzialności (Single Responsibility Principle) każdy moduł odpowiada za dostarczanie konkretnej, jasno określonej funkcjonalności. Specyficzne wymagania dotyczące zarządzania danymi czujników, komunikacji w czasie rzeczywistym czy automatyzacji zadań naturalnie wskazywały na potrzebę separacji tych obszarów. Zamiast tworzenia monolitycznej struktury, rozbicie systemu na wyspecjalizowane komponenty pozwoliło na osiągnięcie większej przejrzystości kodu oraz lepszych możliwości rozbudowy. Dzięki temu podejściu mogłem zapewnić większą elastyczność, łatwiejsze testowanie oraz możliwość rozwoju poszczególnych funkcjonalności niezależnie od siebie.

Poniżej znajduje się diagram ilustrujący podział warstwy serwerowej na moduły funkcjonalne, których szczegółowe omówienie zostanie przedstawione w rozdziale 3.



Rysunek 1.1: Schemat modułów warstwy serwerowej systemu Climatly

Rozdział 2

Technologie i narzędzia wykorzystane w projekcie

Wybór odpowiednich technologii stanowił fundamentalny element procesu projektowania każdego systemu informatycznego. Podejmując decyzje technologiczne, kierowałem się przede wszystkim trzema nadrzędnymi zasadami: efektywną współpracą komponentów, elastycznością architektury oraz stabilnością zastosowanych rozwiązań. Dążyłem do stworzenia systemu, w którym poszczególne warstwy i moduły funkcjonują jako spójna całość, jednocześnie zachowując luźne powiązania między sobą. Ta cecha okazała się szczególnie cenna w kontekście ewolucji wymagań biznesowych — umożliwiła mi elastyczne dostosowywanie pojedynczych elementów systemu bez naruszania jego ogólnej struktury. Zaprojektowana architektura modułowa pozwala na rozwijanie i modyfikację poszczególnych funkcjonalności niezależnie od siebie, co znacząco usprawnia proces rozbudowy systemu. Świadomie wybrałem technologie o ugruntowanej pozycji na rynku i szerokiej dokumentacji, co zminimalizowało ryzyko wystąpienia problemów typowo technicznych. Dzięki temu mogłem skoncentrować wysiłki na dostarczaniu funkcjonalności biznesowych, zamiast poświęcać czas na rozwiązywanie niespodziewanych problemów infrastrukturalnych. Niniejszy rozdział przedstawia szczegółowe uzasadnienie wyboru poszczególnych komponentów technologicznych warstwy serwerowej systemu Climatly.

2.1 Java jako język programowania dla aplikacji backendowych

W nowoczesnej architekturze aplikacji, warstwa backendowa odpowiada za przetwarzanie danych, implementację logiki biznesowej oraz komunikację z bazami danych. Współcześnie, dominującym standardem implementacji interfejsów backendowych stał

się REST API (Representational State Transfer), który wykorzystuje protokół HTTP do komunikacji między klientem a serwerem, zapewniając skalowalność i elastyczność systemów rozproszonych [1].

Jako język programowania warstwy backendowej wybrałem Javę, która dzięki swoim unikalnym właściwościom, stała się jednym z wiodących języków w obszarze tworzenia warstw serwerowych, szczególnie w sektorze aplikacji korporacyjnych. Według raportu State of Java in the Enterprise, aż 70% organizacji buduje nowe aplikacje pełnostackowe w Javie [2]. Ta silna pozycja wynika z kilku kluczowych cech tego języka, które okazały się wyjątkowo przydatne w moim projekcie:

Obiektowość

Java to w pełni obiektowy język programowania, którego fundamenty opierają się na czterech głównych filarach koncepcji obiektowych: abstrakcji, hermetyzacji, dziedziczeniu i polimorfizmie. Obiektowość w Javie nie jest jedynie dodatkiem do języka, ale jego integralną częścią, co pozwala na tworzenie bardziej modularnego, skalowalnego i łatwiejszego w utrzymaniu kodu GeeksforGeeks, 2023 GeeksforGeeks. Abstrakcja pozwala na ukrycie wewnętrznej złożoności systemu, eksponując jedynie niezbędne detale. Hermetyzacja (enkapsulacja) umożliwia ochronę danych poprzez ich ukrycie wewnątrz klasy i udostępnianie jedynie za pomocą kontrolowanych metod dostępowych. Dziedziczenie wspomaga wielokrotne wykorzystanie kodu poprzez tworzenie hierarchii klas, gdzie klasy potomne dziedziczą cechy i zachowania klas nadrzędnych. Polimorfizm zaś pozwala na wykorzystanie jednego interfejsu do różnych implementacji, co znacząco zwiększa elastyczność systemu Raygun, 2023

Wieloplatformowość

Jedną z najbardziej charakterystycznych cech Javy jest jej niezależność od platformy sprzętowej i systemowej. Zasada „write once, run anywhere" umożliwia uruchamianie tego samego kodu na różnych systemach operacyjnych bez konieczności rekompilacji [3]. Jest to możliwe dzięki Wirtualnej Maszynie Javy (JVM), która zapewnia warstwę abstrakcji między kodem aplikacji a systemem operacyjnym. W środowiskach korporacyjnych, gdzie często występuje heterogeniczna infrastruktura, ta cecha pozwala na znaczące obniżenie kosztów wdrożenia i utrzymania aplikacji.

Wydajność i skalowalność

Java oferuje zaawansowane mechanizmy do obsługi wielowątkowości i przetwarzania współbieżnego, co jest kluczowe dla wysokowydajnych aplikacji backendowych.

Wprowadzone w Javie 21 wirtualne wątki umożliwiają efektywną obsługę tysięcy równoczesnych połączeń, co jest niezbędne w systemach REST API o dużym obciążeniu. Dodatkowo, zaawansowane mechanizmy optymalizacji w JVM zapewniają wysoką wydajność aplikacji przy zachowaniu wszystkich zalet języka wysokiego poziomu.

Istotnym elementem wydajności Javy jest automatyczne zarządzanie pamięcią, realizowane przez moduł zwany Garbage Collector (GC). W przeciwieństwie do innych języków programowania, jak C czy C++, programista Javy nie musi ręcznie alokować i zwalniać pamięci, co eliminuje wiele potencjalnych błędów związanych z wyciekami pamięci. Garbage Collector automatycznie identyfikuje obiekty, które nie są już używane i zwalnia zajmowaną przez nie pamięć, co zwiększa stabilność aplikacji działających przez długi czas.

Skalowalność aplikacji Java wynika również z możliwości efektywnego wykorzystania zasobów sprzętowych. JVM automatycznie dostosowuje się do dostępnej pamięci i liczby procesorów, co pozwala na optymalne działanie zarówno na małych urządzeniach, jak i na potężnych serwerach.

Dojrzałość

Java istnieje na rynku od ponad 25 lat, co czyni ją jednym z najbardziej ugruntowanych języków programowania w sektorze przedsiębiorstw. Ta długa obecność przełożyła się na stabilny i dojrzały ekosystem, bogaty w narzędzia, biblioteki oraz sprawdzone praktyki programistyczne.

Jedną z kluczowych zalet Javy w kontekście jej dojrzałości jest jej kompatybilność wsteczna. Kod napisany w starszych wersjach języka działa bez modyfikacji w nowszych wersjach, co minimalizuje koszty migracji i pozwala na ochronę dotychczasowych inwestycji w oprogramowanie. Ta cecha ma szczególne znaczenie w projektach długoterminowych, gdzie stabilność platformy jest priorytetem.

Mimo wspomnianiej kompatybilności wstecznej, Oracle regularnie wydaje nowe wersje Javy, wprowadzając innowacyjne rozwiązania przy jednoczesnym zachowaniu stabilności. W obecnym modelu wydawniczym, niektóre wersje otrzymują status LTS (Long—Term Support), co gwarantuje im długoterminowe wsparcie techniczne i aktualizacje bezpieczeństwa. Wsparcie dla wersji LTS może trwać nawet ponad dekadę — przykładowo Java 8 będzie wspierana do 2030 roku. Ta przewidywalność cyklu życia jest nieoceniona w środowiskach korporacyjnych, gdzie nagłe zmiany technologiczne mogą generować znaczące ryzyko operacyjne

Bogaty ekosystem narzędzi

Jednym z największych atutów Javy jest jej rozbudowany ekosystem bibliotek i frameworków wspierających tworzenie aplikacji backendowych. Dla twórców REST API szczególne znaczenie mają frameworki takie jak Spring Boot, który dzięki podejściu „convention over configuration" radykalnie upraszcza proces tworzenia aplikacji, automatyzując wiele aspektów konfiguracji i dostarczając gotowe rozwiązania dla typowych problemów.

Java oferuje również uniwersalne interfejsy dostępu do baz danych (JDBC) oraz zaawansowane frameworki ORM (Object—Relational Mapping), które abstrakcyjnie traktują różne systemy bazodanowe, upraszczając operacje na danych i zwiększając przenośność aplikacji.

Te kluczowe cechy czynią Javę językiem doskonale dopasowanym do wymagań nowoczesnych aplikacji backendowych, szczególnie w standardzie REST API.

2.2 Spring Boot jako główny framework projektu

Testowo cos innego

2.3 Systemy baz danych i ich integracja w koncepcji ORM

W projekcie Climatly zastosowałem dwa komplementarne systemy baz danych: PostgreSQL jako tradycyjną bazę relacyjną oraz InfluxDB jako specjalistyczną bazę szeregów czasowych. Wybór dwóch różnych systemów bazodanowych nie był przypadkowy – wynikał z analizy specyficznych wymagań projektu oraz charakteru przetwarzanych danych.

PostgreSQL — relacyjna baza metadanych

PostgreSQL to zaawansowany, otwartoźródłowy system zarządzania relacyjnymi bazami danych (RDBMS), który w projekcie Climatly pełni rolę głównego repozytorium metadanych czujników. Wybór PostgreSQL był podyktowany kilkoma kluczowymi czynnikami:

- **Integralność danych** — PostgreSQL oferuje pełne wsparcie dla ACID (Atomicity, Consistency, Isolation, Durability), co zapewnia spójność danych nawet w

przypadku awarii systemu. Jest to krytyczne dla danych kontekstowych, takich jak lokalizacje czujników czy ich statusy operacyjne.

- **Złożone relacje** — Model relacyjny doskonale sprawdza się przy modelowaniu złożonych zależności między obiektami biznesowymi. W projekcie Climatly pozwoliło to na efektywne reprezentowanie powiązań między czujnikami, lokalizacjami, budynkami i piętrami.
- **Zaawansowane zapytania** — PostgreSQL oferuje rozbudowany język SQL, który umożliwia tworzenie złożonych zapytań analitycznych oraz filtrowanie i agregację danych według wielu kryteriów.

InfluxDB — nierelacyjna baza szeregów czasowych

InfluxDB to wyspecjalizowana, nierelacyjna baza danych zaprojektowana do efektywnego przechowywania i analizy danych szeregów czasowych. W projekcie Climatly pełni ona rolę repozytorium przechowującego surowe odczyty z czujników. Wykorzystanie InfluxDB było zdeterminowane następującymi czynnikami:

- **Optymalizacja pod kątem szeregów czasowych** — InfluxDB jest zoptymalizowana pod kątem przechowywania i przetwarzania dużych wolumenów danych uporządkowanych chronologicznie, co idealnie odpowiada charakterowi odczytów z czujników meteorologicznych.
- **Wysoka wydajność zapisu** — System oferuje znacząco wyższą przepustowość zapisu w porównaniu do tradycyjnych baz relacyjnych, co jest kluczowe przy ciągłym strumieniowaniu danych z sieci czujników.
- **Efektywne zapytania czasowe** — InfluxDB udostępnia język zapytań Flux, który umożliwia wydajne wykonywanie operacji na zakresach czasowych, co jest fundamentalnym wymaganiem dla analizy trendów pogodowych.

Podział odpowiedzialności między bazami danych

W architekturze systemu Climatly przyjąłem jasny podział odpowiedzialności między obydwiema bazami danych:

PostgreSQL przechowuje:

- Metadane czujników (lokalizacja, nazwa, piętro, budynek)
- Statusy operacyjne urządzeń (online, offline, błąd)
- Konfigurację systemu (częstotliwość odświeżania danych)
- Rejestry awarii i zdarzeń systemowych

- Najnowsze agregacje danych do szybkiego dostępu

InfluxDB przechowuje:

- Surowe odczyty z czujników (temperatura, wilgotność, ciśnienie)
- Pełne szeregi czasowe wszystkich pomiarów
- Historyczne dane z precyzyjnymi znacznikami czasowymi

2.4 Integracja baz danych w modelu ORM

Komunikacja z bazą PostgreSQL w warstwie backendowej odbywa się z wykorzystaniem Spring Data JPA, która implementuje koncepcję ORM (Object-Relational Mapping). ORM to paradygmat programistyczny, który umożliwia mapowanie obiektów języka Java bezpośrednio na struktury relacyjne bazy danych. W praktyce oznacza to, że klasy Javy z odpowiednimi adnotacjami stają się reprezentacją tabel w bazie danych, a ich instancje odzwierciedlają poszczególne rekordy.

Dzięki zastosowaniu ORM w projekcie Climatly:

- Nie ma potrzeby pisania ręcznych zapytań SQL - framework generuje je automatycznie na podstawie nazw metod w repozytoriach
- Encje Javy (np. `Sensor`, `Location`) są bezpośrednio przekształcane w tabele bazodanowe
- Relacje między obiektami (jak one-to-many czy many-to-many) są wyrażane za pomocą adnotacji takich jak `@OneToMany` czy `@ManyToMany`
- Operacje CRUD stają się proste i intuicyjne - wystarczy wywołać metody na obiektach repozytoriów

Kluczowe elementy tej integracji to:

- **Repozytoria Spring Data** - interfejsy, które automatycznie implementują podstawowe operacje CRUD na encjach, znacząco redukując ilość kodu boilerplate.
- **Adnotacje JPA** - deklaratywne definicje mapowań obiektowo-relacyjnych, które upraszczają transformację między światem obiektowym Javy a relacyjnym modelem bazy danych.

2.5 Dodatkowe narzędzia i biblioteki

W projekcie Climatly, oprócz podstawowych technologii takich jak Java, Spring Boot i systemy bazodanowe, zastosowałem szereg specjalistycznych narzędzi, które usprawniły codzienną pracę nad kodem i przyspieszyły rozwój aplikacji. Wykorzystanie tych narzędzi znacząco zoptymalizowało wiele procesów twórczych, takich jak migracje

bazy danych, zarządzanie powtarzalnym kodem, konfigurowanie lokalnych środowisk programistycznych czy monitorowanie działania systemu poprzez logowanie.

Docker — konteneryzacja aplikacji

Rola Dockera w nowoczesnym procesie wytwarzania oprogramowania jest niezastąpiona. Docker to platforma umożliwiająca tworzenie, wdrażanie i uruchamianie aplikacji w kontenerach, które zawierają wszystkie niezbędne zależności i biblioteki potrzebne do działania aplikacji. Kontenery te są lekkie, przenośne i izolowane od systemu operacyjnego, co umożliwia spójne działanie aplikacji niezależnie od środowiska.

W pracy developerskiej wykorzystałem Docker Compose, który doskonale integruje się ze Spring Bootem i umożliwia szybkie przygotowanie lokalnego środowiska deweloperskiego. Docker Compose pozwolił mi na tworzenie i zarządzanie wieloma kontenerami jednocześnie, definiując całą infrastrukturę jako kod. Dzięki temu mogłem zdefiniować zarówno bazę PostgreSQL, jak i InfluxDB w jednym pliku konfiguracyjnym:

Kluczową zaletą tego rozwiązania jest możliwość przechowywania danych bez ingerencji w środowisko produkcyjne, co pozwoliło mi na swobodne eksperymentowanie, testowanie zmian i rozwijanie funkcjonalności bez ryzyka uszkodzenia danych produkcyjnych.

Najważniejszym elementem konteneryzacji dla środowiska produkcyjnego jest Dockerfile, który definiuje wszystkie procesy niezbędne do uruchomienia aplikacji na docelowej infrastrukturze, niezależnie czy będzie to Kubernetes (K8s), czy AWS Elastic Container Service (ECS) w razie potrzeby przeniesienie backendu na zupełnie nową infrastrukturę. Ten Dockerfile automatyzuje proces budowania i wdrażania aplikacji, co znacząco upraszcza konfigurację środowiska produkcyjnego i minimalizuje ryzyko błędów związanych z różnicami między środowiskami.

```
version: '3.8'
services:
  postgres-db-local:
    image: 'postgres:latest'
    container_name: postgres-db
    environment:
      - POSTGRES_DB=mydatabase
      - POSTGRES_USER=kamil.golawski
      - POSTGRES_PASSWORD=secret
    ports:
      - '5432:5432'
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    restart: unless-stopped

  influxdb:
    image: influxdb:2.7
    container_name: influxdb
    ports:
      - "8086:8086"
    volumes:
      - ./influxdb_data:/var/lib/influxdb2
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=admin
      - DOCKER_INFLUXDB_INIT_PASSWORD=adminadmin
      - DOCKER_INFLUXDB_INIT_ORG=SKNI
      - DOCKER_INFLUXDB_INIT_BUCKET=czujniki
    restart: unless-stopped
```

Listing 2.1: Fragment pliku `docker-compose.yaml` definiującego lokalne środowisko deweloperskie

Liquibase - zarządzanie zmianami schematu bazy danych

Liquibase to narzędzie open-source przeznaczone do zarządzania i wersjonowania zmian w schemacie bazy danych. W projekcie Climatly odgrywa ono kluczową rolę w utrzymaniu spójności struktury bazy danych PostgreSQL na różnych środowiskach.

Omawiane narzędzie używa koncepcji "changesetów" - atomowych, ponumerowanych zestawów zmian, które są sekwencyjnie aplikowane do bazy danych. Każdy changeset jest identyfikowany przez unikalne ID oraz autora, co zapewnia pełną kontrolę nad historią zmian i możliwość śledzenia, kto i kiedy wprowadził konkretne modyfikacje. Z początku projektu Liquibase został przeze mnie wykorzystany do zdefiniowania początkowego schematu tabel, który bardziej szczegółowo będzie omówiony w rozdziale 3.

```
1  <changeSet id="1736355579438-3" author="kamilgolawski">
2      <createTable tableName="sensor">
3          <column autoIncrement="true" name="id" type="SMALLINT">
4              <constraints nullable="false" primaryKey="true" primaryKeyName="pk_sens
5          </column>
6          <column name="status" type="VARCHAR(255)"/>
7          <column name="last_update" type="DATETIME"/>
8          <column name="temperature" type="SMALLINT"/>
9          <column name="humidity" type="INT"/>
10         <column name="pressure" type="INT"/>
11         <column name="location_id" type="SMALLINT"/>
12     </createTable>
13 </changeSet>
```

Listing 2.2: Fragment pliku inicjalizującego schemat bazy danych

W miarę ewolucji aplikacji, pojawiały się potrzeby modyfikacji struktury bazy danych. Zamiast ręcznego wykonywania skryptów SQL na każdym środowisku, Liquibase umożliwił mi deklaratywne definiowanie zmian, które następnie były automatycznie aplikowane podczas startu aplikacji, jednocześnie dbając o spójność wcześniejszych changesetów oraz obecnych już realnych danych w tabeli, na których stratę nie mogłem sobie pozwolić.

```
<changeSet id="1739223645331-1" author="kamilgolawski">
    <addColumn tableName="location">
        <column name="faculty_abbreviation" type="VARCHAR(10)"/>
    </addColumn>
</changeSet>

<changeSet id="drop_gas_resistance_columns" author="kamilgolawski">
    <dropColumn tableName="sensor_data">
        <column name="gas_resistance"/>
    </dropColumn>
</changeSet>
```

Listing 2.3: Przykład modyfikacji struktury bazy danych

Lombok - generowanie schematycznego kodu

Lombok to popularna biblioteka Javy, która znacząco usprawnia proces programowania poprzez redukcję powtarzalnego kodu. Java jako język jest często krytykowana za wymaganie dużej ilości tzw. boilerplate code - standardowego, powtarzalnego kodu, który programista musi pisać w wielu miejscach. Dotyczy to szczególnie klas przechowujących dane, gdzie dla każdego pola trzeba często tworzyć gettery, settery, konstruktory, metody equals, hashCode i toString.

Wykorzystanie Lomboka szczególnie usprawnia pracę ze Springiem, który dla optymalnego działania wymaga specyficznych konstruktorów i metod dostępowych. To, co normalnie wymagałoby napisania dziesiątek linii kodu, z Lombokiem sprowadza się do dodania jednej adnotacji co dramatycznie zwiększa czytelność i zmniejsza ilość kodu, który musi być utrzymywany.

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Getter
@Setter
@Entity
@Table(name = "sensor_data")
public class SensorData {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "data_id")
  private Long dataId;

  @ManyToOne(fetch = FetchType.EAGER)
  @JoinColumn(name = "sensor_id", nullable = false)
  private Sensor sensor;

  private ZonedDateTime timestamp;

  @Column(precision = 5, scale = 1)
  private BigDecimal temperature;

  private Integer humidity;

  private Integer pressure;
}
```

Listing 2.4: Fragment klasy encji Sensor z adnotacjami Lombok

Profile Spring Boot - zarządzanie konfiguracją

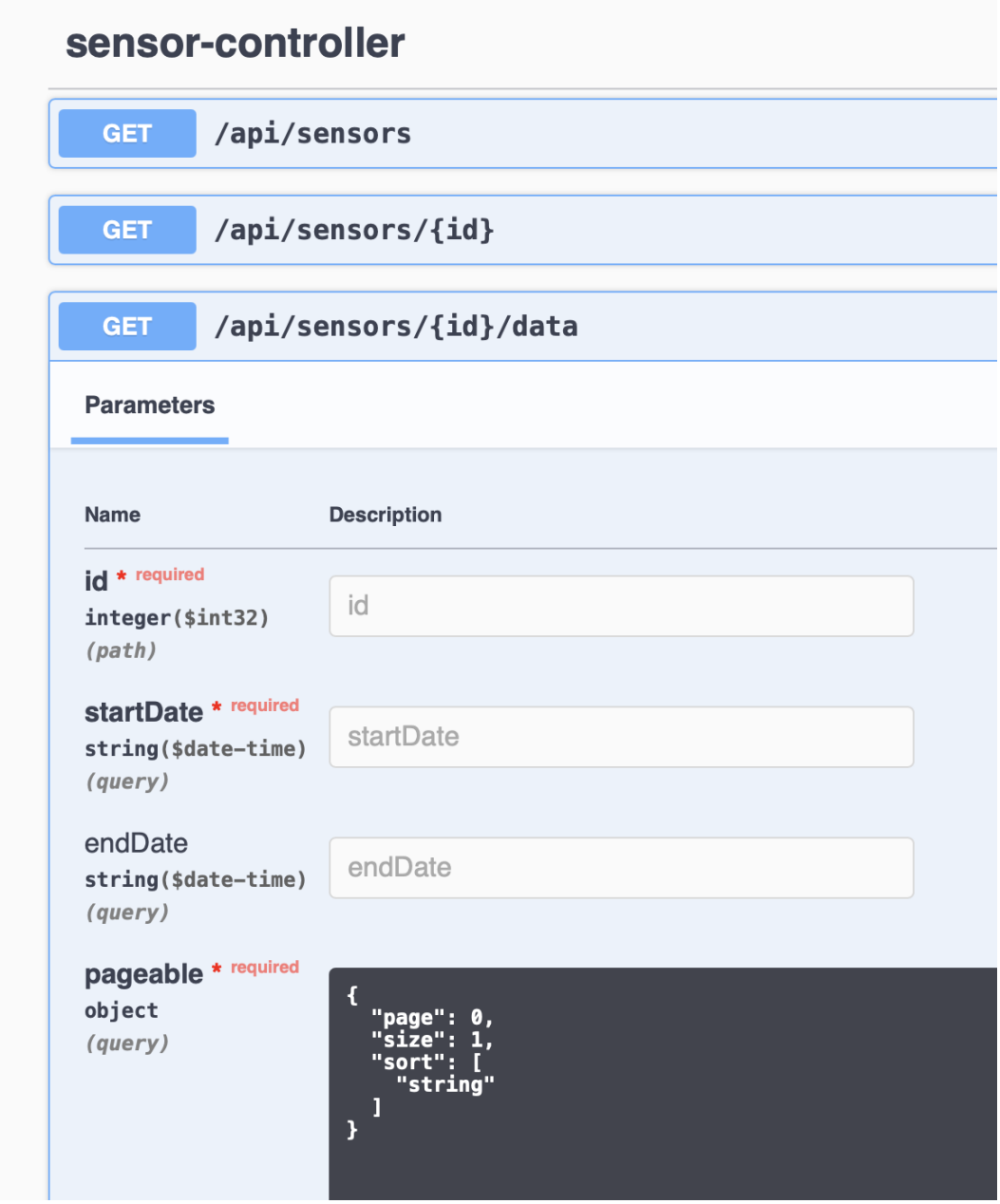
Profile Spring Boot to mechanizm umożliwiający elastyczne dostosowanie konfiguracji aplikacji w zależności od środowiska uruchomieniowego. W projekcie Climatly wykorzystałem profile do oddzielenia konfiguracji środowiska developerskiego od produkcyjnego poprzez zdefiniowanie różnych parametrów połączenia do bazy danych oraz konfigurację poziomów logowania. W środowisku developerskim zastosowałem bardziej szczegółowy poziom logowania, co znacząco ułatwia debugowanie i śledzenie przepływu danych, podczas gdy w środowisku produkcyjnym ograniczyłem logi do najważniejszych komunikatów, co optymalizuje wydajność aplikacji. Oczekiwany profil aktywuje się jako parametr przy uruchamianiu aplikacji z pliku wykonywalnego JAR.

```
# application-local.yaml
spring:
  datasource:
    url: ${DB_URL}
  mvc:
    async:
      request-timeout: 360000
logging:
  level:
    root: INFO
    skni.kamilG: DEBUG
```

Listing 2.5: Fragment konfiguracji dla profilu lokalnego

Swagger — automatyczna dokumentacja API

Swagger to narzędzie do automatycznego generowania dokumentacji interfejsu API w formacie OpenAPI. W projekcie Climatly zastosowałem integrację ze Swagger, co pozwala nie tylko na łatwą integrację aplikacji klienckiej z backendem, lecz również otwiera furtkę dla zewnętrznych klientów, jasno definiując jakie są dostępne endpointy, jakie są ich wymagania wejściowe oraz oczekiwane formaty odpowiedzi. Dzięki tej dokumentacji, deweloperzy frontendowi otrzymują precyzyjny opis wszystkich dostępnych funkcji API bez konieczności ręcznego tworzenia dokumentacji, która mogłaby się szybko zdezaktualizować.



Rysunek 2.1: Przykładowy widok dokumentacji API wygenerowanej przez Swagger UI

Rozdział 3

Modularność oraz szczegółowa architektura systemu

Projektując warstwę serwerową systemu Climatly, przyjąłem architekturę modularną, która wynika bezpośrednio z wcześniej zidentyfikowanych funkcjonalności oraz założeń projektowych. Modularność nie była jedynie wyborem architektonicznym, ale koniecznością wynikającą z różnorodności wymagań technicznych oraz potrzeby zachowania elastyczności systemu, aby potencjalne awarie jednego z modułów z góry nie skreślały działania pozostałych części.

Architektura modularna przynosi kilka istotnych korzyści w kontekście systemu zarządzania danymi pogodowymi. Po pierwsze, umożliwia separację odpowiedzialności — każdy moduł koncentruje się na konkretnym aspekcie działania systemu, co ułatwia zrozumienie kodu i jego utrzymanie. Po drugie, wspiera niezależny rozwój — różne funkcjonalności mogą być rozwijane równolegle bez ryzyka konfliktów. Po trzecie, zwiększa testowalność — każdy moduł może być testowany w izolacji, co upraszcza proces weryfikacji poprawności działania. Wreszcie, ułatwia diagnozowanie problemów — błędy są lokalizowane w obrębie konkretnych modułów, co przyspiesza proces debugowania.

- **Separacja odpowiedzialności** — Każdy moduł koncentruje się na konkretnym aspekcie działania systemu, co ułatwia zrozumienie kodu i jego utrzymanie. Programista pracujący nad modułem Scheduler nie musi znać szczegółów implementacji komunikacji z InfluxDB.
- **Niezależny rozwój** — Różne funkcjonalności mogą być rozwijane równolegle bez ryzyka konfliktów. Podczas gdy jeden developer optymalizuje zapytania Flux, inny może skupić się na ulepszaniu mechanizmów SSE.
- **Zwiększona testowalność** — Każdy moduł może być testowany w izolacji, co upraszcza proces weryfikacji poprawności działania. Testy modułu Admin nie

wymagają uruchomienia bazy InfluxDB.

- **Łatwiejsze diagnozowanie problemów** — Błędy są lokalizowane w obrębie konkretnych modułów, co przyspiesza proces debugowania. Problemy z połączeniami SSE nie wpływają na pobieranie danych historycznych.
- **Elastyczność w rozbudowie** — Nowe funkcjonalności można dodawać jako osobne moduły bez modyfikacji istniejącego kodu, zachowując stabilność działającego systemu.

3.1 Podział odpowiedzialności w systemie

3.1.1 Moduł Sensor — zarządzanie czujnikami

Moduł Sensor stanowi rdzeń systemu Climatly i odpowiada za kompleksowe zarządzanie danymi czujników meteorologicznych. Jest to najbardziej rozbudowany moduł, który integruje dane z dwóch źródeł: metadane z bazy PostgreSQL oraz surowe odczyty z bazy InfluxDB lecz nie odpowiada za ich bezpośredni odczyt.

Główne zadania modułu:

- **Zarządzanie metadanymi czujników** — Moduł przechowuje i zarządza informacjami kontekstowymi o czujnikach, takimi jak lokalizacja geograficzna, przypisanie do konkretnego budynku i piętra, status operacyjny oraz konfigurację częstotliwości odświeżania danych. Te metadane są kluczowe dla nadania sensu surowym odczytom pomiarowym.
- **Udostępnianie danych historycznych** — Realizuje interfejs REST API umożliwiający pobieranie historycznych danych pomiarowych w różnych przekrojach czasowych i przestrzennych. Obsługuje filtrowanie według dat, lokalizacji oraz konkretnych czujników, zapewniając paginację dla dużych zestawów danych.
- **Agregacja i kontekstualizacja danych** — Łączy oraz agreguje surowe odczyty z InfluxDB z metadanymi z PostgreSQL, tworząc kompletne odpowiedzi zawierające zarówno wartości pomiarowe, jak i informacje o lokalizacji oraz statusie czujników.

Serce modułu stanowi encja `Sensor`, która reprezentuje fizyczny czujnik meteorologiczny w systemie. Encja ta łączy w sobie zarówno aktualne odczyty pomiarowe, jak i metadane opisujące czujnik:

```java
@Entity
@Table(name = "sensor")
public class Sensor implements Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Short id;

    @Enumerated(EnumType.STRING)
    private SensorStatus status;

    private ZonedDateTime lastUpdate;
    private BigDecimal temperature;
    private Integer humidity;
    private Integer pressure;
    private Short refreshRate;
    private Short floor;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "location_id")
    private Location location;


    public void updateFromSensorData(SensorData latestSensorData, Clock clock) {
        this.temperature = latestSensorData.getTemperature();
        this.humidity = latestSensorData.getHumidity();
        this.pressure = latestSensorData.getPressure();
        this.lastUpdate = ZonedDateTime.now(clock);
    }
}
```

Listing 3.1: Encja Sensor — centralna klasa modułu

3.1.2 Moduł Influx — integracja z bazą szeregów czasowych

Moduł Influx stanowi wyspecjalizowany komponent systemu dedykowany wyłącznie odczytom danych z czujników meteorologicznych. Jest jedynym modułem posiadającym bezpośrednie połączenie z bazą szeregów czasowych InfluxDB, co zapewnia kontrolowaną i scentralizowaną komunikację z tym krytycznym źródłem danych. Wydzielenie funkcjonalności do osobnego modułu skutecznie rozwiązało wyzwania związane ze złożonym wykorzystywaniem języka Flux do konstruowania zapytań czasowych.

Główne odpowiedzialności modułu:

- **Wyłączny dostęp do InfluxDB** — Stanowi jedyną bramę komunikacyjną z

bazą szeregów czasowych, centralizując wszystkie operacje pobierania danych pomiarowych i eliminując duplikację połączeń w systemie.

- **Walidacja i mapowanie danych** — Sprawdza poprawność i kompletność pobranych odczytów, obsługuje sytuacje z brakującymi lub opóźnionymi danymi. Po poprawnej walidacji przekształca surowe odpowiedzi InfluxDB na typowane obiekty Java wykorzystywane w pozostałych częściach systemu.
- **Dynamiczne generowanie zapytań Flux** — Automatycznie konstruuje zoptymalizowane zapytania w języku Flux na podstawie zewnętrznych parametrów, takich jak identyfikator czujnika czy częstotliwość odświeżania danych. Dzięki temu inne moduły nie muszą wchodzić w bezpośrednią interakcję ze złożoną składnią Flux.

Kluczowym elementem modułu jest metoda `buildQueryWithParams()`, która dynamicznie konstruuje zapytania w języku Flux dostosowane do specyficznych wymagań każdego żądania:

```
1  private String buildQueryWithParams(Short sensorId, Short refreshRate) {
2      return String.format(
3          """
4          from(bucket: "%s")
5            |> range(start: -%ds)
6            |> filter(fn: (r) => r["_measurement"] == "sensor_readings")
7            |> filter(fn: (r) => r["_field"] == "%s" or
8                                 r["_field"] == "%s" or
9                                 r["_field"] == "%s" or
10                                r["_field"] == "%s")
11           |> pivot(rowKey:["_time"], columnKey: ["_field"], valueColumn: "_value")
12           |> filter(fn: (r) => r["sensor_id"] == %s)
13           |> last(column: "_time")
14         """,
15         bucket,
16         refreshRate,
17         HUMIDITY_FIELD,
18         PRESSURE_FIELD,
19         TEMPERATURE_FIELD,
20         SENSOR_ID_FIELD,
21         sensorId);
22 }
```

Listing 3.2: Metoda generująca zapytania Flux w module Influx

3.1.3 Moduł LiveData — komunikacja w czasie rzeczywistym

Moduł LiveData implementuje zaawansowaną funkcjonalność strumieniowania danych w czasie rzeczywistym przy użyciu technologii Server-Sent Events (SSE). Stanowi kluczowy komponent umożliwiający aplikacji klienckiej otrzymywanie aktualnych odczytów z czujników w czasie rzeczywistym bez obciążania serwera.

Główne odpowiedzialności modułu:

- **Zarządzanie połączeniami SSE** — Utrzymuje aktywne połączenia z klientami, zarządza ich cyklem życia oraz automatycznie usuwa nieaktywne połączenia w przypadku przerwania transmisji lub timeout-u.
- **Subskrypcje tematyczne** — Umożliwia klientom subskrybowanie konkretnych czujników lub wszystkich aktualizacji w systemie, efektywnie filtrując i kierując odpowiednie powiadomienia do zainteresowanych odbiorców.
- **Rozgłaszanie aktualizacji** — Natychmiast wysyła powiadomienia o nowych odczytach do wszystkich zainteresowanych klientów, zapewniając synchronizację danych w czasie rzeczywistym.
- **Optymalizacja przepustowości** — Implementuje mechanizmy heartbeat, reconnection oraz buforowania zdarzeń, minimalizując obciążenie sieci przy jednoczesnym zachowaniu niezawodności połączeń.

Kluczowym elementem modułu jest kontroler `LiveController`, który demonstruje zaawansowaną obsługę połączeń Server-Sent Events z precyzyjną konfiguracją nagłówków HTTP:

```
1  @RestController
2  @RequestMapping("/live-api/sensors")
3  public class LiveController {
4
5      private final ISseEmitterService sseEmitterService;
6
7      @GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
8      public ResponseEntity<SseEmitter> streamAllSensorsUpdates(
9          @RequestHeader(value = "Last-Event-ID", required = false) String lastEventI
10         HttpServletResponse response) {
11
12         String clientId = UUID.randomUUID().toString();
13         SseEmitter emitter = sseEmitterService.createSseEmitter(clientId, null);
14         return ResponseEntity.ok().contentType(MediaType.TEXT_EVENT_STREAM).body(em
15     }
```

Listing 3.3: Fragment kontrolera SSE obsługujący strumieniowanie wszystkich czujników

3.1.4 Moduł Scheduler — automatyzacja zadań

Moduł Scheduler stanowi prosty choć zaawansowany komponent systemu, który powstał z konieczności implementacji indywidualnego procesowania każdego czujnikia. Wcześniejsze podejście generowało istotne problemy operacyjne, które szczegółowo omówię w rozdziale 4. Moduł odpowiada za efektywne zarządzanie pulą wirtualnych wątków, przydzielając każdemu z nich niezależne zadania cyklicznej aktualizacji danych pomiarowych.

Główną odpowiedzialością modułu jest koordynacja harmonogramu pobierania najnowszych odczytów z bazy InfluxDB oraz synchronizacja metadanych w bazie PostgreSQL dla każdego czujnika zgodnie z jego indywidualnie skonfigurowanymi parametrami częstotliwości. Rozwiązanie oparte na mapie zadań umożliwia dynamiczne dodawanie, usuwanie oraz modyfikowanie harmonogramów poszczególnych czujników bez destabilizacji pozostałych elementów systemu.

Dodatkowo, moduł implementuje mechanizm automatycznego wykrywania nieaktywnych czujników oraz oznaczania ich statusu jako błędnego, co zapewnia kompleksowe monitorowanie stanu całej infrastruktury pomiarowej.

Centralnym elementem modułu jest klasa `Scheduler`, która funkcjonuje jako koordynator wszystkich zadań cyklicznych w systemie:

```
1  @Service
2  public class Scheduler {
3
4      private final Map<Short, ScheduledFuture<?>> scheduledTasks = new ConcurrentHash
5      private final SensorUpdateService sensorUpdateService;
6      private final TaskScheduler taskScheduler;
7
8      @Value("${scheduler.default-rate:180}")
9      private short defaultRate;
10
11     @PostConstruct
12     public void init() {
13         startDefaultTasks();
14     }
15
16     public void updateTaskRates(List<Sensor> sensorsToUpdate) {
17         sensorsToUpdate.forEach(sensor -> {
18             ScheduledFuture<?> existingTask = scheduledTasks.get(sensor.getId());
19             if (existingTask != null) {
20                 existingTask.cancel(false);
21                 log.debug("Cancelled existing task for sensor {}", sensor.getId());
22             }
```

```
23
24              short rate = Optional.ofNullable(sensor.getRefreshRate()).orElse(default
25              ScheduledFuture<?> newTask = taskScheduler.scheduleAtFixedRate(
26                  () -> updateSingleSensor(sensor),
27                  Duration.ofSeconds(rate)
28              );
29              scheduledTasks.put(sensor.getId(), newTask);
30              log.debug("Scheduled task for sensor {} with rate {} seconds", sensor.g
31          });
32      }
33
34      private void updateSingleSensor(Sensor sensor) {
35          try {
36              sensorUpdateService.updateSingleSensor(sensor);
37          } catch (Exception e) {
38              log.error("Failed to update sensor {}: {}", sensor.getId(), e.getMessage
39          }
40      }
41
42      @PreDestroy
43      public void shutdown() {
44          scheduledTasks.values().forEach(task -> task.cancel(false));
45          scheduledTasks.clear();
46          log.info("Cleaned up all scheduled tasks");
47      }
48  }
```

3.1.5 Moduł Admin — zarządzanie konfiguracją

Moduł Admin stanowi wyspecjalizowany komponent odpowiedzialny za zdalne zarządzanie konfiguracją systemu oraz parametrami operacyjnymi czujników. Jest to moduł o ograniczonym dostępie, zabezpieczony mechanizmem autoryzacji opartym na kluczu API, który umożliwia administratorom wprowadzanie zmian w czasie rzeczywistym bez konieczności fizycznej ingerencji w infrastrukturę. Główną charakterystyką modułu jest implementacja operacji wsadowych, które pozwalają na jednoczesną modyfikację wielu czujników, co znacząco usprawnia administrację rozległą siecią urządzeń pomiarowych.

Moduł oferuje dwie kluczowe funkcjonalności: zarządzanie statusem czujników poprzez aktywację, dezaktywację lub oznaczanie jako niesprawnych oraz konfigurację częstotliwości odświeżania danych dla poszczególnych urządzeń. Operacje wsadowe umożliwiają administratorom efektywne zarządzanie całymi grupami czujników jednocześnie, co jest szczególnie istotne w przypadku konieczności szybkich zmian kon-

figuracyjnych na dużą skalę. Dodatkowo, moduł automatycznie propaguje zmiany do modułu Scheduler, zapewniając natychmiastowe odzwierciedlenie nowych ustawień w harmonogramie zadań systemowych.

Centralnym elementem modułu jest serwis `AdminService`, który implementuje generyczną metodę obsługującą różne typy poleceń administracyjnych:

Listing 3.1: Serwis Admin z metodą aktualizacji wsadowej

```
@Service
public class AdminService implements IAdminService {

    private final SensorRepository sensorRepository;
    private final Scheduler scheduler;

    @Transactional
    public <T extends SensorCommand> void updateSensors(List<T> commands) {
        List<Sensor> sensorsToUpdate = commands.stream()
            .map(command -> sensorRepository.findById(command.sensorId())
                .map(sensor -> {
                    if (command instanceof RefreshRateCommand refreshRateCommand) {
                        sensor.setRefreshRate(refreshRateCommand.refreshRate());
                        log.info("Updating sensor {} refresh rate to {} seconds",
                            sensor.getId(), refreshRateCommand.refreshRate());
                    } else if (command instanceof StatusCommand statusCommand) {
                        sensor.setStatus(statusCommand.status());
                        log.info("Updating sensor {} status to {}",
                            sensor.getId(), statusCommand.status());
                    }
                    return sensor;
                })
                .orElseThrow(() -> new SensorNotFoundException(command.sensorId())))
            .collect(Collectors.toList());

        List<Sensor> savedSensors = sensorRepository.saveAll(sensorsToUpdate);
        scheduler.updateTaskRates(savedSensors);
        log.info("Updated {} sensors", savedSensors.size());
    }
}
```

3.2 Przepływ danych w systemie

Testowo cos innego

Rozdział 4

Podsumowanie doświadczeń projektowych

Testowo cos innego

4.1 Napotkane wyzwania i zastosowane rozwiązania

Testowo cos innego

4.2 Potencjalne kierunki rozwoju systemu Climatly

Testowo cos innego zeby sprawdzic czy dziala

4.3 Osiągnięte rezultaty i wnioski końcowe

Testowo cos innego zeby skompilowac

Podsumowanie

Tu treść podsumowania ktora zostala zmieniona.

Spis listingów

Spis rysunków

Bibliografia

[1] Vaadin, *State of Java in the Enterprise Report*, 2023, https://vaadin.com/blog/2023-trends-for-java-in-the-enterprise, [dostęp: 05.05.2025].

[2] Oracle, *Java Documentation - Get Started*, 2023, https://docs.oracle.com/en/java/, [dostęp: 05.05.2025].

[3] GUVI, *What is Platform Independence – Java & .net – "write once, run anywhere"*, 2023, https://www.guvi.in/blogs/what-is-platform-independence-java-net-write-once-run-anywhere/, [dostęp: 05.05.2025].

[4] Roy T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, 2000, https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, [dostęp: 05.05.2025].

[5] Carson Gross, *REST - Explained For Beginners*, 2023, https://htmx.org/essays/rest-explained/, [dostęp: 05.05.2025].

[6] IBM, *What Is a REST API (RESTful API)?*, 2024, https://www.ibm.com/think/topics/rest-apis, [dostęp: 05.05.2025].