Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Telekomunikacji

Praca dyplomowa inżynierska

na kierunku Telekomunikacja
w specjalności Teleinformatyka i Zarządzanie w Telekomunikacji

System do zarządzania warunkami technicznymi w pomieszczeniach biurowych oparty na architekturze mikrousługowej

Stanisław Skrzypek
Numer albumu 300501

promotor
dr hab. inż. Artur Tomaszewski

WARSZAWA 2022

System do zarządzania warunkami technicznymi w pomieszczeniach biurowych oparty na architekturze mikrousługowej

Streszczenie. Brak odpowiedniego zarządzania energią w budynkach biurowych od wielu lat powodował straty zarówno finansowe, jak i środowiskowe. Te pierwsze są nie tylko wynikiem nieoptymalnej gospodarki energetycznej powodującej zwiększone koszty eksploatacji budynków ale również wynikiem obniżonej efektywności pracowników przebywających w niekomfortowych i niesprzyjających warunkach oświetleniowych i termicznych.

Celem tej pracy było zaproponowanie rozwiązania umożliwiającego pomiar wartości kluczowych parametrów pomieszczeń biurowych oraz sugerującego podjęcie stosownych działań mających na celu zarówno poprawienie komfortu pracowników, jak i redukcję zużywanej energii.

W odpowiedzi na przedstawione powyżej problemy przygotowano dedykowany system informatyczny. Przyjmuje on aktualne pomiary wykonane przez specjalnie zaprojektowany zestaw czujników, przetwarza je oraz generuje na ich podstawie wynik przedstawiający użytkownikowi proponowane działania. Rozwiązanie zostało przygotowane w oparciu o architekturę mikrousługową, która umożliwia podział projektu na wiele małych, połączonych ze sobą modułów usługowych. Konsekwencją takiego podejścia jest zwiększona odporność na awarie, skalowalność oraz łatwość wdrożenia względem pozostałych architektur.

Duży nacisk położono na automatyzację procesu rozwoju i wdrażania aplikacji. Projekt został utworzony w oparciu o metodykę ciągłej integracji oraz ciągłego wdrażania w celu skrócenia czasu potrzebnego do wydania nowych wersji systemu. W pracy opisano sposób wykorzystania różnorodnych narzędzi służących do zwiększenia stopnia automatyzacji rozwoju systemu.

W ramach pracy przeprowadzono przegląd istniejących prac naukowych dotyczących optymalnej wartości temperatury oraz natężenia światła w pomieszczeniach biurowych. Na ich podstawie przygotowano zestaw domyślnych wartości odzwierciedlających oczekiwane warunki panujące w pomieszczeniach. Użytkownicy korzystający z systemu mają także możliwość definiowania własnych reguł dotyczących preferowanych wartości mierzonych parametrów. Celem systemu jest powiadamianie użytkowników, czy określone przez nich warunki zgadzają się z rzeczywistymi pomiarami.

Słowa kluczowe: Zarządzanie energią, system informatyczny, architektura mikrousługowa

An IT system for managing internal conditions in office spaces based on microservices architecture

Abstract. Lack of proper energy management in office buildings has caused both financial, as well as environmental loss in many a year. The former are not only the result of suboptimal energy economy resulting in increased operating costs of buildings, but also as a result of reduced work efficiency of people staying in uncomfortable and unfavorable lighting and thermal conditions.

The aim of this paper was to propose a solution that would both enable the measurement of values of key parameters of office spaces and suggest significant actions aimed at both improving the comfort of employees and reduction of energy consumption.

In response to the problems presented above, a dedicated IT system was prepared. It takes the actual measurements made through a specially prepared set of sensors, processes them and generates the result presenting the proposed actions to the user. The solution was prepared based on the microservice architecture, which allows for a division of the project into many small, connected microservices. The consequence of this approach is increased fault tolerance, scalability and ease of implementation in relation to other architectures.

Much emphasis was placed on the automation of the system development and implementation process. The project was created based on the methodology of continuous integration and continuous delivery in order to reduce the time needed to release new system versions. The paper describes the use of various tools to increase the degree of automation of system development.

As part of the work, a review of existing scientific works concerning optimal temperature and light intensity in offices has been conducted. On their basis, a set of default values that reflect the preferred indoor conditions has been prepared. Users also have the option of defining their own rules regarding the expected values of measured parameters. The purpose of the system is to notify users whether the conditions they set are in line with the actual measurements.

Keywords: Energy management, IT system, microservices architecture

Spis treści

1. Wstęp

Zgodnie z wynikami badań opublikowanymi przez Institute for market transformation z 2015 roku około 40% całkowitej konsumpcji energii w Stanach Zjednoczonych przypada na zasilanie budynków [1]. Przekłada się to w ciągu roku na wydatek rzędu 450 miliardów dolarów. Najsłabiej zagospodarowane budowle zużywają od trzech do siedmiu razy więcej energii od tych najbardziej oszczędnych. Istnieje zatem potrzeba przygotowania i wdrożenia rozwiązań, które zoptymalizowały by wykorzystanie zasobów energetycznych.

Utworzenie systemu miało przyczynić się do osiągnięcia dwóch głównych celów, stawianych od początku jego przygotowywania:

1. Poprawa warunków pracy - badania [2] dowodzą, że warunki panujące w pomieszczeniach do pracy mają wpływ na efektywność pracowników. Utrzymanie ich na optymalnym poziomie może spowodować wzrost wydajności do 2.5%
2. Redukcja zużywanej energii - w praktyce często zdarza się, że po zakończeniu pracy zostawiane są włączone światła na całą noc. Innym przykładem może być sytuacja, w której pomieszczenie jest ogrzewane, mimo iż nikt z niego nie korzysta. Przy wsparciu systemu będzie możliwe zapobieganie takim wydarzeniom, co w konsekwencji ograniczy zużycie energii

W odpowiedzi na przedstawione problemy zaprojektowano system do zarządzania warunkami technicznymi w pomieszczeniach biurowych oparty na architekturze mikrousługowej. Produkt ma na celu poprawę oraz kontrolę zgodności z zadanymi wartościami warunków panujących w pomieszczeniach przeznaczonych do pracy przy jednoczesnym ograniczeniu zużycia energii. Wybrane parametry przeznaczone do optymalizacji to temperatura oraz natężenie światła.

Implementacja projektu przewiduje umieszczenie w pomieszczeniach odpowiedniego rodzaju czujników, które będą na bieżąco monitorować stan danej przestrzeni. Zintegrowany z czujnikami system informacyjny powinien odczytywać przesyłane pomiary, a następnie je interpretować. Wynik interpretacji powinien być widoczny dla zainteresowanych osób. W wiadomości będą znajdować się informacje dotyczące działań, które należy podjąć, aby umożliwić ustalenie się badanych parametrów na właściwym poziomie.

Rozwiązanie było przygotowywane z myślą o jak najprostszym etapie wdrażania. Proces instalacji systemu w danym biurze powinien składać się z trzech kroków:

- Utworzenie nowej organizacji w bazie danych systemu
- Umieszczenie czujników w każdym z pomieszczeń, które powinno być monitorowane. Po podłączeniu do prądu czujniki powinny przesyłać pomiary
- Dodanie umieszczonych urządzeń do listy urządzeń w ramach danej organizacji

Procedura instalacji powinna przebiegać szybko, a po jej zakończeniu wszyscy użyt-

kownicy posiadający odpowiednie uprawnienia powinni otrzymywać wyniki pomiarów oraz rekomendacje dotyczące dalszych działań.

Dedykowana mikrousługa nasłuchuje na przychodzące pomiary, po czym przesyła je dalej w celu analizy. W wyniku procesu odpowiednia mikrousługa przetwarzania generuje odpowiedź, która zawiera opis aktualnego stanu danego pomieszczenia. Zależnie od warunków odpowiedź może przyjmować różną postać, np.:

- 'Aktualne warunki są zgodne z oczekiwaniami.' - jeśli wartości mierzonych parametrów zgadzają się z ustaleniami
- 'Zbyt wysoka temperatura. Rozważ uchylenie okna.' - jeśli pomiar temperatury wykracza ponad dozwoloną górną granicę
- 'Jest zbyt ciemno. Rozważ włączenie światła.' - gdy pomiar natężenia światła wypada poniżej ustalonego dolnego poziomu

Oczekiwane wartości mogą zostać wyznaczone przez administratora organizacji oraz przez pracownika, który jest do danego pomieszczenia przypisany. W oczekiwaniach określane są wartości temperatury oraz natężenia światła, a także dopuszczalne odchylenia od pożądanych wartości. Taki zbiór informacji określany jest jako reguła.

Dla każdego z pomieszczeń można określić wiele reguł, z których każda będzie obowiązywać w innym okresie w ciągu tygodnia. Przykładowo, dobrą praktyką byłoby określenie odrębnej reguły obowiązującej w ciągu godzin pracy, gdzie oczekiwana wartość natężenia światła wynosiłaby 600 lx oraz innej reguły, obowiązującej poza godzinami pracy, gdzie oczekiwana wartość natężenia światła wynosiłaby 200 lx. Po przekroczeniu tej granicy system poinformowałby użytkowników, że w danej sali najprawdopodobniej pozostało zapalone światło. System automatycznie wykrywałby te i inne nieprawidłowości, które można by następnie wyeliminować poprzez włączenie-/wyłączenie światła lub ogrzewania.

Wewnątrz systemu zostały zdefiniowane reguły domyślne obowiązujące tak długo, dopóki użytkownicy nie utworzą swoich własnych. Wszystkie rodzaje reguł zostały przedstawione poniżej w kolejności obowiązywania:

- domyślna
- organizacyjna
- oddziałowa
- spersonalizowana

Reguły organizacyjne mają pierwszeństwo przed regułami domyślnymi. Obowiązują dla każdego oddziału i pomieszczeń należących do danej organizacji. Reguły oddziałowe nadpisują reguły organizacyjne i obowiązują we wszystkich pomieszczeniach wewnątrz danego wydziału. Reguły spersonalizowane posiadają największy priorytet i obowiązują wyłącznie dla konkretnego pomieszczenia, dla którego zostały utworzone.

W kolejnych rozdziałach opisano zagadnienia związane z utworzeniem systemu. W rozdziale 2. przedstawiono różnice między już istniejącymi rozwiązaniami a propono-

wanym rozwiązaniem. Rozdział 3. został poświęcony założeniom, na których opierano się podczas tworzenia pracy. Treść rozdziału 4. zawiera porównanie dwóch konkurencyjnych architektur systemu oraz argumenty przemawiające za wyborem architektury opartej na mikrousługach. W rozdziale 5. przybliżono zagadanienia związane z modelowaniem danych oraz ich przechowywaniem. Rozdział 6. opisuje różne sposoby komunikacji między mikrousługami, wykorzystane w ramach pracy. Przygotowano zestaw testów sprawdzających poprawność działania systemu, przedstawionych szczegółowo w rozdziale 7. Treść rozdziału 8. zawiera szczegóły implementacyjne dotyczące dokonywania pomiarów oraz przesyłania ich w taki sposób, aby wiadomości były zrozumiałe dla systemu. Kolejny rozdział przybliża zasady działania oraz możliwości oferowane przez konteneryzację systemu. Rozdział 10. stanowi podsumowanie dokonań. Ostatni rozdział prezentuje możliwe dalsze kierunki rozwoju pracy.

2. Istniejące rozwiązania

Firma Sharp przygotowała podobne rozwiązanie, za pomocą którego można mierzyć kluczowe parametry danego pomieszczenia, przesyłać je na platformy chmurowe i analizować [3]. Różnica między tym produktem a rozwiązaniem proponowanym w niniejszej pracy polega na tym, że w rozwiązaniu firmy Sharp czujniki są wbudowane w monitor służący jako centrum telekonferencyjne. W ten sposób wykonywane pomiary stają się raczej dodatkiem do monitora, niż głównym powodem wstawienia urządzenia do konkretnej sali. W konsekwencji, wykonywanie pomiarów w wielu salach wiązałoby się z koniecznością zakupu drogiego monitora dla każdej z nich. Proponowane w tej pracy rozwiązanie zawiera jedynie zestaw czujników zintegrowanych z systemem, bez innych dodatków, co znacznie minimalizuje koszt wdrożenia takiego rozwiązania. Dzięki temu opłacalne staje się zbieranie pomiarów z wielu sal jednocześnie.

3. Założenia

W tej sekcji przedstawiono główne założenia dotyczące rozwoju pracy, a także przeprowadzono przegląd prac naukowych estymujących optymalną wartość temperatury oraz natężenia światła w pomieszczeniach biurowych.

Funkcjonalność i architektura systemu została zaprojektowana w oparciu o kilka istotnych założeń:

- Szczególny nacisk jest położony na łatwość wdrożenia - system powinien być gotowy do wdrożenia na środowisko chmurowe. Organizacja zainteresowana uruchomieniem systemu dla swoich potrzeb może wybrać opcję, w której dostarczane są obrazy odpowiednich mikroserwisów oraz skrypty konfigurujące środowisko. Takie rozwiązanie mogłoby być ofertą skierowaną do banków, które chcą zminimalizować ruch zewnętrzny. Może także skorzystać z opcji, w której system jest hostowany na serwerach firmy będącej autorem oprogramowania
- System składa się z czujników zbierających pomiary, które następnie przesyłane są do mikroserwisów, które je przetwarzają. Do pomiaru zalicza się aktualna temperatura oraz natężenie światła
- System oparty jest na regułach określających oczekiwaną wartość powyższych parametrów w danej chwili czasu. Po otrzymaniu każdego z pomiarów porównywane są wartości oczekiwane z rzeczywistymi i na tej podstawie przygotowywany jest wynik. Domyślnie istnieje reguła podstawowa, gdzie oczekiwana temperatura wynosi 24.5 °C, natomiast oczekiwane natężenie światła wynosi 550 lx. Więcej informacji odnośnie ustalonych wartości zawarto w tabeli 3.1 oraz tabeli 3.2.
- System przewiduje dwie role użytkowników: pracowników danej organizacji, którzy mogą tworzyć własne reguły dla pomieszczeń do nich przypisanych, oraz administratorów organizacji, którzy posiadają wszystkie uprawnienia przypisane pracownikom, a ponadto możliwość zarządzania informacjami dotyczącymi organizacji, budynków, pomieszczeń i czujników
- System jest rozwijany przy wykorzystaniu platformy .NET w wersji 5.0

Tabela 3.1 pokazuje porównanie wyników z różnych artykułów traktujących o optymalnej temperaturze w pomieszczeniach:

Tabela 3.1. Porównanie wyników badań estymujących optymalną temperaturę

Badanie	Optymalna temperatura
[4]	23.5°C - 25.5°C
[5]	23.0°C - 26.5°C
[6]	24.0°C - 25.0°C

W oparciu o powyższe badania wyliczono optymalną temperaturę jako średnią z uzyskanych rezultatów. Jej wartość wynosi 24.5°C. Natomiast w tabeli 3.2 porównywane są rezultaty badań nad optymalnym poziomem natężenia światła, który skutkował najlepszą efektywnością pracowników:

Tabela 3.2. Porównanie wyników badań estymujących optymalne natężenie światła

Badanie	Optymalne natężenie światła
[7]	500 lx
[8]	600 lx

W oparciu o powyższe badania wyliczono optymalne natężenie światła jako średnią z uzyskanych rezultatów. Jej wartość wynosi 550 lx.

4. Metodyka

Treść sekcji zawiera zestawienie dwóch konkurencyjnych architektur wykorzystywanych do projektowania systemów informatycznych: monolitycznej oraz opartej na mikrousługach.

Największy nacisk w trakcie tworzenia pracy został położony na łatwość wdrożenia. W poniższych podrozdziałach przedstawiono kwestie, które należy wziąć pod uwagę podczas projektowania rozwiązania, które ma tę cechę spełniać.

4.1. Architektura systemu

Sekcja przedstawia porównanie architektury monolitycznej oraz mikrousługowej. Podrozdziały 4.1.1 - 4.1.4 przedstawiają zagadnienia, na które należy zwrócić uwagę, przygotowując system utworzony z wielu modułów usługowych. W podrozdziale 4.1.5 spisano wszystkie moduły, które składają się na zaprojektowany w ramach niniejszej pracy system.

Wśród możliwych architektur systemów można wyłonić dwie najważniejsze gałęzie: architektura monolityczna lub oparta na mikrousługach, które są małymi, niezależnymi od siebie modułami, wspólnie ze sobą współpracującymi. Pierwsza z opcji opiera się na idei polegającej na tym, że wszystkie usługi oferowane przez dany system są zamknięte w jednym projekcie i funkcjonują jako całość. Druga możliwość opiera się na rozdzieleniu usług i umieszczeniu ich w oddzielnych modułach, które działają niezależnie od siebie. Obydwie architektury posiadają swoje wady i zalety i wybór jednej z nich zależy od specyficznych potrzeb każdego projektu. W tabeli 4.1. przedstawiono porównanie obydwu architektur [9], które uwzględnia:

- Odporność systemu na awarie. Oferowane przez system usługi powinne być w każdym momencie dostępne dla użytkowników
- Skalowalność. Potrzeba skalowania wynika ze zbyt dużego obciążenia dla jednego lub większej liczby modułów w jednostce czasu. Brak dopasowania zasobów do aktualnego zapotrzebowania może prowadzić do tego, że system nie będzie odpowiadał na żądania użytkowników
- Łatwość wdrożenia. Architektura systemu nie powinna utrudniać wdrożenia nowych funkcjonalności
- Zespół deweloperski. Architektura systemu nie powinna wymagać zatrudnienia wielu programistów

Tabela 4.1. Porównanie popularnych architektur systemów

Cecha	System monolityczny	System oparty na architekturze mikrousługowej
Odporność systemu na awarie	W przypadku awarii w jednym punkcie, cały system przestaje działać	Awaria jednego z modułów niekoniecznie musi oznaczać niesprawność całego systemu
Skalowalność	Wymusza zwiększanie liczby uruchomionych instancji systemu, nawet w przypadku silnego zapotrzebowania jedynie na część oferowanych usług	Możliwość zwiększania liczby instancji tylko tych mikrousług, które w danym momencie są silnie obciążone
Łatwość wdrożenia	Nawet mała zmiana w kodzie systemu monolitycznego wymaga ponownego wdrożenia całego kodu	Możliwość szybkiego wdrożenia poprawek w obrębie danego mikroserwisu
Zespół deweloperski	Rozbudowany projekt zazwyczaj wymaga zespołu liczącego setki programistów, co utrudnia komunikację i zmniejsza efektywność pracy	Nie wymaga rozbudowanego zespołu, możliwość oddelegowania małej grupy pracowników do oddzielnych mikroserwisów

Biorąc pod uwagę założenia dotyczące projektu, lepszą decyzją jest wykorzystanie architektury mikrousługowej. Zestawienie z tabeli 4.1 jasno wskazuje, że konsekwencją dokonanego wyboru jest szybsze wdrożenie kolejnych wersji systemu.

4.1.1. Serwisy zorientowane usługowo

Docelowo, system oparty na architekturze mikrousługowej powinien składać się z mikroserwisów zorientowanych usługowo (ang. *service-oriented architecture*). Stanowią one konstrukcję, w której wiele modułów współpracuje ze sobą w celu zapewnienia zbioru funkcjonalności. Serwis oznacza tutaj oddzielny proces pracujący na danej maszynie. Procesy te komunikują się ze sobą przez sieć.

4.1.2. Sprzężenie serwisów

Architektura mikrousługowa opiera się na tym, że poszczególne mikroserwisy działają niezależnie od siebie. Dzięki temu zmiany wprowadzone w jednym z modułów nie powinny wymagać zmian w drugim. Ponadto wdrożenie danego mikroserwisu nie powinno wymagać jednoczesnego wdrożenia innych. O tak rozdzielonych mikroserwisach mówi się, że są ze sobą luźno sprzężone (ang. *loose coupling*). Prawidłowo skonstruowany mikroserwis powinien wiedzieć jedynie tyle, w jaki sposób może komunikować się z innymi mikroserwisami w celu uzyskania wymaganych danych.

4.1.3. Spójność mikroserwisów

W prawidłowo skonstruowanym systemie mikrousługowym funkcjonalność związana ze sobą (np. w kontekście biznesowym) jest umieszczona w jednym miejscu. O tak zaprojektowanych modułach mówi się, że są one spójne (ang. *high cohesion*). Przykładem błędnej implementacji może być edycja danych osobowych klienta w wielu mikroserwisach. Wtedy zmiana w jednym module może wymagać zmiany w innych.

4.1.4. Wyznaczanie granic między mikroserwisami

Jednym z wyzwań stojących przed projektantem systemu jest wyznaczenie granic między mikroserwisami. Każdy z mikroserwisów powinien być niezależny od innych. Aby to osiągnąć, należy skupić się na wyznaczeniu odrębnych modeli danych wykorzystywanych przez system [10]. Każdy z modeli może odgrywać różną rolę w kontekście biznesowym. Dzięki podziałowi modelu danych na mniejsze fragmenty można utworzyć oddzielne moduły, każdy z których będzie odpowiedzialny za osobny fragment. W konsekwencji uzyskiwana jest niezależność mikroserwisów. Mogą one komunikować się między sobą w celu otrzymania wymaganych informacji.

4.1.5. Architektura systemu do zarządzania energią w pomieszczeniach biurowych

Opierając się na poprzednich podrozdziałach oraz w oparciu o założenia utworzono architekturę systemu będącego rezultatem tej pracy inżynierskiej. Rysunek 4.1 przedstawia pełny zestaw modułów składających się na całość rozwiązania:

W tabeli 4.2 opisano rolę mikroserwisów danych.

Tabela 4.2. Mikroserwisy danych

Nazwa	Funkcja
Addresses data service	Przechowuje adresy organizacji oraz poszczególnych budynków
Facilities data service	Przechowuje szczegółowe dane dotyczące budynków

Policies data service	Przechowuje reguły określające oczekiwaną wartość mierzonych parametrów
Sensors data service	Przechowuje szczegółowe dane dotyczące wykorzystywanych czujników
Sensor state history data service	Przechowuje historyczne pomiary z poszczególnych czujników
Sensor state data service	Przesyła pomiary od czujników

Poza mikroserwisami danych aplikacja posiada również mikroserwisy przetwarzające dane, opisane w tabeli 4.3:

Tabela 4.3. Mikroserwisy przetwarzające

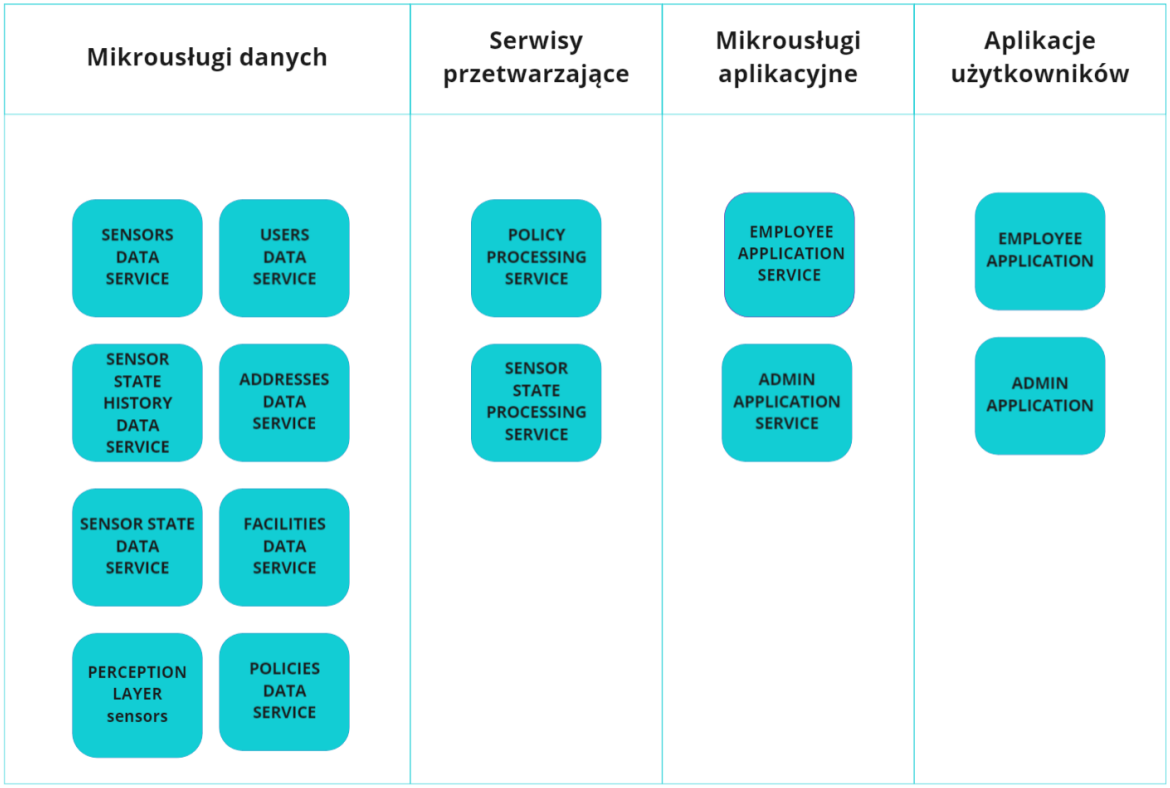
Nazwa	Funkcja
Sensor state processing service	Otrzymuje dane z czujników. Zajmuje się ich prawidłowym zapisaniem, po czym wysyła je dalej do PPS
Policy processing service	Przetwarza dane z czujników. Porównuje pomiary rzeczywiste z oczekiwanymi, które zostały określone w regułach

Na system składają się także mikroserwisy aplikacyjne opisane w tabeli 4.4:

Tabela 4.4. Mikroserwisy aplikacyjne

Nazwa	Funkcja
Employee application service	Mikrousługa aplikacyjna dla aplikacji pracowników. Oferuje usługi umożliwiające zarządzanie kontem, tworzenie własnych reguł dla pomieszczeń przypisanych do konkretnego użytkownika oraz sprawdzanie ich aktualnego stanu
Admin application service	Mikrousługa aplikacyjna dla aplikacji administratorów. Oferuje wszystkie usługi udostępniane pracownikom, a ponadto usługi umożliwiające zarządzanie informacjami dotyczącymi organizacji, budynków, pomieszczeń i czujników

Ostatnimi elementami systemu są aplikacje dla poszczególnych ról użytkowników, opisane w tabeli 4.5:



Rysunek 4.1. Architektura systemu. Opracowanie własne

Tabela 4.5. Aplikacje użytkowników

Nazwa	Funkcja
Employee application	Oferuje graficzny interfejs do interakcji z mikrousługą aplikacyjną pracowników
Admin application	Oferuje graficzny interfejs do interakcji z mikrousługą aplikacyjną administratorów

4.2. Zwinne zarządzanie

Podczas tworzenia systemu zastosowano metodykę Scrum, będącą metodyką zarządzania projektem typu zwinnego[11]. Cechuje się iteracyjnym podejściem do implementacji systemu. Metodyka zakłada trzy fazy rozwoju. W pierwszej definiuje się ogólne założenia dotyczące projektu, zakłada jego cele oraz zastosowania. Przygotowuje się także architekturę systemu, składającą się z mikroserwisów, baz danych, brokerów wiadomości. W następnej fazie następuje seria cykli tzw. sprintów, gdzie każdy cykl oznacza iteracyjne wzbogacanie systemu o nowe funkcjonalności. W ostatniej fazie następuje zakończenie rozwoju projektu, jego udokumentowanie, po czym spisuje się nabyte doświadczenia.

Na początku każdego ze sprintów definiowany jest zakres pracy, który powinien

zostać ukończony. Na podstawie ustalonych wymagań dzieli się pracę na mniejsze, niezależne od siebie fragmenty. Następnie każdy członek zespołu deweloperskiego wybiera jedno z dostępnych zadań i implementuje rozwiązanie. Sprint zazwyczaj trwa od dwóch do czterech tygodni.

5. Przechowywanie danych

Poszczególne mikroserwisy odwołują się do różnych źródeł danych w celu uzyskania wymaganych informacji. W tej pracy wykorzystano dwa różne sposoby przechowywania informacji:

- Relacyjna baza danych MySQL
- Baza danych szeregów czasowych InfluxDB

Podrozdział 5.1 przybliża szczegóły dotyczące utworzonych modeli danych oraz sposobu przechowywania informacji w relacyjnej bazie danych. W podrozdziale 5.2 opisano sposób przechowywania wykonanych przez czujniki pomiarów w bazie danych szeregów czasowych.

5.1. MySQL

Sekcja opisuje utworzone modele danych oraz sposób przechowywania informacji w relacyjnej bazie danych. Podrozdziały 5.1.1 - 5.1.5 przedstawiają szczegółowo kolejne schematy danych.

Relacyjna baza danych MySQL oferuje szybki, wielowątkowy serwer bazodanowy w oparciu o język SQL (ang. *Structured Query Language*) [12]. Została ona wybrana ze względu na to, że jest produktem typu open-source dostępnym na licencji GNU (ang. *general public license*).

Dobrą praktyką, którą warto mieć na uwadze podczas tworzenia systemu opartego na architekturze mikrousługowej, jest zapewnienie dostępu do konkretnego modelu danych tylko jednemu mikroserwisowi, który następnie może udostępniać posiadane informacje przy pomocy odpowiednio skonfigurowanego API (ang. *Application Programming Interface*) [13]. Takie podejście umożliwia zachowanie luźnego sprzężenia między mikroserwisami. W konsekwencji należy utworzyć oddzielne modele danych, zwane schematami, które mogą być zarządzane przez pojedynczy mikroserwis.

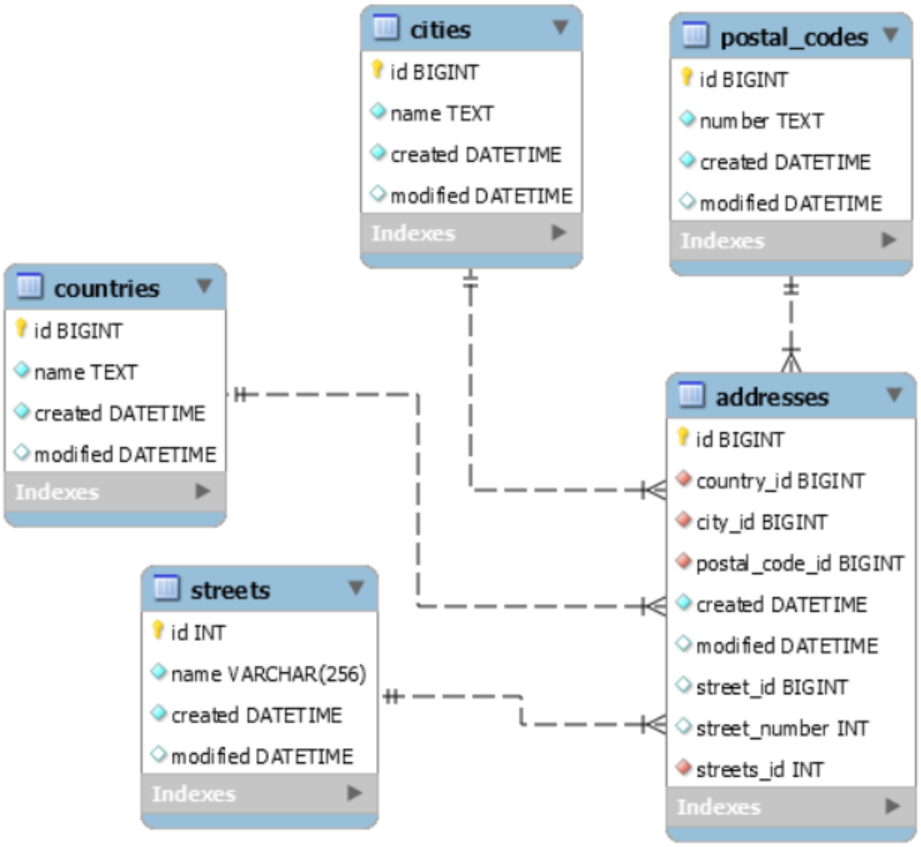
W ramach utworzonego serwera bazodanowego zostały wdrożone schematy opisane w tabeli 5.1:

Tabela 5.1. Utworzone schematy bazodanowe

Nazwa schematu	Funkcja
Addresses (adresy)	Przechowuje dane dotyczące adresów
Facilities (organizacje)	Przechowuje dane dotyczące organizacji oraz budynków
Identity (użytkownicy)	Przechowuje dane użytkowników
Policies (reguły)	Przechowuje dane dotyczące reguł określających oczekiwaną wartość mierzonych parametrów
Sensors (sensory)	Przechowuje dane dotyczące wykorzystywanych sensorów

5.1.1. Schemat adresów

Rysunek 5.1 przedstawia diagram UML modelu danych adresów.



Rysunek 5.1. Diagram modelu danych adresów. Opracowanie własne

Każda encja posiada unikalny identyfikator oraz kolumny opisujące czas utworzenia oraz ewentualnej modyfikacji. Encja *addresses* odwołuje się do pozostałych encji za pomocą odpowiadających im identyfikatorów.

Schemat adresów składa się z encji opisanych w tabeli 5.2:

Tabela 5.2. Encje w schemacie adresów

Nazwa encji	Przechowywane dane
Addresses	Encja główna, zawiera odwołania do kraju, miasta, kodu pocztowego, numeru ulicy oraz numeru budynku
Cities	Miasto
Countries	Kraj
Postal codes	Kod pocztowy
Streets	Ulica

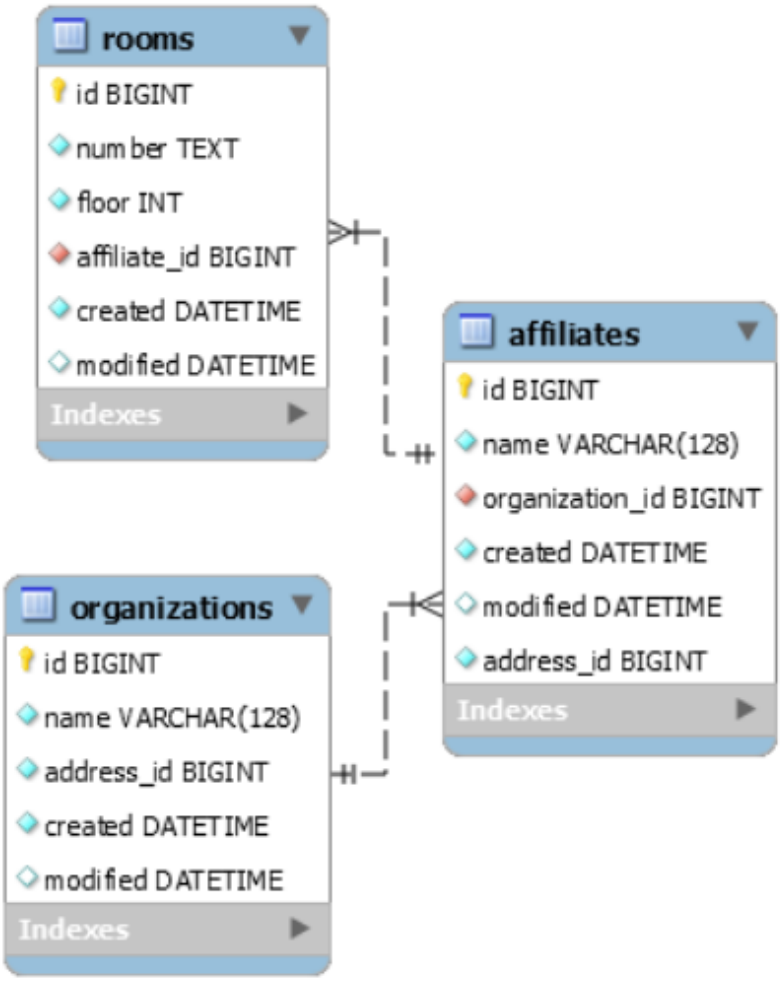
Związki między poszczególnymi encjami zostały opisane w tabeli 5.3.

Tabela 5.3. Związki między encjami w schemacie adresów

Relacja		Typ związku	Opis
Addresses	Countries	1:N	Każdy kraj może występować w wielu adresach
Addresses	Cities	1:N	Każde miasto może występować w wielu adresach
Addresses	Postal codes	1:N	Każdy kod pocztowy może występować w wielu adresach
Addresses	Strees	1:N	Każda ulica może występować w wielu adresach

5.1.2. Schemat organizacji

Rysunek 5.2 przedstawia diagram UML modelu danych organizacji.



Rysunek 5.2. Diagram modelu danych organizacji. Opracowanie własne

Każda z encji posiada unikalny identyfikator oraz kolumny opisujące czas utworzenia oraz ewentualnej modyfikacji. Oddziały są skojarzone z organizacjami poprzez przechowywanie powiązanego z nią identyfikatora *organization_id*. Pomieszczenia natomiast powiązane są z konkretnym oddziałem, również za pomocą stosownego identyfikatora *affiliate_id*.

Schemat organizacji składa się z encji opisanych w tabeli 5.4:

Tabela 5.4. Encje w schemacie organizacji

Nazwa encji	Przechowywane dane
Affiliates	Oddziały danej organizacji
Rooms	Pomieszczenia w oddziałach
Organizations	Organizacje

Związki między poszczególnymi encjami zostały opisane w tabeli 5.5.

Tabela 5.5. Związki między encjami w schemacie organizacji

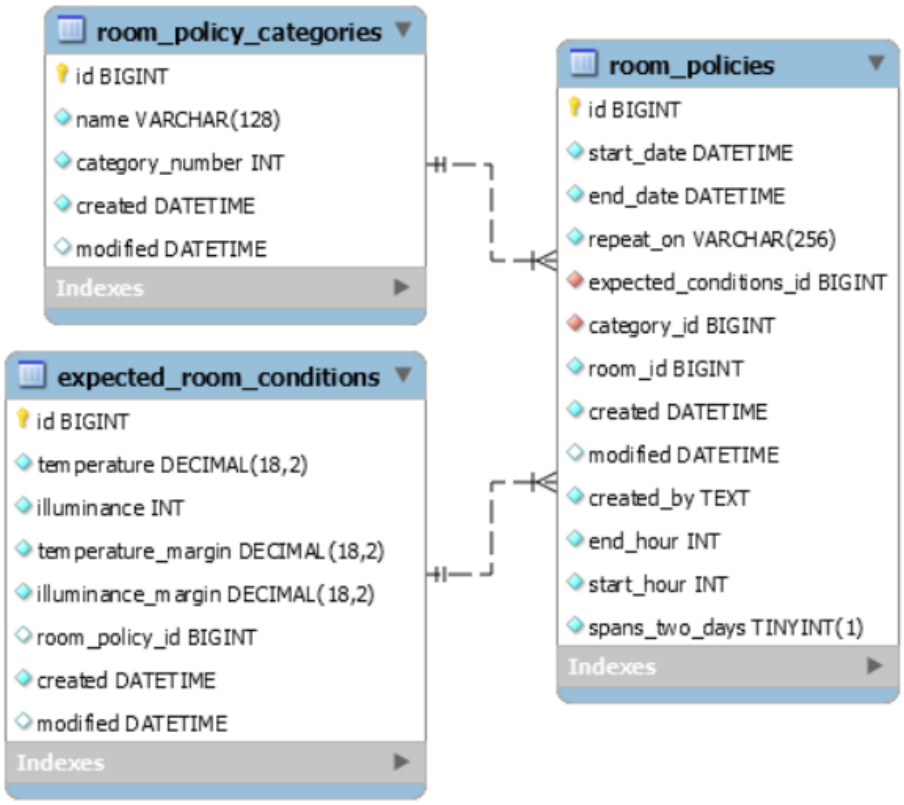
Relacja		Typ związku	Opis
Organizations	Affiliates	1:N	Każdy organizacja może posiadać wiele oddziałów
Affiliates	Rooms	1:N	Każdy oddział może posiadać wiele pomieszczeń

5.1.3. Schemat reguł

Rysunek 5.3 przedstawia diagram UML modelu danych reguł. Każda z encji posiada unikalny identyfikator oraz kolumny opisujące czas utworzenia oraz ewentualnej modyfikacji. Encja *expected_room_conditions* przechowuje zestawy oczekiwanych warunków, jakie powinny panować w danym pomieszczeniu. Poza konkretną wartością temperatury oraz natężenia światła określa się także maksymalne tolerowane odchylenie wartości rzeczywistych od oczekiwanych.

Encja *room_policies* zawiera odwołanie do *expected_room_conditions* poprzez przechowywanie identyfikatora *expected_conditions_id*. Zawiera ona także daty, przechowywane w kolumnach *start_date* oraz *end_date* oraz oznaczające okres, w jakim dana reguła powinna obowiązywać. Godziny obowiązywania zawarte są w kolumnach *start_hour* oraz *end_hour*.

Schemat reguł składa się z encji opisanych w tabeli 5.6:



Rysunek 5.3. Diagram modelu danych reguł. Opracowanie własne

Tabela 5.6. Encje w schemacie reguł

Nazwa encji	Przechowywane dane
Room policies	Encja główna, zawiera odwołania do kategorii oraz oczekiwanych warunków. Przechowuje okres obowiązywania danej reguły
Room policy categories	Kategorie reguł
Expected room conditions	Oczekiwane warunki w pomieszczeniu

Związki między poszczególnymi encjami zostały opisane w tabeli 5.7.

Tabela 5.7. Związki między encjami w schemacie reguł

Relacja		Typ związku	Opis
Room policies	Room policy categories	1:N	Każda kategoria może być przypisana wielu regułom
Room policies	Expected room conditions	1:N	Ten sam zbiór oczekiwanych warunków może być przypisany do wielu reguł

5.1.4. Schemat sensorów

Rysunek 5.4 przedstawia diagram UML modelu danych sensorów.

Rysunek 5.4. Diagram modelu danych sensorów. Opracowanie własne

Każda z encji posiada unikalny identyfikator oraz kolumny opisujące czas utworzenia oraz ewentualnej modyfikacji. Producenci określają swoją nazwę oraz datę

założenia działalności. Produkują modele sensorów, które opisane są w encji *sensor_models*. Zawiera ona stosowny identyfikator *producer_id* łączący obydwie encje. Każdy z modeli posiada własną nazwę oraz numer wersji.

Rekordy encji *sensors* reprezentują kolejne instancje konkretnego modelu sensora. Encje są ze sobą powiązane przy pomocy identyfikatora *sensor_model_id*. *Sensors* zawiera także odwołania do oddziału oraz pomieszczenia, w którym znajduje się urządzenie pomiarowe, przy użyciu identyfikatorów *affiliate_id* oraz *room_id*.

Szczegółowe dane dotyczące oddziałów oraz pomieszczeń są przechowywane w schemacie organizacji. Aby zachować niskie sprzężenie oraz wysoką spójność serwisów, wewnątrz schematu sensorów należy się odwoływać do tych obiektów wyłącznie za pomocą unikalnego identyfikatora. Nie powinno się także umieszczać dodatkowych kolumn opisujących oddział lub pomieszczenie.

Schemat sensorów składa się z encji opisanych w tabeli 5.8:

Tabela 5.8. Encje w schemacie sensorów

Nazwa encji	Przechowywane dane
Sensors	Szczegóły dotyczące danego sensora
Sensor models	Model sensora
Producers	Producent sensorów

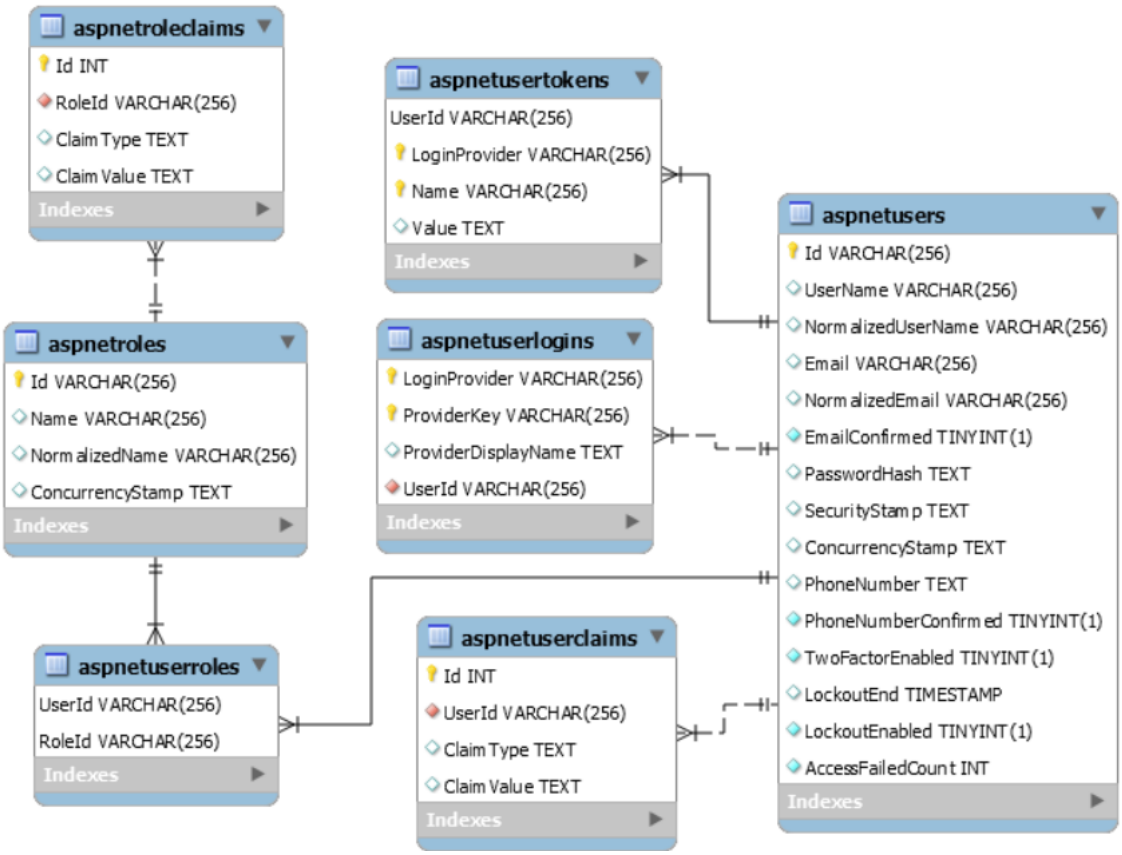
Związki między poszczególnymi encjami zostały opisane w tabeli 5.9.

Tabela 5.9. Związki między encjami w schemacie sensorów

Relacja		Typ związku	Opis
Sensor models	Producers	1:N	Każdy producent może oferować wiele modeli sensorów
Sensor models	Sensors	1:N	Każdy model może być wyporodukowany wielokrotnie

5.1.5. Schemat użytkowników

Rysunek 5.5 przedstawia diagram UML modelu danych użytkowników.



Rysunek 5.5. Diagram modelu danych użytkowników. Opracowanie własne

Schemat użytkowników został oparty na schemacie oferowanym przez Microsoft [14] i składa się z następujących encji:

Tabela 5.10. Encje w schemacie użytkowników

Nazwa encji	Przechowywane dane
Asnetusers	Reprezentuje użytkownika
Aspnetroles	Reprezentuje rolę użytkownika w systemie
Aspnetuserlogins	Łączy użytkownika z loginem
Aspnetusertokens	Reprezentuje token uwierzytelniający dla użytkownika
Aspnetuserclaims	Reprezentuje prawa, które posiada użytkownik
Aspnetroleclaims	Reprezentuje prawa gwarantowane dla wszystkich użytkowników pełniących daną rolę
Aspnetuserroles	Łączy użytkowników z poszczególnymi rolami

Związki między poszczególnymi encjami zostały opisane w tabeli 5.11.

Tabela 5.11. Związki między encjami w schemacie użytkowników

Relacja		Typ związku	Opis
Aspnetusers	Aspnetuserlogins	1:N	Każdemu użytkownikowi może być przypisanych wiele loginów
Aspnetusers	Aspnetusertokens	1:N	Każdemu użytkownikowi może być przypisanych wiele tokenów
Aspnetusers	Aspnetuserroles	1:N	Każdy użytkownik może pełnić wiele ról
Aspnetusers	Aspnetuserclaims	1:N	Każdy użytkownik może posiadać wiele praw
Aspnetroles	Aspnetroleclaims	1:N	Każdej roli może być przypisanych wiele praw
Aspnetroles	Aspnetuserroles	1:N	Każda rola może być pełniona przez wielu użytkowników

5.2. InfluxDB

Sekcja opisuje sposób przechowywania wykonanych przez czujniki pomiarów w bazie danych szeregów czasowych.

InfluxDB jest bazą danych szeregów czasowych służącą do przechowywania metryk i szeregów czasowych [15]. Umożliwia efektywne przechowywanie miliardów rekordów dzięki stosowaniu kompresji danych, oferuje także język zapytań pozwalający przeprowadzać kompleksowe analizy uzyskanych informacji. Dodatkową funkcjonalnością wyróżniającą tą bazę od standardowych relacyjnych baz danych jest możliwość zdefiniowania czasu, po którym rekordy będą usuwane. Można w ten sposób określić na przykład, że baza powinna przechowywać tylko rekordy z ostatniego miesiąca.

W niniejszej pracy baza danych szeregów czasowych została wykorzystana do przechowywania danych zbieranych z czujników temperatury oraz natężenia światła. Przychodzące informacje są zapisywane w kolejnych rekordach, które zawierają przedstawione w tabeli 5.12 parametry:

Tabela 5.12. Parametry pojedynczego rekordu przechowującego pomiar

Parametr	Znaczenie
time	Czas otrzymania danych
field	Rodzaj danych
value	Zmierzona wartość

Domyślnie czas jest zapisywany z dokładnością do nanosekund. W tej pracy zostały zdefiniowane dwa rodzaje danych: *temperature* dotyczące informacji o temperaturze oraz *illuminance* dotyczące informacji o natężeniu światła.

6. Komunikacja między mikroserwisami

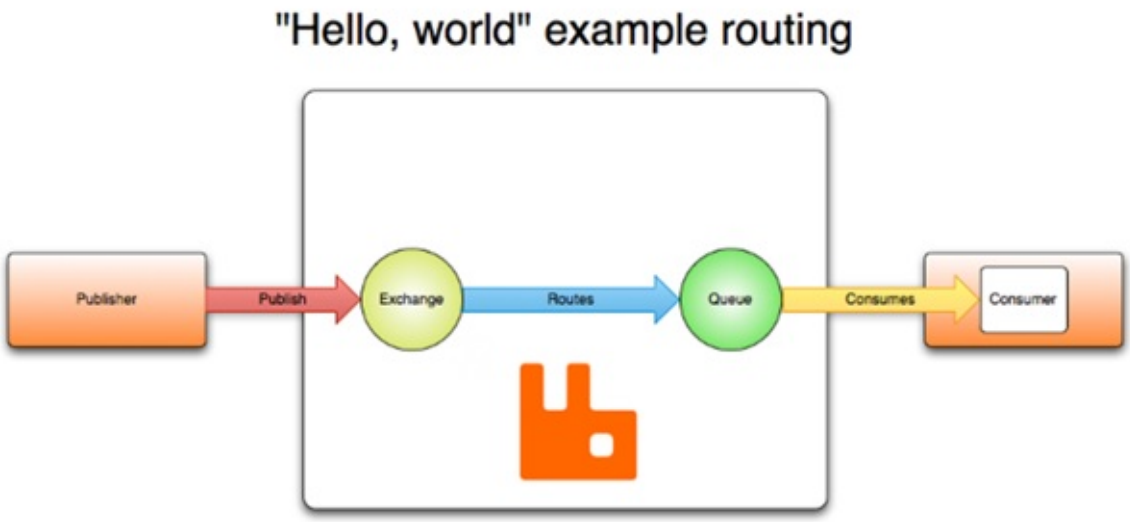
Sekcja przedstawia metody wykorzystane do przesyłania informacji między mikroserwisami. Opisano rolę i różnice między brokerem wiadomości a stykami oferowanymi przez poszczególne moduły usługowe. Podrozdział 6.1 przybliża szczegóły dotyczące zasad działania brokera wiadomości. W podrozdziale 6.2 przedstawiono listę usług oferowanych przez kolejne mikroserwisy.

6.1. Broker wiadomości

Sekcja opisuje wykorzystanie brokera wiadomości jako narzędzia do przekazywania wiadomości między serwisami. Podrozdziały 6.1.1 oraz 6.1.2 prezentują szczegóły dotyczące szyny danych oraz jej wykorzystania.

Aby mikroserwisy mogły działać poprawnie, potrzebują wymieniać ze sobą informacje w sposób szybki i niezawodny. W przypadku komunikacji nie wymagającej żadnej interakcji z użytkownikiem, jednym z rozwiązań jest wykorzystanie brokera wiadomości, który pełni rolę pośrednika przekazującego wiadomości między mikroserwisami. Do grona takich narzędzi należy RabbitMQ. Spośród konkurencyjnych rozwiązań wyróżnia się tym, że jest to narzędzie typu open source, ponadto cechuje się małymi wymaganiami sprzętowymi oraz łatwością wdrażania w chmurze.

RabbitMQ wspiera różne protokoły do przekazywania wiadomości, jednak domyślnie wykorzystuje protokół AMQP 0-9-1 (ang. *advanced message queuing protocol*). Uogólniony schemat, na którym opiera się protokół, został przedstawiony na rysunku 6.1.



Rysunek 6.1. Schemat protokołu AMQP 0-9-1. Źródło: [16]

Wydawcy wiadomości (ang. *publisher*) publikują wiadomości do pośrednika, który następnie je przekazuje do odpowiednich konsumentów (ang. *consumer*). Ponieważ jest to protokół sieciowy, to wydawcy, konsumenci oraz pośrednicy mogą być uruchomieni na różnych maszynach. Pośrednik RabbitMQ działa według następujących zasad:

- Wiadomości są publikowane na giełdy (ang. exchange)
- Giełdy mogą być połączone z wieloma kolejkami (ang. *queue*)
- Zależnie od zastosowanej polityki kopia wiadomości może być przekazana do każdej powiązanej kolejki lub tylko podzbiorowi kolejek
- Kolejka po otrzymaniu wiadomości od giełdy przesyła ją do konsumentów

Przesyłanie informacji przez sieć wiąże się z ryzykiem tego, że dane nie zostaną dostarczone. Wobec tego protokół zapewnia funkcjonalność zwaną potwierdzeniem wiadomości (ang. *message acknowledgements*), która gwarantuje otrzymanie wiadomości przez konsumentów. Działa ona w ten sposób, że wiadomość jest usuwana z kolejki tylko wtedy, gdy uzyska potwierdzenie jej otrzymania od zainteresowanych mikrousług.

Giełdy po otrzymaniu wiadomości od wydawców mogą ją rozesłać do zera lub większej liczby kolejek. Używany algorytm routingu zależy od typu wymiany i reguł nazywanych powiązaniami (ang. *bindings*). Pośrednicy korzystający z protokołu AMQP 0-9-1 zapewniają cztery typy wymiany:

- Wymiana bezpośrednia (ang. *direct exchange*) - dostarcza wiadomości do kolejek na podstawie klucza routingu. Jest idealna do wiadomości typu unicast (chociaż może być także używana do wiadomości typu multicast)
- Wymiana do wszystkich (ang. *fanout exchange*) - kieruje komunikaty do wszystkich kolejek, które są z nią powiązane, a klucz routingu jest ignorowany. Jeśli do giełdy dowiązanych jest N kolejek, to po publikacji wiadomości przez wydawcę dotrze ona do wszystkich N kolejek. Jest idealna do rozsyłania wiadomości do wszystkich konsumentów
- Wymiana tematyczna (ang. *topic exchange*) - kieruje wiadomości do jednej lub wielu kolejek na podstawie dopasowania klucza routingu oraz wzorca użytego do powiązania kolejki z giełdą. Jest często używana do implementacji różnych odmian wzorca publish/subscribe. Typowo używa się jej dla wiadomości typu multicast. Warto ją rozważyć w przypadku, gdy należy dostarczyć wiadomość do wielu konsumentów, które selektywnie wybierają rodzaj wiadomości, które chcą otrzymywać
- Wymiana nagłówków (ang. *headers exchange*) - przeznaczona do routingu na podstawie wielu atrybutów, które łatwiej można wyrazić w postaci nagłówków wiadomości niż klucza routingu, który jest ignorowany. Wiadomość uważana jest

za zgodną i rozsyłana dalej, jeśli wartość nagłówka jest równa wartości określonej podczas wiązania

AMQP 0-9-1 jest protokołem poziomu aplikacji, który używa protokołu TCP do niezawodnego przesyłania wiadomości. Połączenia (ang. *connections*) korzystają z metod uwierzytelnienia i mogą być chronione za pomocą protokołu TLS. Ze względu na dodatkowe środki zapewniające niezawodność zaleca się, aby połączenia między klientem a pośrednikiem były ustanawiane na dłuższy okres czasu. W przypadku gdy klient potrzebuje nawiązać wiele połączeń, powinien wykorzystać kanały (ang. *channels*), o których można myśleć jak o lekkich połączeniach dzielących jedno połączenie TCP. Każda operacja przeprowadzana przez klienta odbywa się z użyciem kanału, a komunikacja na różnych kanałach jest od siebie całkowicie odseparowana. Z tego względu każda metoda zawiera identyfikator kanału dla rozróżnienia, przez który kanał należy wysłać wiadomość.

6.1.1. MassTransit

Przy tworzeniu systemu została wykorzystana szyna danych o nazwie MassTransit przeznaczona dla aplikacji napisanych w modelu (ang. *framework*) .NET Core [17]. Zapewnia ona poziom abstrakcji umożliwiający wykorzystanie wielu różnych pośredników wiadomości, w tym RabbitMQ. Spośród wielu swoich zalet, szyna zapewnia:

- Równoczesne, asynchroniczne przetwarzanie wiadomości dla zwiększenia przepustowości
- Zarządzanie połączeniem. Jeśli dany mikroserwis zostanie rozłączony z pośrednikiem wiadomości, MassTransit spróbuje połączyć się ponownie oraz przywrócić dotychczasowe giełdy, kolejki, a także połączenia między nimi
- Serializacja danych. Pośrednik wiadomości RabbitMq przesyła wiadomości w postaci bajtów. Aby za jego pomocą przesłać obiekty specyficzne dla języka C#, trzeba zapisać je w odpowiednim formacie, w procesie zwanym serializacją. MassTransit implementuje narzędzia do serializacji obiektów
- Testy jednostkowe. MassTransit zawiera implementację przygotowaną specjalnie do testów w taki sposób, by testy nie były zależne od reszty infrastruktury systemu. Przykładem jest poniższa metoda:

```
internal static async Task
PublishAndWaitToBeConsumed<T>(
    T @event,
    InMemoryTestHarness testHarness)
{
    var messageIdentifier =
        await PublishMessage(@event, testHarness);
```

```
        var messageHasBeenConsumed =
        await testHarness
            .Consumed
            .Any(x =>
            x.Context.MessageId == messageIdentifier);
        messageHasBeenConsumed.Should().BeTrue();

        var message =
        await testHarness!
            .Consumed
            .SelectAsync(x =>
            x.Context.MessageId == messageIdentifier)
            .First();

        message.Exception.Should()
            .BeNull(
                "Message has been consumed
                without any errors");
    }
```

Metoda publikuje testową wiadomość, po czym sprawdza, czy została prawidłowo przetworzona.

Dzięki zastosowaniu szyny danych tworzenie konsumentów oraz publikowanie wiadomości staje się dużo łatwiejsze. Aby utworzyć nowego konsumenta, wystarczy jedynie utworzyć nową klasę implementującą interfejs *IConsumer<T>*, gdzie *T* jest oczekiwanym typem wiadomości. Klasa musi zawierać implementację metody *Consume(ConsumeContext<MeasurementSentEvent> context)*, która zawiera logikę przetwarzania wiadomości. Przykładem jest poniższa metoda:

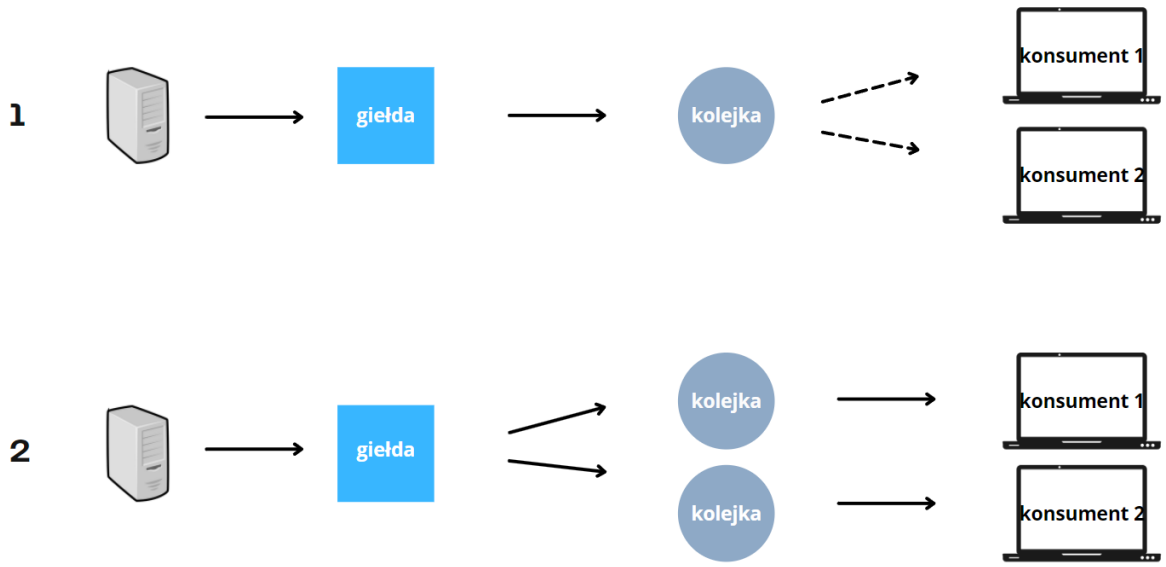
```
    public async Task Consume(
        ConsumeContext<MeasurementSentEvent> context)
    {
        var policiesEvaluationResultEvent =
        await _evaluatePoliciesCommand
            .Handle(context.Message);

        await _eventPublisher
            .Publish(policiesEvaluationResultEvent);
        _logger.LogInformation(
            $"PoliciesEvaluationResultEvent sent
            from PolicyNode. Message:
```

```
                {policiesEvaluationResultEvent.Message}");
    }
```

Zapisuje ona w logach szczegóły dotyczące przychodzącej wiadomości, przetwarza otrzymane wartości, po czym publikuje nową wiadomość o innym typie, której konsument jest implementowany przez inny mikroserwis.

6.1.2. Połączenie mikroserwisów z brokerem wiadomości

Utworzone w ramach pracy mikroserwisy łączą się z brokerem wiadomości i w ten sposób wymieniają między sobą dane. Mogą one być połączone w różny sposób, zależnie od oczekiwanego rezultatu. Dwa główne przypadki zostały przedstawione na rysunku 6.2.



Rysunek 6.2. Rodzaje połączenia z brokerem wiadomości. Opracowanie własne

1. W pierwszym przypadku celem jest wysłanie wiadomości tylko do jednego konsumenta. Jedną z występujących sytuacji jest przetwarzanie otrzymanych z czujników pomiarów przez SSPS. Wykonanie tej samej operacji przez kilka instancji tego serwisu byłoby stratą dostępnych zasobów. Aby osiągnąć ten efekt, wszystkie instancje tego serwisu powinny być dołączone do jednej kolejki. Wtedy kolejka wysyła przychodzące pomiary tylko do wybranej przez siebie instancji. Wybór odbywa się za pomocą mechanizmu szeregowania procesów Round-robin.
2. W drugim przypadku celem jest wysłanie wiadomości do wielu konsumentów. Jedną z występujących sytuacji jest wysłanie wyniku przetwarzania pomiarów do wszystkich pracowników oraz administratorów posiadających uprawnienia do jego wyświetlenia. Aby osiągnąć ten efekt, każda instancja serwisów otrzymujących wynik (w tym wypadku są to AAS oraz EAS) powinna być dołączona do osobnej kolejki, z których wszystkie dołączone są do tej samej giełdy publikującej

wiadomości. W ten sposób kopia wysłanej przez wydawcę wiadomości zostanie dostarczona do każdej instancji.

6.2. Styki

Sekcja przedstawia opis styków jako narzędzia do komunikacji między serwisami. W podrozdziałach 6.2.1 - 6.2.5 przedstawiono listę udostępnianych przez kolejne mikroserwisy usług, ich wymagane parametry wejściowe oraz rodzaj zwracanej odpowiedzi.

Mikroserwisy mogą komunikować się między sobą przy użyciu oferowanych przez nie styków (ang. *Application Programming Interface*). Definiują parametry wymagane do realizacji usługi oraz zbiór danych zwracany w odpowiedzi.

Mikrousługi aplikacyjne oraz mikrousługi danych oferują styki zwracające jasno zdefiniowany zbiór danych. Zostały one szczegółowo opisane w poniższych podrozdziałach. Każda z oferowanych usług zawiera:

- Krótki opis funkcjonalności
- Wykorzystywany czasownik protokołu http. Jeden z GET, POST, UPDATE, DELETE
- wymagane parametry wejściowe
- numer statusu oraz typ zwracanego obiektu

6.2.1. Usługi udostępniane przez mikrousługę danych adresów

ADS oferuje usługi zwracające szczegółowe dane dotyczące adresów oraz pozwalające na ich manipulację.

Tabela 6.1. Usługi udostępniane przez ADS

Ścieżka	GET /addresses-api/addresses/{addressId}	
Opis	Zwraca informacje na temat adresu o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	AddressDtoJsonApiDocument
	404	JsonApiError
Ścieżka	POST /addresses-api/addresses	
Opis	Tworzy nowy adres o parametrach podanych w modelu AddNewAddressCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	AddressDtoJsonApiDocument
	400	JsonApiError
Ścieżka	GET /addresses-api/cities/{cityId}	
Opis	Zwraca informacje na temat miasta o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**

	200	CityDtoJsonApiDocument
	404	JsonApiError

Ścieżka	GET /addresses-api/countries/{countryId}	
Opis	Zwraca informacje na temat kraju o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	CountryDtoJsonApiDocument
	404	JsonApiError

Ścieżka	GET /addresses-api/postal-codes/{postalCodeId}	
Opis	Zwraca informacje na temat kodu pocztowego o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	PostalCodeDtoJsonApiDocument
	404	JsonApiError

6.2.2. Usługi udostępniane przez mikrousługę danych organizacji

FDS udostępnia zbiór usług pozwalających na zarządzanie przechowywanymi informacjami o organizacjach, oddziałach oraz pomieszczeniach.

Tabela 6.2. Usługi udostępniane przez FDS

Ścieżka	GET /facilities-api/affiliates/{affiliateId}	
Opis	Zwraca informacje na temat oddziału o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	AffiliateDtoJsonApiDocument
	404	JsonApiError

Ścieżka	DELETE /facilities-api/affiliates/{affiliateId}	
Opis	Usuwa informacje na temat oddziału o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	204	Brak zawartości
	404	JsonApiError

Ścieżka	POST /facilities-api/organizations	
Opis	Tworzy nową organizację o parametrach podanych w modelu AddNewOrganizationCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	OrganizationDtoJsonApiDocument
	400	JsonApiError
	409	JsonApiError

Ścieżka	GET /facilities-api/organizations	
Opis	Zwraca informacje na temat wszystkich organizacji	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	OrganizationDtoListJsonApiDocument

Ścieżka	GET /facilities-api/organizations/{organizationId}	
Opis	Zwraca informacje na temat organizacji o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	OrganizationDtoJsonApiDocument
	404	JsonApiError

Ścieżka	DELETE /facilities-api/organizations/{organizationId}	
Opis	Usuwa informacje na temat organizacji o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	204	Brak zawartości
	404	JsonApiError

Ścieżka	POST /facilities-api/rooms	
Opis	Tworzy nowe pomieszczenie o parametrach podanych w modelu AddNewRoomCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	RoomDtoJsonApiDocument
	400	JsonApiError
	409	JsonApiError

Ścieżka	GET /facilities-api/rooms/{roomId}	
Opis	Zwraca informacje na temat pomieszczenia o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	RoomDtoJsonApiDocument
	404	JsonApiError

Ścieżka	DELETE /facilities-api/rooms/{roomId}	
Opis	Usuwa informacje na temat pomieszczenia o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	204	Brak zawartości
	404	JsonApiError

6.2.3. Usługi udostępniane przez mikrousługę danych reguł

Usługi oferowane przez PDS pozwalają na zwracanie informacji dotyczących istniejących reguł, a także dodawanie nowych rekordów o różnych typach reguł.

Tabela 6.3. Usługi udostępniane przez PDS

Ścieżka	POST /policies-api/expected-room-conditions	
Opis	Tworzy nowy zestaw oczekiwanych wartości mierzonych parametrów określonych w modelu AddNewExpectedRoomConditionsCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	OrganizationDtoJsonApiDocument
	400	JsonApiError

Ścieżka	GET /policies-api/expected-room-conditions/{expectedRoomConditionsId}	
Opis	Zwraca informacje na temat zestawu oczekiwanych wartości mierzonych parametrów o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	RoomDtoJsonApiDocument
	404	JsonApiError

Ścieżka	POST /policies-api/policies	
Opis	Tworzy nową regułę o parametrach określonych w modelu AddNewRoomPolicyCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	RoomPolicyDtoJsonApiDocument
	400	JsonApiError

Ścieżka	POST /policies-api/policies/past-policies	
Opis	Zwraca listę reguł poprzednio obowiązujących w pomieszczeniu określonym w modelu GetPastPoliciesCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	PastMeasurementsDtoJsonApiDocument
	400	JsonApiError

Ścieżka	GET /policies-api/policies/{roomId}	
Opis	Zwraca informacje na temat aktualnie obowiązującej polityki w pomieszczeniu o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	RoomPolicyDtoJsonApiDocument
	404	JsonApiError

6.2.4. Usługi udostępniane przez mikrousługę danych sensorów

SDS udostępnia usługi pozwalające na uzyskiwanie informacji o istniejących sensorach oraz dodawanie nowych rekordów.

Tabela 6.4. Usługi udostępniane przez SDS

Ścieżka	GET /sensors-api/sensors	
Opis	Zwraca informacje na temat wszystkich sensorów	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	SensorDtoListJsonApiDocument
Ścieżka	GET /sensors-api/sensors/room-id/{roomId}	
Opis	Zwraca informacje na temat sensorów znajdujących się w pomieszczeniu o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	SensorDtoJsonApiDocument
	404	JsonApiError
Ścieżka	POST /sensors-api/sensors/sensor-name	
Opis	Zwraca informacje na temat sensorów o parametrach określonych w modelu GetSensorBySensorNameCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	SensorDtoJsonApiDocument
	400	JsonApiError
Ścieżka	GET /sensors-api/sensors/{sensorId}	
Opis	Zwraca informacje na temat sensorów o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	SensorDtoJsonApiDocument
	404	JsonApiError

6.2.5. Usługi udostępniane przez mikrousługę aplikacyjną administratorów

Usługi udostępniane przez AAS pozwalają na uzyskiwanie danych dotyczących organizacji, oddziałów i pomieszczeń, a także na ich dodawanie.

Tabela 6.5. Usługi udostępniane przez AAS

Ścieżka	POST /admin-node/addresses	
Opis	Tworzy nowy adres o parametrach określonych w modelu AddNewAddressCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	AddressDtoJsonApiDocument
	400	JsonApiError
Ścieżka	POST /admin-node/affiliates	

Opis	Tworzy nowy oddział o parametrach określonych w modelu AddNewAffiliateCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	AffiliateDtoJsonApiDocument
	400	JsonApiError
	409	JsonApiError

Ścieżka	GET /admin-node/affiliates	
Opis	Zwraca informacje na temat wszystkich oddziałów	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	AdminNodeAffiliateDtoListJsonApiDocument
	404	JsonApiError

Ścieżka	GET /admin-node/affiliates/{affiliateId}	
Opis	Zwraca informacje na temat oddziału o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	AdminNodeAffiliateDtoJsonApiDocument
	404	JsonApiError

Ścieżka	DELETE /admin-node/affiliates/{affiliateId}	
Opis	Usuwa informacje na temat oddziału o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	204	Brak zawartości
	404	JsonApiError

Ścieżka	POST /admin-node/organizations	
Opis	Tworzy nową organizację o parametrach określonych w modelu AddNewOrganizationCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	OrganizationDtoJsonApiDocument
	400	JsonApiError
	409	JsonApiError

Ścieżka	GET /admin-node/organizations	
Opis	Zwraca informacje na temat wszystkich organizacji	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	AdminNodeOrganizationDto ListJsonApiDocument
	404	JsonApiError

Ścieżka	GET /admin-node/organizations/{organizationId}	
Opis	Zwraca informacje na temat organizacji o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**

	200	AdminNodeOrganizationDto ListJsonApiDocument
	404	JsonApiError

Ścieżka	DELETE /admin-node/organizations/{organizationId}	
Opis	Usuwa informacje na temat organizacji o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	204	Brak zawartości
	404	JsonApiError

Ścieżka	POST /admin-node/rooms	
Opis	Tworzy nowe pomieszczenie o parametrach określonych w modelu AddNewRoomCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	201	RoomDtoJsonApiDocument
	400	JsonApiError
	409	JsonApiError

Ścieżka	POST /admin-node/rooms/get-room	
Opis	Zwraca informacje o pomieszczeniu o parametrach podanych w modelu GetRoomCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	AdminNodeRoomDtoJsonApiDocument
	404	JsonApiError

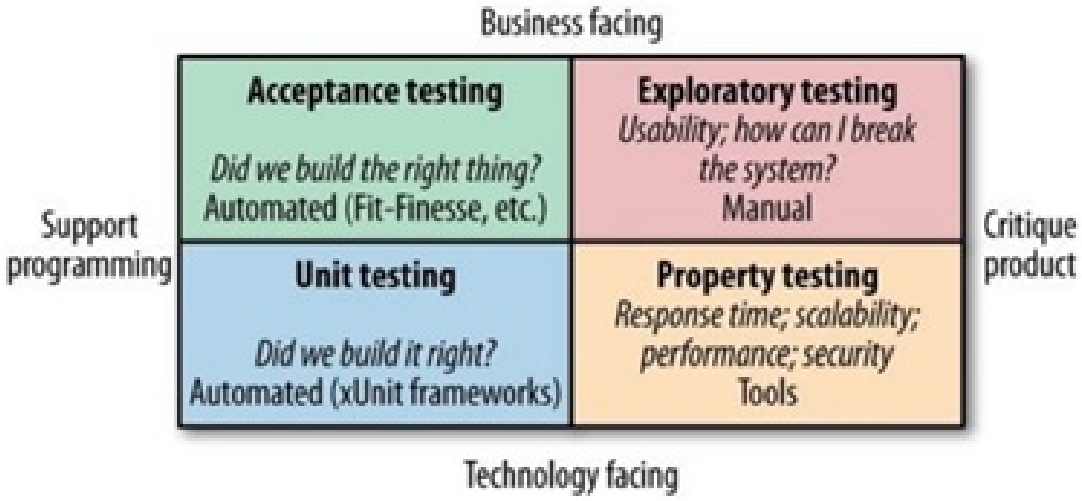
Ścieżka	POST /admin-node/rooms/historic-measurements	
Opis	Zwraca informacje o historycznych pomiarach z pomieszczenia o parametrach podanych w modelu GetRoomCommand	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	200	PastMeasurementsDtoJsonApiDocument
	400	JsonApiError

Ścieżka	DELETE /admin-node/rooms/{roomId}	
Opis	Usuwa informacje na temat pomieszczenia o danym identyfikatorze	
Odpowiedź	**Kod odpowiedzi**	**Schemat**
	204	Brak zawartości
	404	JsonApiError

7. Testy

Sekcja opisuje wykorzystane rodzaje testów oraz przedstawia przypadki ich użycia. Podrozdział 7.1 prezentuje szczegóły dotyczące testów jednostkowych oraz przedstawia argumenty przekonujące do zapewnienia dużej liczby tego rodzaju testów. W podrozdziale 7.2 opisano rolę testów integracyjnych. Poprawność funkcji biznesowych można sprawdzić dzięki testom typu end-2-end, przybliżonych w podrozdziale 7.3. Proces przeprowadzania testów można zautomatyzować przy pomocy narzędzi opisanych w podrozdziale 7.4

Dobrą praktyką pozwalającą znacznie ograniczyć występowanie błędów w ostatecznej wersji systemu jest przygotowanie testów sprawdzających działanie poszczególnych funkcji. Istnieje kilka rodzajów testów, które można pogrupować tak jak na rysunku 7.1.



Rysunek 7.1. Rodzaje testów. Źródło: [18]

Dwie kategorie znajdujące się na dole - *unit testing* oraz *property testing* - mają pomóc deweloperom utworzyć działający kod. Tego rodzaju testy mają na celu sprawdzenie, czy system nie jest obciążony usterkami związanymi z implementacją. Celem testów należących do dwóch kategorii znajdujących się na górze - *acceptance testing* oraz *exploratory testing* - jest pomoc w zrozumieniu jak dany system działa. Do tego rodzaju testów można zaliczyć m. in. szerokie testy obejmujące działanie dużej liczby mikroserwisów, sprawdzenie funkcjonalności systemu oraz tzw. *user acceptance testing*, czyli testy przeprowadzone przez klienta, który zlecił budowę systemu.

7.1. Testy jednostkowe

Tego rodzaju testy sprawdzają poprawność pojedynczej metody w kodzie. Nie sprawdza się działania się całego mikroserwisu, a jedynie jego wyodrębioną część.

Wszystkie parametry, które przyjmuje dana funkcja, są tworzone w trakcie testu. Testy jednostkowe są wykonywane w pierwszej fazie testów ze względu na szybkość ich wykonania. Ich celem jest wykrycie błędów związanych z daną technologią, nie zaś sprawdzenie, czy działanie systemu jest zgodne z oczekiwaniami klienta. Należy zapewnić dużą liczbę testów jednostkowych, ponieważ jest to najszybszy sposób zlokalizowania potencjalnych awarii.

Przykładem jest poniższy test:

```
[Test]
public void MeasurementsTooHighIndicatorsTest()
{
    // Arrange
    var currentMeasurement =
    CreateMeasurementSentEvent();
    var policy = TestPoliciesDataService
        .CreateNewRoomPolicyDto(
            15, 80, 0.3f, 2, 20, 0.1f);

    // Run
    var result =
    _policyEvaluator!
        .Evaluate(currentMeasurement, policy);

    // Assert
    Assert.AreEqual(
        result.TemperatureStatus,
        EvaluatorResult.TooHigh);
    Assert.AreEqual(
        result.IlluminanceStatus,
        EvaluatorResult.TooHigh);
    Assert.AreEqual(
        result.HumidityStatus, EvaluatorResult.TooHigh);
}
```

Sprawdza on działanie logiki odpowiedzialnej za porównanie aktualnie panujących warunków w pomieszczeniu z warunkami oczekiwanymi. Test składa się z trzech części:

- Arrange - przygotowanie niezbędnych komponentów potrzebnych do przetestowania fragmentu kodu
- Run - faktyczne uruchomienie testowanego kodu
- Assert - sprawdzenie otrzymanego wyniku z oczekiwanym rezultatem

7.2. Testy integracyjne

Testy integracyjne mają na celu sprawdzenie, czy poszczególne mikroserwisy będą w stanie się ze sobą skutecznie komunikować. Przykładem jest poniższy test:

```
[Test]
public async Task
GetExpectedRoomConditionsAvailabilityTest()
{
    var path =
    $"policies-api/expected-room-conditions/1";

    var response = await _client.GetAsync(path);

    response.StatusCode.Should().Be(HttpStatusCode.OK);
}
```

Klient testowy korzysta z oferowanej przez testowany mikroserwis usługę poprzez wysłanie żądania http. W tym przypadku nie jest testowana logika zawarta w testowanym mikroserwisie, a jedynie odpowiedź, którą odsyła. Status odpowiedzi powinien oznaczać sukces, sygnalizowany przez kod 200 OK [19].

7.3. Testy end-2-end

Testy typu end-2-end mają za zadanie przetestować pewne biznesowe funkcjonalności, których realizacja może być obsługiwana przez wiele mikroserwisów. Dobrą praktyką jest utrzymywanie niewielkiej liczby tego typu testów, ponieważ obejmują one zakres działania całego systemu i przeprowadzenie każdego z nich zajmuje dużo czasu. Ponadto, w przypadku wystąpienia usterki ciężko jest wykryć miejsce, które było źródłem błędu. Przykładem jest poniższy test, napisany w języku programowania Python:

```
def measurement_triggers_policies_evaluation_result_event():
    new_measurement_test_helper = NewMeasurementTestHelper()

    new_measurement_test_helper
        .check_influxdb_is_available()

    new_measurement_test_helper
        .check_sensors_api_is_available()

    new_measurement_test_helper
        .send_new_measurement_to_rabbitmq()
```

```
new_measurement_test_helper
.check_message_sent_to_queue(
    new_measurement_test_helper
    .rabbitmq_configuration.vhost,
    consts.sensors_test_queue,
    1)

new_measurement_test_helper.
check_message_sent_to_queue(
    new_measurement_test_helper
    .rabbitmq_configuration.vhost,
    consts.measurement_sent_event_test_queue,
    1)

new_measurement_test_helper.
check_message_sent_to_queue(
    new_measurement_test_helper
    .rabbitmq_configuration.vhost,
    consts.policies_evaluation_result_event_test_queue,
    1)
```

W tym teście sprawdza się odpowiedź całego systemu na otrzymanie nowego pomiaru z sensora. Jako skutek powinna zostać wygenerowana odpowiednia wiadomość z wynikiem przetworzenia pomiaru, która została wysłana na kolejkę wiadomości.

7.4. Automatyzacja testów

Ręczne uruchamianie każdego z testów pojedynczo może szybko stać się żmudnym zajęciem. Aby temu zaradzić, w ramach pracy inżynierskiej wykorzystano narzędzie do automatyzacji o nazwie Nuke [20]. Za jego pomocą uruchomienie wszystkich testów sprowadza się do wykonania jednej komendy.

Nuke pozwala na tworzenie własnych metod zwanych zamiarami (ang. *target*), które automatyzują wykonywanie pewnych czynności. Każdy z zamiarów wykonuje określoną logikę. Dodatkowo, można tworzyć kompleksową strukturę, w której każdy z zamiarów jest zależny od tego, czy prawidłowo zostanie wykonany inny. Istnieje także możliwość dodania reguły wyzwalania poszczególnych zamiarów po wykonaniu innego. Przykładowo, metoda uruchamiająca test dla konkretnego projektu wygląda w sposób następujący:

```
Target TestProject => _ => _
.DependsOn(CompileTestProject)
```

```
.Executes(() =>
{
    var solution = (this as IHaveSolution).Solution;
    var project = solution
        .AllProjects
        .Single(x =>
        x.Name == TestProjectNames[ProjectName]);
    DotNet($"test {project} --no-build -c {Configuration}");
});
```

Wykonuje ona metodę *dotnet test*, uruchamianą dla konkretnego projektu. Wykonanie zamiaru zależy od pomyślnego wykonania innego, o nazwie *CompileTestProject*.

```
Target CompileTestProject => _ => _
.DependsOn(RestoreTestProject)
.Executes(() =>
{
    var solution = (this as IHaveSolution).Solution;
    var project = solution
        .AllProjects
        .Single(x =>
        x.Name == TestProjectNames[ProjectName]);
    DotNetBuild(s => s
        .EnsureNotNull(
            this as IHaveSolution, (_, o) =>
            s.SetProjectFile(project))
        .SetConfiguration(Configuration)
        .EnableNoRestore());
});
```

Metoda wykonuję komendę *dotnet build*. Jest ona zależna od zamiaru *RestoreTestProject*.

```
Target RestoreTestProject => _ => _
.DependsOn(Clean)
.Requires(() => ProjectName)
.Executes(() =>
{
    var solution = (this as IHaveSolution).Solution;
    foreach (var proj in solution.AllProjects)
    {
        Logger.Info(proj.Name);
```

```
    }
    var project = solution
        .AllProjects
        .Single(x =>
        x.Name == TestProjectNames[ProjectName]);
    DotNetRestore(s =>
    s.EnsureNotNull(
        this as IHaveSolution, (_, o) =>
        s.SetProjectFile(project)));
});
```

Metoda wykonuje funkcję *dotnet restore*. Jednym z warunków uruchomienia tego zamiaru jest konieczność podania nazwy projektu, wyrażona przez funkcję *.Requires(() => ProjectName)*. Uruchomienie zamiaru jest zależne od poprawnego wykonania innego, o nazwie *Clean*.

```
Target Clean => _ => _
.Executes(() =>
{
    SourceDirectory
    .GlobDirectories("**/bin", "**/obj")
    .ForEach(DeleteDirectory);
    EnsureCleanDirectory(ArtifactsDirectory);
});
```

Metoda czyści repozytorium z artefaktów. Nie jest zależna od żadnego innego zamiaru. Wszystkie zamiary można uruchomić jednocześnie przy pomocy jednej metody:

```
./build.sh TestProject --ProjectName facilities
    --verbosity verbose
```

8. Pomiary

Treść sekcji zawiera opis czujników oraz innych elementów elektronicznych wykorzystanych do zbudowania zestawu wykonującego pomiary temperatury oraz natężenia światła. Podrozdział 8.1 przedstawia szczegóły implementacyjne dotyczące utworzonego oprogramowania sterującego wykonywaniem pomiarów.

W ramach niniejszej pracy przygotowano zestaw pomiarowy, który w regularnych odstępach czasu bada aktualną wartość temperatury oraz natężenia światła. Zestaw został złożony z następujących elementów:

- Moduł WEMOS D1 Uno R3 ESP8266 WIFI
- Fotorezystor LDR GL5528
- Czujnik temperatury i wilgotności DHT11
- Rezystory 1 kOhm
- przewody połączeniowe żeńsko-męskie

Moduł WEMOS jest mikrokontrolerem opartym o kontroler ATmega328P. Posiada 11 portów wejścia-wyjścia pozwalających na dołączenie zewnętrznych urządzeń. Płytka została fabrycznie wyposażona w moduł WIFI ESP8266. Za jego pomocą mogą być wysyłane pomiary.

Fotorezystor służy do pomiaru natężenia światła. Został wykonany z półprzewodników, które w temperaturze działania nie mają elektronów w paśmie przewodnictwa. Padające na półprzewodnik fotony o energii większej od przerwy energetycznej przemieszczają elektrony z pasma walencyjnego do pasma przewodnictwa, w wyniku którego powstają pary dziura-elektron. Zjawisko nazywane jest efektem fotoelektrycznym wewnętrznym.

Czujnik temperatury DHT11 bada aktualne wartości tego parametru. Mierzony zakres temperatury to -20°- +60°C. Jego rozdzielczość wynosi 0.1°C, a dokładność 2°C.

8.1. Oprogramowanie mikrokontrolera

Do rozwoju oprogramowania do mikrokontrolera WEMOS wykorzystano narzędzie Arduino IDE, będące rozbudowanym edytorem pozwalającym na kompilację, przesyłanie plików wykonywalnych na płytkę przy pomocy kabla USB oraz debugowanie i podgląd logów produkowanych przez mikrokontroler w czasie rzeczywistym. Do utworzenia kodu został wykorzystany język C.

Zadaniem mikrokontrolera było:

- Zebranie aktualnych pomiarów temperatury i natężenia światła
- przygotowanie wiadomości z uzyskanymi pomiarami
- wysłanie wiadomości przy pomocy protokołu MQTT przez moduł WIFI na kolejkę brokera wiadomości RabbitMQ

Poniżej został przedstawiony przygotowany kod.

```
void setup() {
  Serial.begin(115200);
  dht.begin();

  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASS);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(100);
  }

  client.setServer(RABBITMQ_BROKER, RABBITMQ_PORT);
  client.setCallback(callback);
}

void loop() {
  if ( !client.connected() ) {
    reconnect();
  }

  float temperature = readTemperature(&dht);
  int illuminance = analogRead(ILLUMINANCEPIN);
  publish_measurements(temperature, illuminance, false);

  delay(60000);

  client.loop();
}
```

Każdy z programów wgrywanych na płytkę powinien zawierać dwie główne funkcje:

- *void setup()* - wewnątrz metody definiowane są obiekty oraz zmienne potrzebne przez cały czas działania mikrokontrolera
- *void loop()* - metoda wywoływana jako druga w kolejności po *setup()*, jest powtarzana przez resztę cyklu działania mikrokontrolera

Na początku definiowana jest szyna, za pomocą której przesyłane są dane do modułu WIFI. Moduł wspiera prędkość transmisji na poziomie 115200 bitów na sekundę (ang. *baud rate*). Następnie tworzony jest klient, którego zadaniem jest wysyłanie pomiarów na kolejkę brokera wiadomości.

Wewnątrz metody *loop()* co 60 sekund zbierane są pomiary, po czym zostają opublikowane na kolejkę. Pomiary wykonywane są przy użyciu poniższego kodu.

```
#include "DHT.h"

#define DHTPIN 4
#define DHTTYPE DHT11

int ILLUMINANCEPIN = A0;

DHT dht(DHTPIN, DHTTYPE);

float readTemperature(DHT *dht){
  float t = dht->readTemperature();

  if (isnan(t))
  {
    Serial.println(
        "Error while reading current
        temperature value");
  }

  return t;
}
```

Główną rolę pełni obiekt dht, za pomocą którego można komunikować się z czujnikiem. Typ czujnika został zdefiniowany przy użyciu makra *DHTTYPE*. Numer pinu, do którego został wpięty kabel łączący płytkę z czujnikiem, został oznaczony pzy użyciu makra *DHTPIN*. Pomiar temperatury odbywa się przez wywołanie funkcji *readTemperature()*.

Pomiar z fotorezystora można odczytać przy użyciu komendy:

```
analogRead(ILLUMINANCEPIN)
```

gdzie *ILLUMINANCEPIN* oznacza numer pinu, do którego wpięty jest kabel łączący płytkę z fotorezystorem.

Warto wyjaśnić wartości zmiennych związanych z brokerem wiadomości:

```
const char* RABBITMQ_BROKER = "192.168.0.12";
int         RABBITMQ_PORT   = 1883;
const char* RABBITMQ_TOPIC  = "room_measurements";
const char* RABBITMQ_SUBSCRIPTION
= "request_measurement";
```

```
const char* RABBITMQ_USER = "guest";
const char* RABBITMQ_PASSWORD = "guest";
const char* RABBITMQ_SENSOR_ID = "968376";
```

- RABBITMQ_BROKER: adres IP serwera, na którym uruchomiony jest broker wiadomości
- RABBITMQ_PORT: numer portu serwera, na którym nasłuchuje broker
- RABBITMQ_TOPIC: temat, na który wysyłane są wiadomości z mikrokontrolera na kolejkę
- RABBITMQ_SUBSCRIPTION: temat, na który nasłuchuje mikrokontroler
- RABBITMQ_USER: nazwa użytkownika, za pomocą którego płytka jest uwierzytelniana
- RABBITMQ_PASSWORD: hasło dla wykorzystywanego użytkownika

Wiadomość do brokera jest wysyłana w metodzie *send_measurements()*:

```
void send_measurements(
    float temperature, int illuminance){
  char temperatureChar[64];
  int ret = snprintf(
      temperatureChar, sizeof temperatureChar,
      "%f", temperature);
  if (ret < 0) {
      return;
  }
  if (ret >= sizeof temperatureChar) {
       return;
  }
  char illuminanceChar[64];
  ret = snprintf(
      illuminanceChar, sizeof illuminanceChar,
      "%d", illuminance);
  if (ret < 0) {
      return;
  }
  if (ret >= sizeof illuminanceChar) {
       return;
  }

  char* measurement = (char*)malloc(256+1+3);
  //4*64 + 3 semicolons + EOF
  strcpy(measurement, temperatureChar);
```

```
    strcat(measurement, ";");
    strcat(measurement, illuminanceChar);
    strcat(measurement, ";");
    strcat(measurement, RABBITMQ_SENSOR_ID);

    time_t t = time(NULL);
    struct tm tm = *localtime(&t);

    client.publish(RABBITMQ_TOPIC, measurement);

    free(measurement);
  }
```

Mikrokontroler nasłuchuje na wiadomości o temacie *RABBITMQ_SUBSCRIPTION*:

```
  void callback(
      char* topic, byte* payload, unsigned int length) {
    char* sensorId = (char*)payload;

    String messageTemp;

    for (int i = 0; i < length; i++) {
      Serial.print((char)payload[i]);
      messageTemp += (char)payload[i];
    }

    if(strcmp(topic, RABBITMQ_SUBSCRIPTION)
    == 0 && strcmp(sensorId, RABBITMQ_SENSOR_ID)){
      float temperature = readTemperature(&dht);
      int illuminance = analogRead(ILLUMINANCEPIN);
      publish_measurements(
          temperature, humidity, illuminance, true);
    }
  }
```

Serwis SSDS może zażądać wysłania aktualnych pomiarów poprzez wysłanie do brokera wiadomości o temacie *RABBITMQ_SUBSCRIPTION*.

9. Automatyzacja

Sekcja przedstawia narzędzia wykorzystane do automatyzacji procesu rozwoju i wdrażania systemu. Podrozdział 9.1 opisuje zasady działania platformy Docker oraz sposób jej wykorzystania w pracy. Podrozdziały 9.2 - 9.3 przybliżają metodyki ciągłej integracji oraz ciągłego dostarczania, bezpośrednio powiązanymi z automatyzacją rozwiązania. W podrozdziale 9.4 opisano zasady działania oraz sposób wykorzystania narzędzia do orkiestracji systemu Kubernetes.

Pełna automatyzacja regularnie wykonywanych zadań, pozwalająca znacznie przyspieszyć wdrażanie całości systemu, była jednym z najważniejszych zagadnień poruszonych w trakcie tworzenia pracy. Prawidłowe podejście do wdrażania aplikacji znacząco wpływa na szybkość, z jaką zmiany wprowadzone lokalnie mogą zostać wykorzystane w produkcyjnej wersji systemu.

9.1. Docker

Sekcja zawiera opis zasad działania oraz sposób wykorzystania platformy Docker. Podrozdział 9.1.1 przybliża sposób komunikacji kontenerów ze sobą oraz ze światem zewnętrznym.

Docker jest otwartą platformą do rozwoju oraz wdrażania aplikacji. Pozwala oddzielić aplikacje od dostępnej na danym serwerze infrastruktury, co pozwala przyspieszyć proces dostarczania najnowszych wersji systemu.

Platforma zapewnia możliwość spakowania i uruchomienia aplikacji w odizolowanym środowisku zwanym kontenerem. Izolacja powoduje, że możliwe jest bezpieczne uruchomienie wielu kontenerów na tym samym hoście. Platforma pozwala zarządzać infrastrukturą potrzebną do uruchomienia kontenera, dzięki czemu eliminowany jest problem instalacji wszystkich wymaganych zależności na każdym serwerze z osobna.

Docker wykorzystuje architekturę typu klient-serwer. Klient komunikuje się z tzw. *docker daemon*, którego zadaniem jest budowanie, uruchamianie oraz dystrybuowanie kontenerów.

W celu utworzenia kontenerów należy w pierwszej kolejności utworzyć ich obraz, który jest szablonem zawierającym instrukcje dotyczące jego budowy. Szablony są przechowywane w plikach o nazwie *Dockerfile*. Każda instrukcja tworzy jedną warstwę obrazu. Mechanizm ten jest szczególnie przydatny, gdy szablony są regularnie rozwijane o nowe instrukcje. Wtedy, przy ponownym budowaniu obrazu, tylko warstwy, które uległy zmianie są odświeżane. Pozwala to znacznie przyspieszyć proces budowania obrazów.

Kontener jest instancją obrazu, którą można uruchomić. Działa on tak długo, dopóki nie zostanie zakończony proces główny.

Przykładowy szablon, który przedstawia obraz ADS, został przedstawiony poniżej:

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /app
COPY [".", "."]

RUN dotnet restore
    ./DagAir_Addresses/DagAir.Addresses
    /DagAir.Addresses.csproj
RUN dotnet build
    ./DagAir_Addresses/DagAir.Addresses
    /DagAir.Addresses.csproj
    --no-restore
RUN dotnet publish
    ./DagAir_Addresses/DagAir.Addresses
    /DagAir.Addresses.csproj
    -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "DagAir.Addresses.dll"]
```

Pierwszym etapem jest utworzenie obrazu pośredniego o nazwie base. Oparty on jest na obrazie aspnet:5.0 dostępnym publicznie na platformie dockerhub [21]. Na tym etapie deklarowane są numery portów, na których aplikacja powinna nasłuchiwać na przychodzące żądania.

Drugim etapem jest utworzenie obrazu pośredniego, opartego o obraz sdk:5.0, który także jest dostępny publicznie na platformie dockerhub [22]. Na tym etapie kopiowane są wszystkie zależności, których wymaga aplikacja, by mogła zostać prawidłowo uruchomiona. Za pomocą komend oferowanych przez dotnet CLI publikowana jest gotowa do wdrożenia wersja aplikacji.

W ostatnim etapie kopiowana jest wersja aplikacji przygotowana w ramach etapu drugiego, po czym następuje uruchomienie procesu za pomocą komendy:

```
dotnet DagAir.Addresses.dll
```

Podział na poszczególne etapy wynika z różnych rozmiarów wykorzystywanych obrazów. Możliwe byłoby utworzenie obrazu aplikacji na podstawie obrazu sdk:5.0. Jednak jego rozmiar to ok. 630 MB, podczas gdy rozmiar obrazu aspnet:5.0 to ok. 205 MB. Dzięki temu prostemu zabiegowi można oszczędzić znaczną część zasobów obliczeniowych.

Dzięki szablonom można tworzyć obrazy poszczególnych serwisów. Jednak uruchamianie każdego z nich pojedynczo byłoby kosztownym czasowo zajęciem. Rozwiązaniem tego problemu jest wykorzystanie narzędzia *docker-compose*, które umożliwia uruchamianie wielu kontenerów za pomocą jednej komendy. W tym celu należy przygotować szablon zawierający instrukcje uruchomienia każdego z utworzonych na wcześniejszym etapie obrazów. Szablon jest przechowywany w pliku o nazwie *docker-compose.yml*.

Przykładowy szablon został przedstawiony poniżej:

```
version: "3.9"
services:
  addresses_api:
    build:
      target: final
      context: ./src
      dockerfile: ./DagAir_Addresses/Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Docker
      - ConnectionKeys__DagAir.Addresses=
            ${ADDRESSES_CONNECTIONKEYS}
    ports:
      - "8094:80"
    networks:
      - dagair_network
    restart: always

networks:
dagair_network:
```

W szablonie został przedstawiony proces uruchomienia ADS. Zdefiniowano:

- Szablon obrazu, z którego należy skorzystać
- Zmienne środowiskowe potrzebne do uruchomienia aplikacji w środowisku dockerowym
- Port, na którym ma nasłuchiwać aplikacja. Występuje tutaj mapowanie między portem serwera, na którym będzie uruchomiony kontener, a portem w kontenerze. Dzięki mapowaniu aplikacja będzie dostępna na serwerze pod adresem:

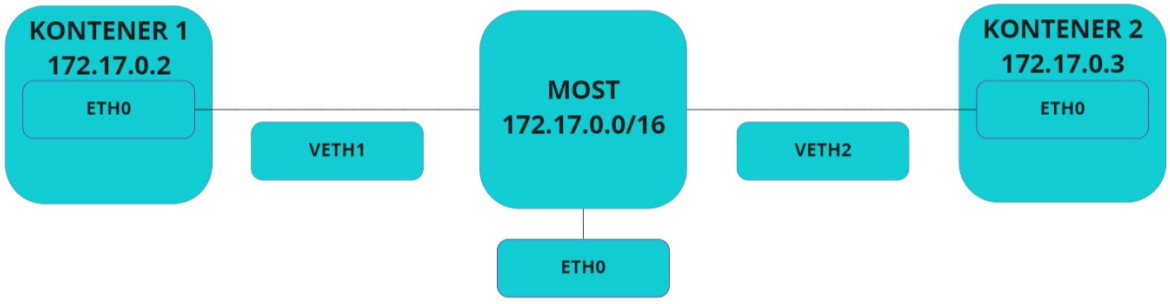
```
http://localhost:8094/
```

- Wirtualna sieć, do którego ma zostać dołączony kontener w środowisku dockerowym
- Polityka ponownego uruchamiania. Flaga *always* oznacza, że w przypadku zatrzymania pracy instancji kontenera, zostanie ona usunięta, a na jej miejsce zostanie uruchomiona nowa

9.1.1. Łączność sieciowa w środowisku skonteneryzowanym

W celu zapewnienia izolacji między procesami różnych kontenerów wykorzystuje się przestrzenie nazw (ang. *namespace*). Jest to funkcjonalność polegająca na podziale zasobów w taki sposób, aby każdy zestaw procesów korzystał z innej, wydzielonej części. Przestrzeni nazw przypisuje się dedykowany interfejs, za pomocą którego może komunikować się z innymi przestrzeniami. Łączność między kontenerami jest osiągana poprzez połączenie utworzonych interfejsów za pomocą mostu (ang. *bridge*). Pełni on rolę tzw. *switch'a* odpowiedzialnego za przesyłanie pakietów między przestrzeniami nazw w oparciu o protokół ARP lub IP. Do mostu dołączony jest także interfejs pozwalający na komunikację z procesami uruchomionymi na oddzielnych maszynach.

Rysunek 9.1 przedstawia podział zasobów na przestrzenie nazw.



Rysunek 9.1. Łączność sieciowa w środowisku skonteneryzowanym. Opracowanie własne

Platforma docker domyślnie tworzy most należący do podsieci 172.17.0.0/16.

Każdy kontener może uruchamiać procesy, które będą nasłuchiwać żądań na ustalonych portach. Przestrzenie nazw posiadają oddzielne zestawy portów, dzięki czemu w każdej z nich może zostać uruchomiony proces nasłuchujący przykładowo na porcie 80. Mapowanie portów w środowisku kontenerów polega na skojarzeniu portów udostępnianych przez maszynę, na której uruchomione sobą kontenery, z portami udostępnianymi przez przestrzenie nazw. Poniższy fragment przedstawia mapowanie między portem 80 udostępnianym przez przestrzeń nazw, w której uruchomiono mikrousługę ADS, a portem 8094 udostępnianym przez hosta:

```
ports:
    - "8094:80"
```

Dzięki temu usługi oferowane przez ten moduł danych są dostępne pod adresem *http://localhost:8094*.

9.2. Ciągła integracja

Podstawowym wymaganiem, które należy spełnić przy tworzeniu rozbudowanych systemów informatycznych, jest przechowywanie rozwijanego oprogramowania przy pomocy wybranego narzędzia kontroli wersji, takiego jak Git [23]. Dane są zapisywane w folderze zwanym repozytorium. Zadaniem takiego narzędzia jest śledzenie wprowadzonych zmian oprogramowania i zapisywanie ich w historii repozytorium. Zapewnia to wiele korzyści, z których najważniejsze to:

- Podgląd zmian wprowadzonych przez każdego dewelopera
- Możliwość powrotu do poprzedniej wersji w przypadku, gdy wprowadzone zmiany były przyczyną błędów w działaniu systemu

Głównym celem ciągłej integracji (ang. *continuous integration*) jest regularne włączanie bieżących zmian w kodzie do głównego repozytorium i każdorazowa weryfikacja wprowadzonych zmian poprzez utworzenie nowego zbioru plików wykonywalnych i przeprowadzenie na nich testów jednostkowych. Zaletą tego podejścia jest fakt, że po wysłaniu przez programistę zmian do repozytorium głównego, reszta czynności wykonywana jest automatycznie przez serwer ciągłej integracji, bez ingerencji człowieka. Dodatkowo programista otrzymuje szybką odpowiedź zwrotną w razie wystąpienia błędów. Aby wykorzystać potencjał ciągłej integracji, należy zwrócić uwagę na następujące punkty:

- Częste i regularne wysyłanie kodu do głównego repozytorium w celu weryfikacji integracji nowych zmian z resztą kodu, przynajmniej raz dziennie
- Zapewnienie testów jednostkowych sprawdzających poprawność zachowania systemu. Może się zdarzyć, że wprowadzone zmiany będą zgodne pod względem składni, jednak nie oznacza to, że serwis będzie prawidłowo spełniał swoje funkcje
- Nadanie wysokiego priorytetu naprawieniu kodu, który nie integruje się z dotychczasowym kodem w repozytorium. Odkładanie poprawy na później może spowodować spiętrzenie się kolejnych błędów, co w konsekwencji bardziej spowolni wdrażanie nowych funkcji

W trakcie tworzenia pracy wykorzystano platformę do ciągłej integracji i wdrażania o nazwie Github Actions. Pozwala ona na automatyzację tworzenia nowych wersji oprogramowania, testowania oraz wdrażania. Kolejne powtórzenia przepływów pracy (ang. *workflow*) są wykonywane na maszynach wirtualnych oferowanych przez

GitHub, zwanych pracownikami (ang. *worker*). W zależności od potrzeby na maszynach zainstalowany jest odpowiedni system operacyjny spośród sytrybucji linux-owych, Windowsa oraz macOS.

GitHub Action jest przepływem pracy, który może zostać wywołany zawsze wtedy, gdy zostanie zarejestrowane nowe zdarzenie dotyczące wykorzystywanego repozytorium. Przykładem zdarzenia jest wprowadzenie nowych zmian do repozytorium lub utworzenie żądania typu *pull request*. *GitHub Action* składa się z jednej lub większej liczby zadań (ang. *job*), które mogą zostać wykonane jedno po drugim lub równolegle. Z kolei każde zadanie składa się z jednego lub większej liczby kroków (ang. *step*), z których każde może wykonać własnoręcznie utworzony skrypt lub akcję (ang. *action*), będącą rozszerzeniem umożliwiającym uproszczenie całego przepływu pracy.

Przykładowy przepływ pracy utworzony na potrzeby projektu wykonuje następujące zadania:

- Wybiera odpowiednią gałąź z repozytorium, na której zostały wprowadzone nowe zmiany
- Instaluje wymagane oprogramowanie niezbędne do wykonania wszystkich pozostałych zadań, takie jak środowisko .NET 5.0
- Uruchamia testy jednostkowe i integracyjne
- Tworzy nową wersję obrazu sprawdzonego mikroserwisu
- Wypycha obraz do rejestru kontenerów

Warto szczegółowo prześledzić poszczególne kroki danego przepływu.

```
jobs:
  docker-build-and-push:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v2
      with:
        fetch-depth: 0
```

Powyższy wyciąg deklaruje nowe zadanie oraz pierwszy z kroków, który wybierze odpowiednią gałąź z repozytorium. Parametr runs-on wskazuje jaki rodzaj systemu operacyjnego powinien zostać wykorzystany.

```
on:
  workflow_dispatch: # allows to trigger workflow on demand
  push:
    branches:
      - main
      - develop
```

```
    paths:
      - src/DagAir_Facilities/**
```

GitHub Actions oferuje rozbudowany system do określania warunków, które muszą być spełnione, aby uruchomić przepływ pracy. Powyższy wyciąg przedstawia fragment, który określa, że przepływ ma być uruchomiony gdy:

- Użytkownik manualnie uruchomi przepływ za pomocą interfejsu graficznego
- Zostaną wprowadzone nowe zmiany na gałęzi main lub develop oraz zmiany będą się znajdować w katalogu *src/DagAir_Facilities/*

```
- name: Build & Test
    shell: bash
    run: ./build.sh TestProject
        --ProjectName facilities --verbosity verbose
```

Powyższy krok wykonuje skrypt uruchamiający testy jednostkowe oraz integracyjne.

```
- name: Set image names & main tags
run: |
  appImageName="${{ env.
    CONTAINER_REGISTRY }}/${{ env.SERVICE_NAME }}"
  migrationsApplierImageName=
    "${{ env.CONTAINER_REGISTRY }}/${{ env.
    MIGRATIONS_APPLIER_NAME }}"
  echo "APP_IMAGE_NAME=$appImageName" >> $GITHUB_ENV
  echo "MIGRATIONS_APPLIER_IMAGE_NAME
    =$migrationsApplierImageName"
    >> $GITHUB_ENV

  version="${{ steps.gitversion.outputs
    .nugetVersionV2 }}-${{ steps.gitversion
    .outputs.shortSha }}"

  if [ ${{ steps.gitversion.outputs
    .commitsSinceVersionSource }} -gt 0 ]; then
  version="${{ steps.gitversion.outputs
    .escapedBranchName }}-$version"
  fi

  echo "APP_IMAGE_TAG=${appImageName}:$version"
    >> $GITHUB_ENV
```

```
echo "MIGRATIONS_APPLIER_IMAGE_TAG=
${migrationsApplierImageName}:$version" \
>> $GITHUB_ENV
```

Powyższy krok generuje nazwę oraz tag nowego obrazu testowanego mikroserwisu. Na nazwę obrazu składa się nazwa repozytorium obrazów oraz nazwa mikroserwisu. Numer wersji obrazu za każdym razem powinien być unikalny, ponadto powinien wskazywać, która z wersji obrazu jest najnowsza. Wobec tego na wersję składa się nazwa gałęzi repozytorium, nazwa utworzonej paczki Nuget-owej oraz krótki unikalny numer przypisany do migawki (ang. *commit*), która spowodowała uruchomienie przepływu.

```
- name: Docker login
  uses: docker/login-action@v1
  with:
    registry: ${{ env.CONTAINER_REGISTRY }}
    username: ${{ secrets.AZURE_CR_USERNAME }}
    password: ${{ secrets.AZURE_CR_PASSWORD }}

- name: Docker push images
  run: |
    docker push ${{ env.APP_IMAGE_NAME }} --all-tags
    docker push ${{ env.MIGRATIONS_APPLIER_IMAGE_NAME }}
        --all-tags
```

Na samym końcu gotowy obraz zostaje wypchnięty do repozytorium obrazów. W tym celu używana jest komenda *docker push*. Aby zakończyła się pomyślnie, należało w poprzednim kroku zalogować się do repozytorium, wykorzystując nazwę repozytorium obrazów oraz danych uwierzytelniających, przechowywanych w bezpieczny sposób za pomocą Github Secrets.

9.3. Ciągłe dostarczanie

W trakcie prac nad systemem wykorzystano metodykę ciągłego dostarczania, która polega na tym, że kod przechodzi przez kolejne fazy testowania, gdzie za każdym razem jest weryfikowana jego poprawność pod względem prawidłowego funkcjonowania.

Na początku przeprowadzane są szybkie testy jednostkowe sprawdzające punktowo poprawność poszczególnych funkcji. Jeśli zostaną wykonane pomyślnie, przechodzi się do następnej fazy testowania uwzględniające wolniejsze testy, które sprawdzają zachowanie wielu serwisów między sobą. Po upewnieniu się, że ta faza została wykonana pomyślnie, system jest weryfikowany przez klienta, który zlecał jego wykonanie (tzw. *user acceptance testing*). Jeśli funkcjonalność zgadza się z oczekiwaniami klienta, ca-

łość jest jeszcze testowana pod kątem wydajności, po czym zostaje wdrożona na etap produkcyjny.

Zgodnie z założeniami metodyki, programiści powinni otrzymywać wiadomości zwrotne dotyczące statusu kolejnych wersji oprogramowania w kolejnych fazach testowania. Wprowadzenie takiego potoku potrafi znacznie poprawić ocenę jakości kodu, ponadto skrócić czas między wdrożeniem kolejnych wersji systemu.

9.4. Kubernetes

Sekcja opisuje zasady działania oraz sposób wykorzystania narzędzia Kubernetes. Podrozdziały 9.4.1 - 9.4.2 przedstawiają poszczególne komponenty, z których składa się narzędzie. Podrozdział 9.4.3 przybliża sposób komunikacji kontenerów znajdujących się w klastrze ze sobą oraz ze światem zewnętrznym.

Kubernetes jest platformą do orkiestracji kontenerów automatyzującą procesy manualne związane z wdrażaniem, zarządzaniem oraz skalowaniem skonteneryzowanych aplikacji. Jest to oprogramowanie typu open-source, początkowo rozwijane przez firmę Google [24].

W przeszłości, organizacje uruchamiały aplikacje na fizycznych serwerach. W momencie gdy wiele aplikacji działało w ramach jednego serwera, dochodziło do sytuacji, w których jedna z nich zajmowała większość zasobów, przez co inne nie działały optymalnie. Jednym z rozwiązań było uruchomienie każdej z aplikacji na innym fizycznym serwerze. Jednak w takim wypadku konsekwencjami były wysokie koszty utrzymania infrastruktury. Innym możliwym rozwiązaniem było wprowadzenie wirtualizacji, które przyczyniło się do bardziej zrównoważonego zarządzania zasobami. Efektem ubocznym było jednak wprowadzanie dużego nakładu zasobów potrzebnych na uruchomienie samej maszyny wirtualnej, ponieważ każda z maszyn instalowała na początku własny system operacyjny.

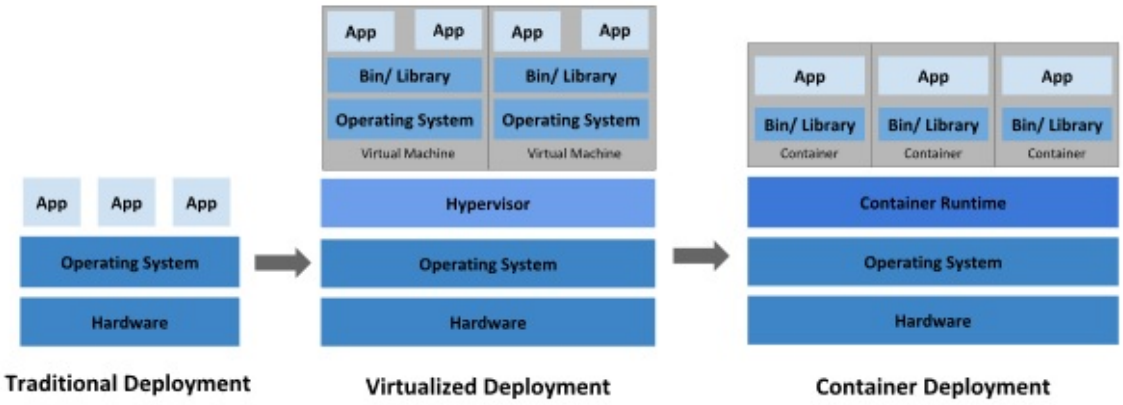
Najlepszym obecnie rozwiązaniem jest wykorzystanie kontenerów. Zawierają zestaw podobnych cech do maszyn wirtualnych z tą różnicą, że nie wymagają osobnego systemu operacyjnego. Każdy z kontenerów może współdzielić jeden system operacyjny z innymi, co znacznie obniża wymagania dotyczące zasobów. Podobnie do maszyn wirtualnych posiadają własny system plików, zasoby obliczeniowe, pamięć. Jednak nie zależą od infrastruktury, na której są uruchamiane, co czyni je przenośnymi wśród różnych dystrybucji danego systemu.

Kontenery stały się popularne ze względu na szereg zalet:

- Utworzenie obrazów następuje szybciej w porównaniu do maszyn wirtualnych
- Obrazy zostają utworzone na etapie budowania nowej wersji systemu zamiast na etapie wdrażania
- Niezależnie od środowiska działają w dokładnie ten sam sposób
- Mogą być uruchomione praktycznie na każdym systemie i dystrybucji

- Cechują się wysoką efektywnością wykorzystania zasobów

Rysunek 9.2. przedstawia różnice między uruchomieniem aplikacji w sposób tradycyjny, przy użyciu maszyn wirtualnych oraz kontenerów.



Rysunek 9.2. Porównanie różnych metod uruchamiania aplikacji. Źródło: [24]

Zalety Kubernetesa to przede wszystkim:

- Orkiestracja kontenerów między różnymi serwerami
- Efektywniejsze wykorzystanie zasobów
- Łatwe skalowanie skonteneryzowanych aplikacji
- Zarządzanie serwisami w sposób deklaratywny
- Kontrola stanu aplikacji, automatyczne restartowanie kontenerów, autoskalowanie

Zbiór hostów wykorzystywanych do uruchomienia na nich systemu zwany jest klastrem. Na każdym z hostów, zwanych węzłami, można uruchomić instancje gotowych obrazów. Każdy z klastrów posiada przynajmniej jeden węzeł.

Cyklem życiowym każdego kontenera zarządza płaszczyzna sterowania (ang. *control plane*), która wystawia API oraz interfejsy umożliwiające ich wdrażanie i zarządzanie. Komponenty płaszczyzny mogą być uruchomione na każdej maszynie w klastrze, chociaż zazwyczaj określa się jedną maszynę, tzw. gospodarza (ang. *master*), na której znajdują się wszystkie komponenty.

9.4.1. Komponenty płaszczyzny sterowania

W tabeli 9.1 zostały opisane komponenty składające się na całość płaszczyzny sterowania.

Tabela 9.1. Komponenty płaszczyzny sterowania

Nazwa komoponentu	Opis
Kube-apiserver	Interfejs pozwalający na interakcję z płaszczyzną sterowania. Weryfikuje i konfiguruje dane dla obiektów takich jak serwisy czy kontrolery replikacji
Etcd	Spójny i wysoce dostępny magazyn par klucz-wartość używany przez Kubernetesa jako miejsce do przechowywania wszystkich danych ważnych z punktu widzenia klastra.
Kube-scheduler	Regularnie sprawdza czy został utworzony nowy zestaw kontenerów, któremu nie został jeszcze przypisany węzeł. W takim przypadku wybiera on maszynę, na której kontenery mają być uruchomione. Przy wyborze pod uwagę brane są takie czynniki jak wymagane zasoby, ograniczenia sprzętowe lub programowe
Kube-controller-manager	Komponent odpowiedzialny za uruchamianie kontrolerów, które monitorują oraz zmieniają stan klastra korzystając z API serwera
Cloud-controller-manager	Element, który wbudowuje logikę związaną z konkretną chmurą, w której tworzone są klastry. Pozwala połączyć dany klaster z API dostawcy chmury oraz oddziela komponenty, które oddziałują z chmurą od komponentów, które oddziałują tylko z klastrem. Jest to komponent, który występuje tylko w przypadku stawiania kontenerów w chmurze. Jeśli Kubernetes działa np. w prywatnym środowisku na jednym komputerze, wtedy klaster nie posiada tego elementu

Istnieje kilka rodzajów kontrolerów:

- Kontroler węzłów (ang. *node controller*) - odpowiedzialny za wykrycie oraz odpowiednią reakcję w przypadku, gdy jeden z węzłów ulega awarii lub staje się niedostępny
- Kontroler prac (ang. *job controller*) - nasłuchuje na pojawienie się obiektów pracy (job objects) reprezentujących zadania, a następnie tworzy zbiór (pod) który te zadania wykona
- Kontroler punktów końcowych - zarządza obiektami punktów końcowych (serwisy oraz zbiory (ang. *pods*))

- Kontroler kont oraz tokenów - tworzy domyślne konta oraz tokeny dostępu do API dla nowych przestrzeni nazw

9.4.2. Komponenty węzła

W tabeli 9.2 opisano komponenty, które działają na każdym węźle w Kubernetesie.

Tabela 9.2. Komponenty węzła

Nazwa komponentu	Opis
Kubelet	Agent, którego rolą jest upewnienie się, że kontenery są uruchomione w zbiorze (pods). Przyjmuje zestaw specyfikacji zbiorów i zapewnia, że wszystkie kontenery podane w specyfikacji działają i są sprawne. Kubelet nie zarządza kontenerami, które nie zostały utworzone przez Kubernetesa.
Kube-proxy	Proxy sieciowe, które implementuje część serwisu pozwalającego wystawić aplikację do świata zewnętrznego. Jego zadaniem jest utrzymanie reguł sieciowych w zarządzanych węzłach. Te reguły pozwalają na komunikację między różnymi zbiorami wewnątrz lub na zewnątrz klastra.
Container runtime	Oprogramowanie odpowiedzialne za uruchamianie kontenerów. Kubernetes wspiera wiele możliwych runtime-ów, m. in. Docker, containerd, CRI-O.
Pod	Grupa złożona z jednego lub większej liczby kontenerów, wdrożona na tym samym węźle. Wszystkie kontenery z grupy współdzielą adres IP oraz przydzielone zasoby.
Replication controller	Narzędzie do kontroli liczby kopii danego poda, które powinny być w danej chwili uruchomione

Płaszczyzna sterowania przyjmuje komendy od administratora klastra, po czym przekazuje je do podległych serwerów. Komendy przyjmowane są za pomocą interfejsu konsolowego, zwanego *kubectl*. Dobrą praktyką jest utworzenie plików deklarujących pożądany stan, w jakim powinien znajdować się klaster. Przykładowa deklaracja znajduje się poniżej.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-admin-app
  labels:
```

```
    app: web-admin-app
    tier: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: admin-application-service
  template:
    metadata:
      labels:
        app: admin-application-service
        tier: backend
    spec:
      containers:
      - name: admin-application-service
        image: admin-application-service:develop-latest
        env:
        - name: ASPNETCORE_ENVIRONMENT
          value: "Kubernetes"
        imagePullPolicy: Always
        ports:
        - containerPort: 80
```

Jest to deklaracja typu *Deployment*, która powinna zawierać następujące parametry:

- Wersja wykorzystywanego API
- Typ deklaracji
- Nazwa deklaracji
- Specyfikacja przedstawiająca pożądany stan, w jakim powinien znajdować się klaster. W tym przypadku deklaruje się, że w klastrze powinny działać dwie instancje obrazu mikrousługi aplikacyjnej dla administratorów, które powinny nasłuchiwać na żądania na porcie 80

Komenda *kubectl apply -f deployment.yml* umożliwia zaaplikowanie deklaracji.

Płaszczyzna sterowania jest odpowiedzialna za to, by stan klastra odpowiadał deklaracji. W konsekwencji zostaną utworzone dwa osobne pody, z których każdy otrzyma unikalny prywatny adres IP wewnątrz klastra. Od tej pory do każdej instancji można się odwołać, wykorzystując jej adres IP oraz numer portu.

Należy wziąć pod uwagę, że pody nie są trwałymi zasobami. Mogą być tworzone i usuwane w sposób dynamiczny. Za każdym razem pod otrzymuje nowy adres IP, który może się różnić od poprzednich. Prowadzi to do problemów przy komunikacji między

mikroserwisami, ponieważ nie wiedzą, że wymagany mikroserwis nie jest już osiągalny pod dotychczasowym adresem.

Rozwiązaniem tego zagadnienia jest wprowadzenie tzw. serwisu (ang. *service*). Jest to abstrakcyjny obiekt, który definiuje zbiór pod-ów oraz reguły umożliwiające do nich dostęp. Serwisowi nadawany jest unikalny adres IP, pod który mogą odwoływać się mikroserwisy. W dalszym ciągu pod-y będą dynamicznie tworzone i usuwane, jednak w tym wypadku będą one ciągle dostępne pod adresem IP serwisu.

Przykładem jest poniższa deklaracja:

```
apiVersion: v1
kind: Service
metadata:
  name: admin-application-service
  labels:
    app: admin-application-service
    tier: backend
spec:
  selector:
    app: admin-application-service
  type: LoadBalancer
  ports:
  - port: 8085
    targetPort: 80
    protocol: TCP
    name: http
```

Jest to deklaracja typu *Service*, która powinna zawierać następujące parametry:

- Wersja wykorzystywanego API
- Typ deklaracji
- Nazwa deklaracji
- Selektor. Od niego zależy, które pod-y zostaną dołączone do zbioru
- Typ publikacji
- Porty

Wyróżnia się trzy główne typy publikacji serwisu:

- ClusterIP - typ domyślny. Przydziela serwisowi wewnętrzny adres IP w klastrze, przez co serwis jest dostępny jedynie dla innych obiektów uruchomionych wewnątrz klastra
- NodePort - przydziela serwisowi statyczny numer portu na każdym węźle w klastrze. Dzięki temu serwis jest dostępny dla obiektów znajdujących się poza kla-

strem i można się do niego dostać przy pomocy adresu IP węzła oraz statycznego numeru portu

- LoadBalancer - przydziela serwisowi adres zewnętrzny przy użyciu load balancer-a zapewnionego przez wykorzystywaną platformę chmurową

Dobrą praktyką jest utworzenie obiektu wejścia do klastra (ang. *ingress*), który zarządza dostępem do klastra z zewnątrz. Typowo jest to obiekt API, który udostępnia ścieżki protokołu HTTP(S) prowadzące do serwisów znajdujących się wewnątrz klastra.

Przykład deklaracji znajduje się poniżej.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/affinity: cookie
    nginx.ingress.kubernetes.io/session-cookie-hash: sha1
    nginx.ingress.kubernetes.io/session-cookie-name:
        REALTIMESERVERID
    nginx.org/websocket-services:
        "admin-application-service"
spec:
  tls:
    - hosts:
      - dagair.info
      secretName: ingress-cert
  rules:
    - host: dagair.info
      http:
        paths:
          - path: /adminapplication
            pathType: Prefix
            backend:
              service:
                name: web-admin-app
                port:
                  number: 8085
```

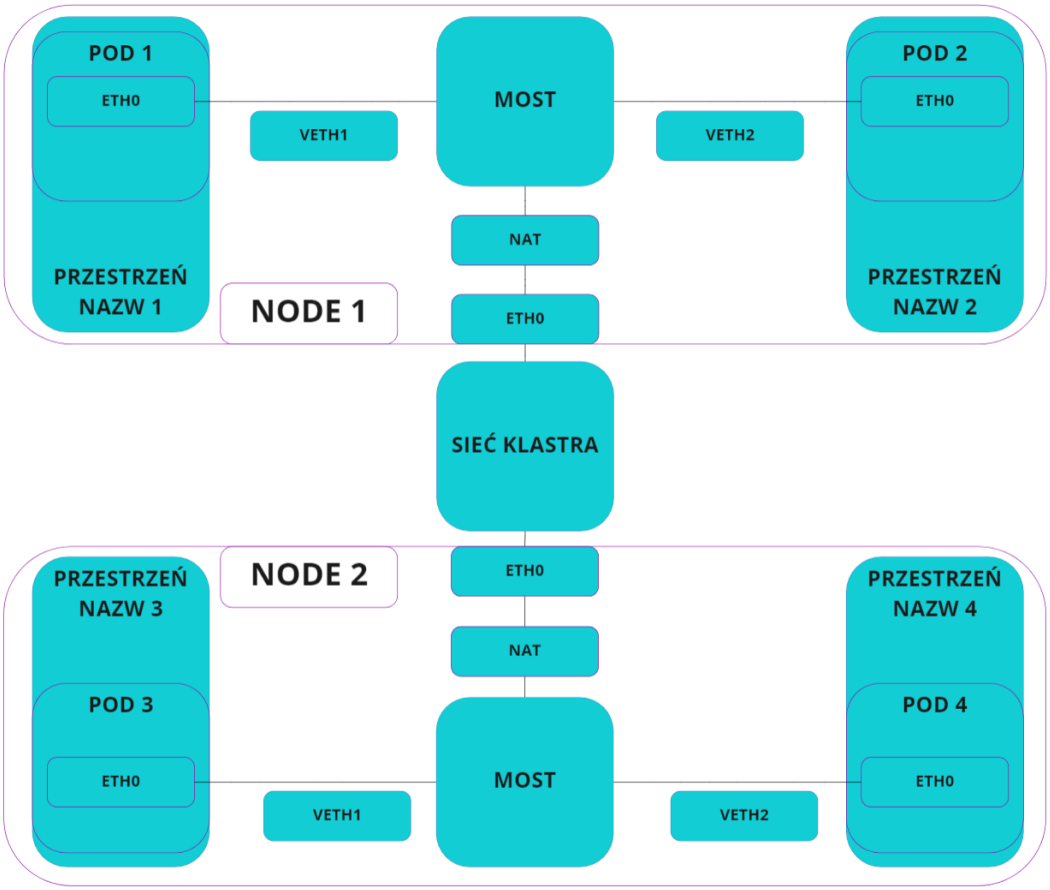
Jest to deklaracja typu *Ingress*, która powinna zawierać następujące parametry:

- Wersja wykorzystywanego API
- Typ deklaracji
- Nazwa deklaracji
- Specyfikacja, która zawiera reguły związane z dostępem do poszczególnych serwisów w klastrze. W tym przypadku aplikacja dla administratorów jest dostępna pod adresem dagair.info/adminapplication

9.4.3. Łączność sieciowa w środowisku Kubernetes

Łączność sieciowa w środowisku Kubernetes w dużym stopniu pokrywa się z zasadami funkcjonowania na platformie Docker, które zostały opisane w podrozdziale 9.1.1. Wewnątrz hosta utworzone są odrębne przestrzenie nazw, połączone ze sobą mostem. Dodatkowym poziomem abstrakcji są pody, wewnątrz których można uruchomić wiele kontenerów. Każdy pod otrzymuje unikalny adres IP. Kontenery wewnątrz poda mogą komunikować się ze sobą za pomocą interfejsu *loopback*. W przypadku konieczności komunikacji z mikrousługami znajdującymi się na innych hostach pakiety należy wysłać poprzez interfejs *ETH0* hosta. Kubernetes oferuje serwis DNS pozwalający na translację nazw serwisów na adresy IP.

Rysunek 9.3 przedstawia strukturę sieciową klastra.



Rysunek 9.3. Łączność sieciowa w środowisku Kubernetes. Opracowanie własne

10. Podsumowanie

W ramach niniejszej pracy udało się osiągnąć następujące rezultaty:

- Każdy z serwisów został utworzony, uruchamia się prawidłowo i skutecznie komunikuje się z innymi serwisami przy pomocy brokera wiadomości lub żądania http
- Utworzono obrazy kontenerów każdego z serwisów. Uruchamiają się prawidłowo i skutecznie komunikują się między sobą, zarówno na środowisku lokalnym, jak i wewnątrz klastra kubernetesowego
- Graficzny interfejs użytkownika pozwala zarządzać organizacją, oddziałami oraz pomieszczeniami, wyświetla także aktualne wyniki pomiarów
- Zestaw pomiarowy w regularnych odstępach czasu zbiera pomiary i wysyła je na kolejkę do brokera wiadomości. Wiadomości są pobierane przez SSDS i przesyłane dalej
- Skonfigurowano środowisko na platformie Azure, dzięki czemu system był publicznie dostępny w internecie. Ze względu na znaczące koszty utrzymania środowiska zdecydowano się na jego usunięcie. Możliwe jest jednak szybkie przywrócenie do pełnej funkcjonalności.

System do zarządzania warunkami technicznymi w pomieszczeniach biurowych jest publicznie dostępny w serwisie GitHub[25].

11. Ograniczenia

Pierwszym z możliwych kierunków dalszego rozwoju jest dodanie następujących komponentów:

- Rozszerzenie styków udostępnianych przez mikroserwisy o nowe usługi. W rozdziale 6. przedstawiono listę zaimplementowanych styków. Nie definiują one m. in. usług, które umożliwiały by aktualizację przechowywanych danych i akceptujących czasownik HTTP UPDATE.
- Rozszerzenie interfejsu graficznego o nowe strony, dodanie przycisków umożliwiających wykorzystanie usług akceptujących czasownik HTTP UPDATE

Projekt został przygotowany z myślą o tym, by można było możliwie łatwo tworzyć nowe serwisy i integrować je z już istniejącymi. Ważnym elementem ułatwiającym dodawanie nowych usług jest jasne zdefiniowane styków oferowanych przez inne serwisy usługowe.

Na ten moment nie zaimplementowano rozwiązań automatyzujących wykonywanie wymaganych czynności w przypadku, gdy warunki rzeczywiste panujące w danym pomieszczeniu nie spełniają oczekiwań. Jednym z kierunków dalszego rozwoju projektu jest wykorzystanie towarów produkowanych przez firmę Ikea. Zastosowanie inteligentnego oświetlenia wykorzystującego protokół ZigBee pozwoliłoby na automatyczne sterowanie poziomem natężenia światła przez aplikację. W tym celu należałoby utworzyć nowy serwis wykorzystujący gotową bibliotekę *pytradfri* [26] pozwalającą na zarządzanie oświetleniem.

Wartą rozważenia opcją jest utworzenie dedykowanej aplikacji mobilnej na urządzenia obsługujące systemy Android lub iOS. Aplikacja oferowałaby podobny zestaw funkcjonalności co aplikacje webowe, a dodatkowo posiadałaby system powiadomień, który generowałby wiadomości typu push za każdym razem przy wystąpieniu konkretnego zdarzenia, jak na przykład przekroczenie ustalonego progu temperatury lub natężenia oświetlenia.

Propozycją dalszego rozwoju projektu jest dodanie innych czujników i przeprowadzanie kompleksowych pomiarów dla danego pomieszczenia. Możliwymi rozszerzeniami są między innymi czujnik wilgotności powietrza lub czujnik jakości powietrza. Na ich podstawie można by generować bardziej szczegółowe wyniki i w ten sposób jeszcze bardziej zwiększyć wydajność pracowników oraz ograniczyć zużycie energii. Atutem utworzonego systemu jest łatwość dodawania nowych czujników, wystarczy je bowiem podpiąć przy pomocy zestawu przewodów połączeniowych do wolnych wejść mikrokontrolera, a następnie dodać oprogramowanie umożliwiające odczyt pomiarów.

Do tej pory wystarczającym instrumentem do sprawdzenia poprawności działania systemu było przeprowadzenie testów jednostkowych, integracyjnych oraz end-2-end.

Jednak wraz z dalszym rozwojem systemu konieczne będzie skonfigurowanie jego monitoringu. Powodów jest wiele, między innymi:

- Monitorowanie ogólnego stanu, w jakim znajduje się system
- Zbieranie metryk umożliwiających oszacowanie wydajności systemu
- Upewnienie się, że oferowane przez mikroserwisy usługi są dostępne i na bieżąco przetwarzają żądania
- Możliwość odnalezienia fragmentów systemu, które spowalniają jego działanie lub w ogóle nie funkcjonują

Jednym z istniejących narzędzi do zbierania kolekcjonowania metryk opisujących stan systemu jest Prometheus[27]. Jest to narzędzie do monitorowania aplikacji, w szczególności przeznaczone do systemów rozproszonych, opartych na architekturze mikroserwisowej. Pozwala agregować dane dotyczące zużycia procesora, pamięci czy dysku, a także tworzyć własne metryki, które mogą przykładowo mierzyć czas potrzebny na przetworzenie przychodzących żądań http.

Bibliografia

[1] I. for Market Transformation. "ENERGY BENCHMARKING AND TRANSPARENCY". (2015), adr.: https://www.imt.org/wp-content/uploads/2018/02/IMTBenefitsofBenchmarking_Online_June2015.pdf (term. wiz. 08. 01. 2022).

[2] N. Oseland i A Burton, "Quantifying the impact of environmental conditions on worker performance for inputting to a business case to justify enhanced workplace design features", *Journal of Building Survey*, t. 1, nr. 2, s. 151–164, 2012.

[3] Sharp. "Monitor do współpracy w systemie Windows". (2022), adr.: https://www.sharp.pl/cps/rde/xchg/pl/hs.xsl/-/html/windows-collaboration-display.htm (term. wiz. 08. 01. 2022).

[4] L. Lan, P. Wargocki i Z Lian, "Optimal thermal environment improves performance of office work", *Rehva*, s. 12–17, 2012.

[5] C. Dai, L. Lan i Z Lian, "Method for the determination of optimal work environment in office buildings considering energy consumption and human performance", *Energy and Buildings*, t. 76, s. 278–283, 2014.

[6] A. Hedge, W. Sakr i A Agarwal, "Thermal Effects on Office Productivity", *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, t. 49, nr. 8, s. 8, 2005.

[7] L. Chin-Chiuan i H. Kuo-Chen, "Effects of Lighting Color, Illumination Intensity, and Text Color on Visual Performance", *International Journal of Applied Science and Engineering*, t. 12, nr. 3, s. 193–202, 2014.

[8] T. Liu, C. Lin, K. Huang i Y Chen, "Effects of noise type, noise intensity, and illumination intensity on reading performance", *Applied Acoustics*, t. 120, s. 70–74, 2017.

[9] S. Newmam, "Building microservices", w O'Reilly, 2015, s. 16–27.

[10] C. de la Torre, B. Wagner i M. Rousos, ".NET Microservices: Architecutre for containerized .NET applications", w Microsoft, 2022, s. 31–32.

[11] I. Sommerville, "Software Engineering", w Pearson, 2011, s. 72–73.

[12] O. Corporation. "General Information". (2022), adr.: https://dev.mysql.com/doc/refman/8.0/en/introduction.html (term. wiz. 08. 01. 2022).

[13] C Richardson. "Microservices Pattern: Database per service". (2021), adr.: https://microservices.io/patterns/data/database-per-service.html (term. wiz. 08. 01. 2022).

[14] A Vickers. "Identity model customization in ASP.NET Core". (2021), adr.: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-5.0#the-identity-model (term. wiz. 08. 01. 2022).

[15] I. Inc. "Why use influxdb". (2022), adr.: https://www.influxdata.com/products/influxdb/ (term. wiz. 08. 01. 2022).

[16] RabbitMq. “AMQP 0-9-1 Model Explained”. (2022), adr.: https://www.rabbitmq.com/tutorials/amqp-concepts.html (term. wiz. 08. 01. 2022).

[17] MassTransit. “Transports”. (2022), adr.: https://masstransit-project.com/usage/transports/ (term. wiz. 08. 01. 2022).

[18] S. Newman, *Building microservices*. O’Reilly, 2015.

[19] R. Fielding, J. Mogul, J. Gettys i in. “rfc2616”. (1999), adr.: https://datatracker.ietf.org/doc/html/rfc2616#section-10 (term. wiz. 08. 01. 2022).

[20] M. Koch. “Getting started”. (2022), adr.: https://nuke.build/docs/getting-started/philosophy.html (term. wiz. 08. 01. 2022).

[21] D. Inc. “ASP.NET Core Runtime”. (2022), adr.: https://hub.docker.com/_/microsoft-dotnet-aspnet (term. wiz. 08. 01. 2022).

[22] D. Inc. “.NET SDK”. (2022), adr.: https://hub.docker.com/_/microsoft-dotnet-sdk (term. wiz. 08. 01. 2022).

[23] S. F. Conservancy. “Git”. (2022), adr.: https://git-scm.com/docs/git (term. wiz. 08. 01. 2022).

[24] Kubernetes. “What is Kubernetes?” (2022), adr.: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (term. wiz. 08. 01. 2022).

[25] S. Skrzypek. “DagAir”. (2022), adr.: https://github.com/fokamdk99/DagAir (term. wiz. 08. 01. 2022).

[26] Ikea. “pytradfri”. (2022), adr.: https://github.com/home-assistant-libs/pytradfri (term. wiz. 08. 01. 2022).

[27] Prometheus. “Overview”. (2022), adr.: https://prometheus.io/docs/introduction/overview/ (term. wiz. 08. 01. 2022).

Wykaz symboli i skrótów

AA – Admin Application
AAS – Admin Application Service
ADS – Addresses Data Service
EA – Employee Application
EAS – Employee Application Service
FDS – Facilities Data Service
PDS – Policies Data Service
PPS – Policy Processing Service
SDS – Sensors Data Service
SSDS – Sensor State Data Service
SSHDS – Sensor State History Data Service
SSPS – Sensor State Processing Service

Spis rysunków

Spis tabel

Spis załączników