Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Wydział Elektroniki i Technik Informacyjnych

Praca dyplomowa inżynierska

na kierunku Telekomunikacja
w specjalności Teleinformatyka i Zarządzanie

Aplikacja do zarządzania warunkami technicznymi w pomieszczeniach biurowych oparta na architekturze mikrousługowej

Stanisław Skrzypek
Numer albumu 300501

promotor
Dr hab. Inż. Prof. Artur Tomaszewski

Warszawa 2022

Streszczenie

Brak odpowiedniego zarządzania energią w budynkach biurowych od wielu lat powodował straty zarówno finansowe, jak i środowiskowe. Dodatkowo, w wyniku nieodpowiednich warunków panujących w pomieszczeniach pracy, osoby w nich przebywające nie mogły pracować w efektywny sposób. Celem tej pracy było zaproponowanie rozwiązania, za pomocą którego można byłoby mierzyć wartości kluczowych parametrów pomieszczeń biurowych, po czym podejmować stosowne działania mające na celu zarówno poprawienie komfortu pracowników, jak i redukcję zużywanej energii. Wykonano przegląd już istniejących prac naukowych dotyczących optymalnej wartości temperatury oraz natężenia światła w pomieszczeniach biurowych. Przygotowano system informatyczny oparty na architekturze mikrousługowej, który przyjmuje aktualne pomiary i je przetwarza. Przygotowano zestaw czujników, które wykonują pomiary oraz przesyłają je do systemu.

Summary

Lack of proper Energy management in office buildings has caused both financial, as well as environmental loss in many a year. Furthermore, as a result of inadequate room conditions, people staying in those rooms could not work effectively. The aim of this paper was to propose a solution by which it would be possible to measure the values of key parameters of office premises, and then take appropriate actions aimed at both improving the comfort of employees and reducing energy consumption. A review of the already existing scientific works on the optimal value of temperature and light intensity in offices was carried out. An IT system based on a microservice architecture was prepared, which takes current measurements and processes them. A set of sensors has been prepared that perform the measurements and send them to the system.

załącznik do zarządzenia nr 28/2016 r.
Rektora PW

Politechnika Warszawska

„załącznik nr 3 do zarządzenia nr 24/2016 Rektora PW

.....
miejscowość i data

.....
imię i nazwisko studenta

.....
numer albumu

.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta"

Spis treści

Temat pracy

Tematem niniejszej pracy jest utworzona w ramach seminarium dyplomowego aplikacja do zarządzania warunkami technicznymi w pomieszczeniach biurowych oparta na architekturze mikrousługowej. Produkt ma na celu poprawę warunków panujących w pomieszczeniach przeznaczonych do pracy codziennej. Wybrane parametry przeznaczone do optymalizacji to temperatura oraz natężenie światła.

Implementacja projektu przewiduje umieszczenie w badanych 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 akcji, które należy podjąć, aby umożliwić ustalenie się badanych parametrów na właściwym poziomie.

Utworzenie aplikacji miało przyczynić się do osiągnięcia dwóch głównych celów, stawianych od początku przygotowywania pracy:

1. Poprawa warunków pracy – badania (Oseland i Burton, 2012) 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 aplikacji będzie możliwe zapobieganie takim wydarzeniom, co w konsekwencji ograniczy zużycie energii

Zgodnie z wynikami badań opublikowanymi przez *Institute for market transformation* z 2016 roku około 40% całkowitej konsumpcji energii przypada na zasilanie budynków (transformation, 2016). Przekłada się to w ciągu roku na wydatek rzędu 450 miliardów dolarów. Najsłabiej zagospodarowane budowle zużywały od trzech do siedmiu razy więcej energii od tych najbardziej oszczędnych. Istnieje zatem potrzeba przygotowania i wdrożenia rozwiązań, które z jednej strony nie byłyby obciążające finansowo, z drugiej strony zaś ograniczające już istniejące koszty.

Istniejące rozwiązania

Firma Sharp przygotowała podobne rozwiązanie, za pomocą którego można mierzyć kluczowe parametry danego pomieszczenia, przesyłać je na platformy chmurowe i je analizować (Sharp, 2022). Różnica między tym produktem a rozwiązaniem proponowanym w tej 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 niejako dodatkiem do monitora, niż głównym celem 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 przesyłających pomiary do systemu, bez innych dodatków, co znacznie minimalizuje koszt wdrożenia takiego rozwiązania.

Założenia

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

- Łatwość wdrożenia – system powinien być gotowy do wdrożenia na środowisko chmurowe. Organizacja zainteresowana uruchomieniem aplikacji dla swoich potrzeb może wybrać opcję, w której dostarczane są obrazy odpowiednich serwisów 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 serwisów, które je przetwarzają. Do pomiaru zalicza się aktualna temperatura, natężenie światła oraz jakość powietrza
- Aplikacja oparta 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 aplikacja przygotowuje wynik. Domyślnie istnieje reguła podstawowa, gdzie oczekiwana temperatura wynosi $24,5^0$. Więcej informacji odnośnie tego skąd taka wartość została ustalona można uzyskać, patrząc na tabelę 1
- 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

Poniższa tabela pokazuje porównanie wyników z różnych artykułów traktujących o optymalnej temperaturze w pomieszczeniach:

Badanie	Optymalna temperatura
(Lan, Wargocki i Lian, 2012)	23.5^0 C – 25.5^0 C
(Dai, Lan i Lian, 2014)	23^0 C – 26.5^0 C
(Hedge, Sakr i Agarwal, 2005)	24^0 C - 25^0 C

Tabela 1: Porównanie wyników badań estymujących optymalną temperaturę

W oparciu o powyższe badania wyliczono średnią optymalną temperaturę wynoszącą 24,5.

Kolejna z tabel porównuje poziom natężenia światła, który skutkował najlepszą efektywnością pracowników:

Badanie	Optymalne natężenie światła
(Lin i Huan, 2014)	500 lx
(Liu, Lin, Huang i Chen, 2017)	600 lx

Tabela 2: Porównanie wyników badań estymujących optymalne natężenie światła

W oparciu o powyższe badania wyliczono średnią optymalne natężenie światła wynoszące 550 lx.

Metodologia

Największy nacisk w trakcie tworzenia pracy został położony na łatwość wdrożenia. W poniższych podrozdziałach został opisany sposób, w jaki ten cel osiągnięto.

Architektura systemu

Wśród możliwych architektur systemów można wyłonić dwie najważniejsze gałęzie: architektura monolityczna lub oparta na mikroserwisach, które są małymi, niezależnymi od siebie aplikacjami, wspólnie ze sobą współpracującymi. Pierwsza z opcji opiera się na idei polegającej na tym, że wszelka funkcjonalność danego systemu jest zamknięta w jednym projekcie i funkcjonuje jako całość. Druga możliwość opiera się na rozdzieleniu funkcjonalności na wiele mniejszych podprogramów, 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 3. Przedstawiono porównanie obydwu architektur, które uwzględnia:

- Odporność systemu na awarie. System powinien być w każdym momencie dostępny dla użytkowników
- Skalowalność. Potrzeba skalowania wynika ze zbyt dużego obciążenia dla jednego lub większej ilości serwisów 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

cecha	System monolityczny	System oparty na architekturze mikroserwisowej
Odporność systemu na awarie	W przypadku awarii w jednym punkcie, cały system przestaje działać	awaria jednego z serwisów niekoniecznie musi oznaczać niesprawność całego systemu
Skalowalność	Wymusza zwiększanie liczby instancji wszystkich usług, nawet jeśli zapotrzebowanie na część z nich jest małe	Możliwość zwiększania liczby instancji tylko tych usług, które w danym momencie są silnie obciążone
Łatwość wdrożenia	Nawet mała zmiana w kodzie aplikacji monolitycznej 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

Tabela 3: porównanie popularnych architektur systemów

Biorąc pod uwagę zestawienie z tabeli 3. Postanowiono wykorzystać architekturę mikrousługową w celu implementacji systemu.

Serwisy zorientowane usługowo

Docelowo, system oparty na architekturze mikrousługowej powinien składać się z serwisów zorientowanych usługowo (ang. service-oriented architecture). Stanowią one konstrukcję, w której wiele serwisów 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.

Sprzężenie serwisów

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

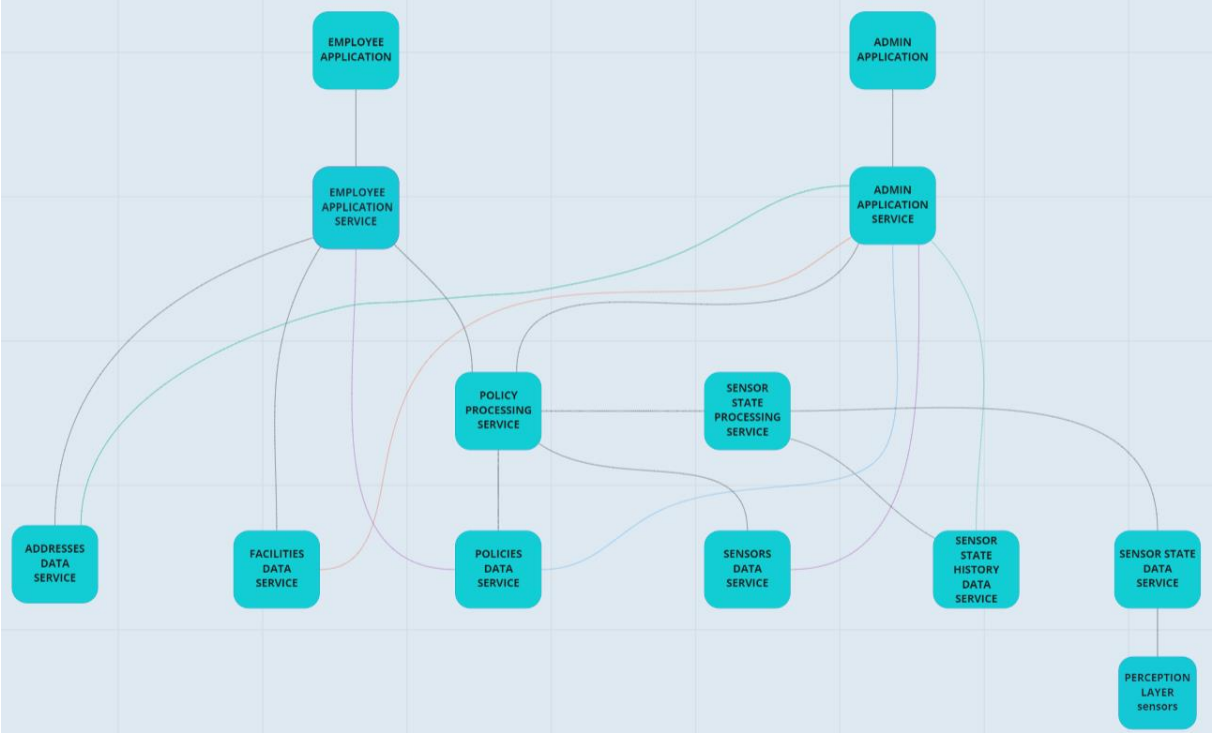
Spójność serwisów

W prawidłowo skonstruowanym systemie mikrousługowym funkcjonalność związana ze sobą (np. w kontekście biznesowym) jest umieszczona w jednym

miejscu. O tak zaprojektowanych serwisach mówi się, że są one spójne (ang. *high cohesion*). Przykładem błędnej implementacji może być edycja danych osobowych klienta w wielu serwisach. Wtedy zmiana w jednym serwisie może wymagać zmiany w innych.

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 1. przedstawia pełną architekturę aplikacji:



Rysunek 1: Architektura systemu

Na całość składają się serwisy opisane poniżej. System zawiera mikrousługi 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

Tabela 4: mikrousługi danych

Poza mikrousługami danych aplikacja posiada również serwisy przetwarzające dane:

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

Tabela 5: serwisy przetwarzające

Na system składają się także mikrousługi aplikacyjne:

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

Tabela 6: mikrousługi aplikacyjne

Ostatnimi elementami systemu są aplikacje dla poszczególnych ról użytkowników:

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

Tabela 7: aplikacje użytkowników

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

MySQL

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

Dobrą praktyką, którą warto mieć na uwadze podczas tworzenia systemu opartego na architekturze mikroserwisowej, jest zapewnienie dostępu do danego zbioru danych tylko jednemu mikroserwisowi, który następnie może je udostępniać przy pomocy odpowiednio skonfigurowanego API (Richardson, 2021). Takie podejście umożliwia zachowanie luźnego sprzężenia między serwisami. W konsekwencji należy utworzyć oddzielne zbiory danych, zwane schematami, które mogą być zarządzane przez pojedynczy mikroserwis.

W ramach utworzonego serwera bazodanowego zostały wdrożone następujące schematy:

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

Tabela 8: utworzone schematy bazodanowe

Schemat adresów

Schemat adresów składa się z następujących encji:

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
street	Ulica
Street_numbers	Numer budynku

Tabela 9: Encje w schemacie adresów

Związki między poszczególnymi encjami zostały opisane w tabeli 10.

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	street	1:N	Każda ulica może występować w wielu adresach
addresses	Street_numbers	1:N	Każdy numer budynku może występować w wielu adresach

Tabela 10: związki między encjami w schemacie adresów

Schemat organizacji

Schemat adresów składa się z następujących encji:

Nazwa encji	Przechowywane dane
affiliates	Oddziały danej organizacji
rooms	Pomieszczenia w oddziałach
organizations	organizacje

Tabela 11: Encje w schemacie organizacji

Związki między poszczególnymi encjami zostały opisane w tabeli 12.

relacja		Typ związku	Opis
organizations	affiliates	1:N	Każda organizacja może posiadać wiele oddziałów
affiliates	rooms	1:N	Każdy oddział może posiadać wiele pomieszczeń

Tabela 12: Związki między encjami w schemacie organizacji

Schemat reguł

Schemat reguł składa się z następujących encji:

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

Tabela 13: Encje w schemacie reguł

Związki między poszczególnymi encjami zostały opisane w tabeli 14.

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

Tabela 14: Związki między encjami w schemacie reguł

Schemat sensorów

Schemat sensorów składa się z następujących encji:

Nazwa encji	Przechowywane dane
sensors	Szczegóły dotyczące danego sensora
Sensor_models	Model sensora
producers	Producent sensorów

Tabela 15: Encje w schemacie sensorów

Związki między poszczególnymi encjami zostały opisane w tabeli 16.

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 wyprodukowany wielokrotnie

Tabela 16: Związki między encjami w schemacie sensorów

Schemat użytkowników

Schemat użytkowników został oparty na schemacie oferowanym przez Microsoft (Vickers, 2021) i składa się z następujących encji:

Nazwa encji	Przechowywane dane
aspnetusers	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

Tabela 17: Encje w schemacie użytkowników

Związki między poszczególnymi encjami zostały opisane w tabeli 18.

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

Tabela 18: Związki między encjami w schemacie użytkowników

InfluxDB

InfluxDB jest bazą danych szeregów czasowych służącą do przechowywania metryk i szeregów czasowych. Umożliwia efektywne przechowywanie miliardów rekordów danych dzięki stosowaniu kompresji danych, oferuje także język zapytań pozwalający przeprowadzać kompleksowe analizy uzyskanych danych. 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 tej pracy inżynierskiej 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 następujące parametry:

Parametr	Znaczenie
_time	Czas otrzymania danych
_field	Rodzaj danych
_value	Zmierzona wartość

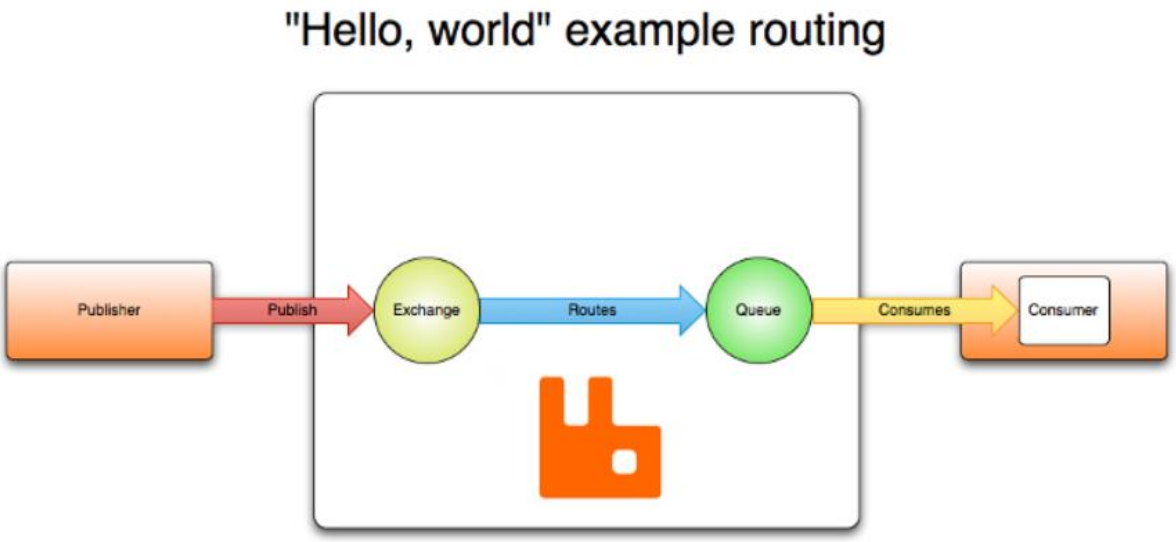
Tabela 19: Parametry pojedynczego rekordu przechowującego pomiar

Komunikacja między serwisami

Broker wiadomości

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 serwisami. 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 protokoł, został przedstawiony na rysunku 3.



Rysunek 2: Schemat protokołu AMQP 0-9-1. Źródło: AMQP 0-9-1 Model Explained — RabbitMQ

Wydawcy wiadomości (ang. *publisher*) publikują wiadomości do pośrednika, który następnie przekuje je do odpowiednich konsumentów (ang. *consumer*). Ponieważ jest to protokół sieciowy, to wydawcy, konsumenci oraz pośrednicy mogą być uruchomieni na różnych maszynach. Pośrednik RabbitMq dysponuje następującymi cechami:

- 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 usług.

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 (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 usług
- Wymiana tematyczna (ang. topic exchange) – kierują 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, może skorzystać z kanałów (ang. channels), o których można myśleć jako 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.

MassTransit

Przy tworzeniu systemu została wykorzystana szyna danych o nazwie MassTransit przeznaczona dla aplikacji napisanych we framework'u .NET Core. 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 infrakstruktury 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)
{
    _logger.LogInformation($"Received MeasurementSentEvent. Temperature:
{context.Message.Temperature}," +
                           $"Illuminance: {context.Message.Illuminance}" +
                           $"Humidity: {context.Message.Humidity}" +
                           $"room id: {context.Message.RoomId}");
    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 znajduje się w innym mikroserwisie.

Styki

Mikroserwisy mogą komunikować się między sobą przy użyciu oferowanych przez nie styków (ang. *Application Programming Interface*). Definiują one kontrakty określające informacje wymagane przy wysyłaniu żądania 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. Każdy ze styków zawiera:

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

Styk mikrousługi danych adresów

Opis: Zwraca informacje o adresie z identyfikatorem *addressId*

Ścieżka: GET /addresses-api/addresses/{addressId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	addressId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	AddressDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Zwraca informacja o mieście z identyfikatorem *cityId*

GET /addresses-api/cities/{cityId}

Parametry wejściowe

Typ	Nazwa	Schemat
Path	cityId *wymagany*	integer (int64)

Odpowiedź

Kod HTTP	Opis	Schemat
200	Success	CityDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Zwraca informacje o kraju z identyfikatorem *CountryId*

Ścieżka: GET /addresses-api/countries/{countryId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	countryId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	CountryDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Zwraca informacje o kodzie pocztowym z identyfikatorem *postalCodeId*

Ścieżka: GET /addresses-api/postal-codes/{postalCodeId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	postalCodeId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	PostalCodeDtoJsonApiDocument
404	Not Found	JsonApiError

Styk mikrousługi danych organizacji

Opis: Tworzy nowy oddział o parametrach przekazanych w obiekcie typu *addNewAffiliateCommand*

Ścieżka: POST /facilities-api/affiliates

Parametry wejściowe:

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewAffiliateCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	AffiliateDtoJsonApiDocument
400	Bad Request	JsonApiError
409	Conflict	JsonApiError

Opis: Zwraca informacja o wszystkich oddziałach

Ścieżka: GET /facilities-api/affiliates

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	AffiliateDtoListJsonApiDocument

Opis: Zwraca informacje o oddziale z identyfikatorem *affiliateId*

Scieżka: GET /facilities-api/affiliates/{affiliateId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	affiliateId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	AffiliateDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Tworzy nową organizację o parametrach przekazanych w obiekcie typu *addNewOrganizationCommand*

Ścieżka: POST /facilities-api/organizations

Parametry wejściowe:

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewOrganizationCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	OrganizationDtoJsonApiDocument
400	Bad Request	JsonApiError
409	Conflict	JsonApiError

Opis: Zwraca informacje o wszystkich organizacjach

Ścieżka: GET /facilities-api/organizations

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	OrganizationDtoListJsonApiDocument
404	Not Found	JsonApiError

Opis:Zwraca informacje o organizacji z identyfikatorem *organizationId*

Ścieżka: GET /facilities-api/organizations/{organizationId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	organizationId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	OrganizationDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Tworzy nowy pokój o parametrach określonych w obieckie typu *addNewRoomCommand*

Ścieżka: POST /facilities-api/rooms

Parametry wejściowe:

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewRoomCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	RoomDtoJsonApiDocument
400	Bad Request	JsonApiError
409	Conflict	JsonApiError

Opis: Zwraca informacje o pomieszczeniu z identyfikatorem *roomId*

Ścieżka: GET /facilities-api/rooms/{roomId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	roomId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	RoomDtoJsonApiDocument
404	Not Found	JsonApiError

Styk mikrousługi danych reguł

Opis:Tworzy nowy zestaw oczekiwanych warunków panujących w pomieszczeniu o parametrach określonych w obiekcie typu *addNewExpectedRoomConditionsCommand*

Ścieżka: POST /policies-api/expected-room-conditions

Parametry wejściowe:

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewExpectedRoomConditionsCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	ExpectedRoomConditionsDtoJsonApiDocument
400	Bad Request	JsonApiError

Opis: Zwraca informacje o zestawie oczekiwanych warunków panujących w pomieszczeniu z identyfikatorem *expectedRoomConditionsId*

Ścieżka: GET /policies-api/expected-room-conditions/{expectedRoomConditionsId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	expectedRoomConditionsId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Nazwa	Schemat
200	Success	ExpectedRoomConditionsDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Tworzy nową regułę o parametrach określonych w obiekcie typu *addNewRoomPolicyCommand*

Ścieżka: POST /policies-api/policies

Parametry wejściowe:

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewRoomPolicyCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	RoomPolicyDtoJsonApiDocument
400	Bad Request	JsonApiError

Opis: Zwraca informacje o obecnie obowiązującej regule w pomieszczeniu z identyfikatorem *roomId*

Ścieżka: GET /policies-api/policies/{roomId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	roomId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Nazwa	Schemat
200	Success	RoomPolicyDtoJsonApiDocument
404	Not Found	JsonApiError

Styk mikrousługi danych sensorów

Opis: Zwraca informacje o wszystkich sensorach

Ścieżka: GET /sensors-api/sensors

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	SensorDtoListJsonApiDocument

Opis: Zwraca informacje o sensorze z identyfikatorem *sensorId*

Ścieżka: GET /sensors-api/sensors/{sensorId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	sensorId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	SensorDtoJsonApiDocument
404	Not Found	JsonApiError

Styk mikrousługi danych mikrousługi aplikacyjnej administratorów

Opis: Tworzy nowy adres o parametrach określonych w obiekcie typu *AddNewAddressCommand*

Ścieżka: POST /admin-node/addresses

Parametry wejściowe

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewAddressCommand

Odpowiedź:

Kod HTTP	Nazwa	Schemat
201	Success	AddressDtoJsonApiDocument
400	Bad Request	JsonApiError

Opis: Tworzy nowy oddział o parametrach określonych w obiekcie typu *AddNewAffiliateCommand*

Ścieżka: POST /admin-node/affiliates

Parametry wejściowe

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewAffiliateCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	AffiliateDtoJsonApiDocument
400	Bad Request	JsonApiError
409	Conflict	JsonApiError

Opis: Zwraca informacje o wszystkich oddziałach

Ścieżka: GET /admin-node/affiliates

Odpowiedź:

Kod http	Opis	Schemat
200	Success	AdminNodeAffiliateDtoListJsonApiDocument
404	Not Found	JsonApiError

Opis: Zwraca informacje o oddziale z identyfikatorem *affiliateId*

Ścieżka: GET /admin-node/affiliates/{affiliateId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	affiliateId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	AdminNodeAffiliateDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Tworzy nową organizację o parametrach określonych w obiekcie typu *AddNewOrganizationCommand*

Ścieżka: POST /admin-node/organizations

Parametry wejściowe:

Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewOrganizationCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	OrganizationDtoJsonApiDocument
400	Bad Request	JsonApiError

Opis: Zwraca informacje o wszystkich organizacjach

Ścieżka: GET /admin-node/organizations

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	AdminNodeOrganizationDtoListJsonApiDocument
404	Not Found	JsonApiError

Opis: Zwraca informacje o organizacji z identyfikatorem *organizationId*

Ścieżka: GET /admin-node/organizations/{organizationId}

Parametry wejściowe:

Typ	Nazwa	Schemat
Path	organizationId *wymagany*	integer (int64)

Odpowiedź:

Kod HTTP	Opis	Schemat
200	Success	AdminNodeOrganizationDtoJsonApiDocument
404	Not Found	JsonApiError

Opis: Tworzy nowe pomieszczenie o parametrach określonych w obiekcie typu *AddNewRoomCommand*

Ścieżka: POST /admin-node/rooms

Parametry wejściowe:

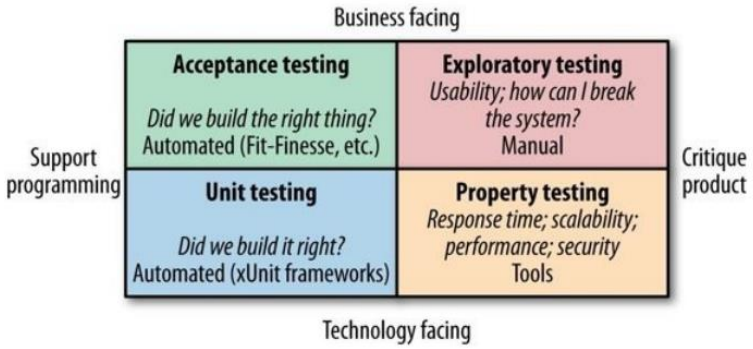
Typ	Nazwa	Schemat
Body	body *opcjonalny*	AddNewRoomCommand

Odpowiedź:

Kod HTTP	Opis	Schemat
201	Success	RoomDtoJsonApiDocument
400	Bad Request	JsonApiError
409	Conflict	JsonApiError

Testy

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



Rysunek 3: Rodzaje testów. Źródło: Newman, S. (2015). Building microservices. O'Reilly.

Dwie kategorie znajdujące się na dole – unit testing oraz property testing – mają pomóc deweloperom stworzenie działającego kodu. 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 ilości serwisów, sprawdzenie funkcjonalności systemu oraz tzw. „user acceptance testing", czyli testy przeprowadzone przez klienta, który zlecił budowę systemu.

Testy jednostkowe

Tego rodzaju testy sprawdzają poprawność pojedynczej funkcji w kodzie. Nie uruchamia się całego serwisu, a jedynie sprawdza jego 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 WhenMeasurementsAreTooHigh_ShouldReturnTooHighIndicators()
{
    // 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

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
GetExpectedRoomConditions_ShouldBeAvailableUnderDefinedPath()
{
    var path = $"policies-api/expected-room-conditions/1";

    var response = await _client.GetAsync(path);

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

Klient testowy korzysta z oferowanego przez testowany mikroserwis styku 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 (multiple authors, 1999).

Testy end-2-end

Testy typu end2-end mają za zadanie przetestować pewne biznesowe funkcjonalności, których realizacja może być obsługiwana przez wiele mikroserwisów. Dobrą praktyką jest utrzymywanie niewielkiej liczbę tego typu testów, ponieważ obejmują one zakres 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:

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

    # check that influxdb is available and bucket exists
new_measurement_test_helper.check_influxdb_is_available()

    # check that sensors_api is available
new_measurement_test_helper.check_sensors_api_is_available()

    # send new measurement to rabbitmq
new_measurement_test_helper.send_new_measurement_to_rabbitmq()
```

```
    # check that new measurement has been sent to sensors test queue

new_measurement_test_helper.check_message_sent_to_queue(new_measurem
ent_test_helper.rabbitmq_configuration.vhost,
                                            consts.sensors_test_queue, 1)

    # check that MeasurementSentEvent has been sent to test queue
new_measurement_test_helper.check_message_sent_to_queue(new_measurem
ent_test_helper.rabbitmq_configuration.vhost,

consts.measurement_sent_event_test_queue, 1)

    # check that PoliciesEvaluationResultEvent has been sent to test queue

new_measurement_test_helper.check_message_sent_to_queue(new_measurem
ent_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 na kolejce wiadomości.

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. Za jego pomocą uruchomienie wszystkich testów sprowadza się do wykonania jednej komendy.

Nuke pozwala na tworzenie własnych metod zwanych *targetami*, które automatyzują wykonywanie pewnych czynności. Każdy z targetów wykonują określoną logikę. Dodatkowo, można tworzyć kompleksową strukturę, w której każdy z targetów jest zależny od tego, czy prawidłowo zostanie wykonany inny target. Istnieje także możliwość dodania reguły wyzwalania poszczególnych targetów 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 projektu o nazwie *ProjectName*. Target zależy od innej metody 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 komenę *dotnet build* dla projektu o nazwie *ProjectName.* Jest ona zależna od targetu *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* na projekcie o nazwie *ProjectName*. Jednym z warunków uruchomienia tego targetu jest konieczność podania nazwy projektu, wyrażona przez funkcję *.Requires(() => ProjectName)*. Target jest zależny od innego targetu 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 targetu.

Wszystkie targety można uruchomić przy wykorzystaniu tylko jednej metody:

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

Automatyzacja wdrożenia

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.

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. Dane są zapisywane w folderze zwanym również 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 syntaktycznym, 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*), która jest rozszerzeniem umożliwiającym na uproszczenie całego przepływu pracy.

Przykładowy przepływ pracy utworzony na potrzeby pracy 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 wersja .NET 5.0
- Uruchamia testy jednostkowe i integracyjne
- Tworzy nową wersję obrazu przestestowanego 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 *commit*-a, który spowodował 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, trzeba było w poprzednim kroku zalogować się, wykorzystując nazwę repozytorium obrazów oraz danych uwierzytelniający, przechowywanych w bezpieczny sposób za pomocą Github Secrets.

Kubernetes

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.

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 aplikacji zajmowała większość zasobów, przez co inne aplikacje 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.

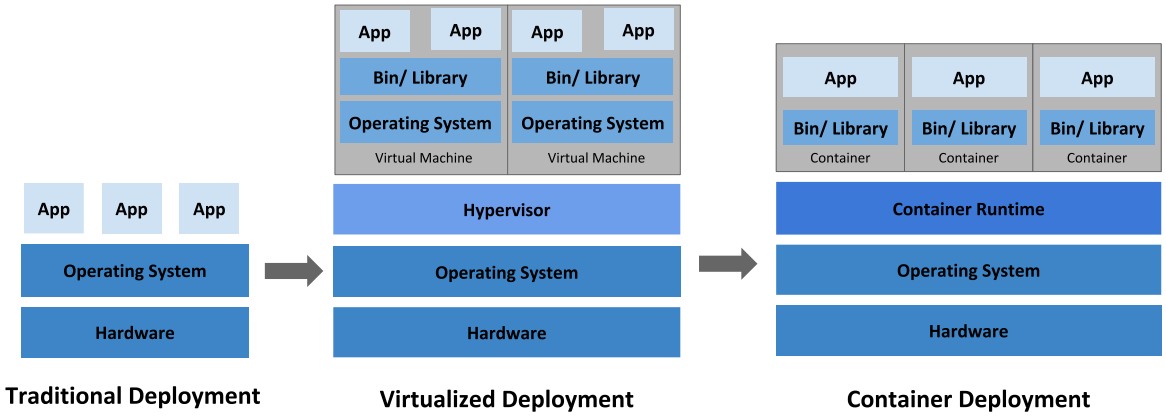
Najlepszym obecnie rozwiązaniem jest wykorzystanie kontenerów. Zawierają zestaw podobnych cech do maszyn wirtualnych z tą różnicą, że nie wymagają osobnego systemu operacyjnego. Każdy z kontenerów może współdzielić jeden

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
- Utworzenie obrazów na etapie budowania nowej wersji systemu zamiast na etapie wdrażania
- Niezależnie od środowiska działa w dokładnie ten sam sposób
- Mogą być uruchomione praktycznie na każdym systemie i dystrybucji
- Wysoka efektywność wykorzystania zasobów

Rysunek 3. przedstawia różnice między uruchomieniem aplikacji w sposób tradycyjny, przy użyciu maszyn wirtualnych oraz kontenerów.



Rysunek 4: Porównanie różnych metod uruchamiania aplikacji. Źródło: [What is Kubernetes? | Kubernetes]

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 gospodarza (ang. *master*), na której znajdują się wszystkie komponenty.

Komponenty płaszczyzny sterowania

W tej części zostały opisane komponenty składające się na całość płaszczyzny sterowania.

kube-apiserver - interfejs pozwalający na interakcję z płaszczyzną sterowania. Weryfikuje i konfiguruje dane dla obiektów takich jak serwisy czy kontrolery replikacji.

Etcd - to 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. 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

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. Cloud-controller-manager może zapewniać poniższe zależności:

- Kontroler węzłów (*node controller*) - sprawdza czy węzeł został usunięty z chmury po tym jak przestał odpowiadać na żądania
- Kontroler routingu (*route controller*) – zapewnia możliwość ustalenia ścieżek między poszczególnymi elementami infrastruktury chmurowej
- Kontroler serwisów (*service controller*) – zapewnia możliwość tworzenia, edytowania oraz usuwania *load balancer-ów*

Komponenty węzła

Poniżej opisano komponenty, które działają na każdym węźle w Kubernetesie:

- 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.

Deklarację można zaaplikować korzystając z komendy:

```
kubectl apply -f <file-name>.yml
```

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 serwis nie jest już osiągalny pod dotychczasowym adresem.

Rozwiązaniem tego zagadnienia jest wprowadzenie tzw. serwisu. 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:

```yaml
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 klastrem 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.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/affinity: cookie
    nginx.org/websocket-services: "admin-node"
spec:
  tls:
    - hosts:
        - dagair.info
      secretName: ingress-cert
  rules:
    - host: hello-world.info
      http:
        paths:
        - path: /adminapplication
          pathType: Prefix
          backend:
            service:
              name: admin-application
              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

Podsumowanie

Możliwości rozszerzenia projektu

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 pomysłów 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 (https://github.com/home-assistant-libs/pytradfri) pozwalającą na zarządzanie oświetleniem.

Bibliografia

Chin-Chiuan, L. and Kuo-Chen, H., 2014. Effects of Lighting Color, Illumination Intensity, and Text Color on Visual Performance. *International Journal of Applied Science and Engineering*, 12(3), pp.193-202.

Dai, C., Lan, L. and Lian, Z., 2014. Method for the determination of optimal work environment in office buildings considering energy consumption and human performance. *Energy and Buildings*, 76, pp.278-283.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, J., Leach, P. and Berners-Lee, T., 1999. *rfc2616*. [online] Datatracker.ietf.org. Available at: <https://datatracker.ietf.org/doc/html/rfc2616#section-10> [Accessed 8 January 2022].

Hedge, A., Sakr, W. and Agarwal, A., 2005. Thermal Effects on Office Productivity. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 49(8), pp.823-827.

Imt.org. 2015. *ENERGY BENCHMARKING AND TRANSPARENCY*. [online] Available at: <https://www.imt.org/wp-content/uploads/2018/02/IMTBenefitsofBenchmarking_Online_June2015.pdf > [Accessed 8 January 2022].

Lan, L., Wargocki, P. and Lian, Z., 2012. Optimal thermal environment improves performance of office work. *Rehva*, pp.12-17.

Liu, T., Lin, C., Huang, K. and Chen, Y., 2017. Effects of noise type, noise intensity, and illumination intensity on reading performance. *Applied Acoustics*, 120, pp.70-74.

Oseland, N. and Burton, A., 2012. 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,*, 1(2), pp.151-164.

Richardson, C., 2021. *Microservices Pattern: Database per service*. [online] microservices.io. Available at: <https://microservices.io/patterns/data/database-per-service.html> [Accessed 8 January 2022].

Sharp, 2022. *Monitor do współpracy w systemie Windows*. [online] Sharp.pl. Available at: <https://www.sharp.pl/cps/rde/xchg/pl/hs.xsl/-/html/windows-collaboration-display.htm> [Accessed 8 January 2022].

Vickers, A., 2021. *Identity model customization in ASP.NET Core*. [online] Docs.microsoft.com. Available at: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-5.0#the-identity-model> [Accessed 8 January 2022].

Wykaz symboli i skrótów

Spis rysunków

Spis tabel

Spis załączników