Politechnika Łódzka
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Wykorzystanie architektury zorientowanej na usługi do implementacji aplikacji rozproszonych

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Promotor: dr inż. Marcin Kwapisz

Dyplomant: Jakub Zygmunt

Nr albumu: 216938

Kierunek: Informatyka

Specjalność: Infrastruktura i Aplikacje Sieciowe

Łódź 15.01.2021 r.

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Streszczenie

Niniejsza praca przedstawia opis zbioru zagadnień związanych z tworzeniem systemu opartego o architekturę zorientowaną na usługi oraz ich zastosowanie. Część teoretyczna została wykorzystana do stworzenia implementacji wydajnego systemu do przeprowadzania ankiet. Następnie przeprowadzono badania w celu analizy założeń, których użyto przy implementacji przykładowej aplikacji. Praca skupia się na przybliżeniu zagadnień związanych z wzorcami: CQRS, Event Sourcing, Saga. Każdy z nich jest omówiony z zaznaczeniem zalet i wad oraz wykorzystany do budowy systemu zorientowanego na usługi.

Słowa kluczowe: SOA, CQRS, Event Sourcing, Saga, Broker wiadomości.

Abstract

This thesis presents a description of a set of issues related to the development of a system based on service-oriented architecture and their application. The theoretical part was used to create an implementation of an efficient system for conducting polls. This was followed by a study to analyze the assumptions that were used in the implementation of a sample application. The work focuses on introducing issues related to the patterns: CQRS, Event Sourcing, Saga. Each of them is discussed with pointing out advantages and disadvantages and used to build a service-oriented system.

Keywords: SOA, CQRS, Event Sourcing, Saga, Message Broker.

Spis treści

Słownik pojęć

Serwis – mechanizm umożliwiający dostęp do jednej lub więcej funkcji, gdzie dostęp jest zapewniony przy użyciu określonego interfejsu i jest wykonywany zgodnie z ograniczeniami i politykami określonymi przez opis usługi. [1]

Broker wiadomości – moduł, który odpowiada za przekazywanie komunikatów pomiędzy odbiorcę i nadawcę zgodnie z ustalonym protokołem. [2]

Skalowalność – atrybut, który umożliwia zmianę ilości zasobów systemu lub jego komponentów. [3]

Wysoka dostępność – cecha systemu, która oznacza spełnienie warunków czasu dostępności funkcjonalności dla użytkownika. Przykładowo system, który spełnia zasadę „pięciu dziewiątek" jest dostępny przez 99.999% czasu w ciągu roku. [4]

Wzorzec – rozwiązanie problemu związanego z implementacją systemu, który może być wielokrotnie użyty. Można wyróżnić wzorce projektowe oraz wzorce architektoniczne, które różnią się zakresem elementów systemu, które opisują. [5]

Dystrybutor żądań – moduł systemu, który odpowiada za rozdzielenie obciążenia na wiele komponentów. [6]

1. Wstęp

1.1 Geneza

Proces tworzenia oprogramowania ma przed sobą zbiór wymagań, które musi spełnić, aby pokryć założenia biznesowe produktu. W zależności od skali złożoności problemów do rozwiązania należy podejmować decyzje, które realizują potrzeby oraz umożliwią elastyczną modyfikację w przyszłości. Każdy przypadek można opisać przy pomocy architektury oprogramowania [7] , która odpowiada za organizację planu budowy oraz rozwoju systemu.

Dla przykładu można rozważyć przypadek, gdzie firma, z małym zespołem programistów, planuje szybkie wdrożenie usługi internetowej, tak aby zyskać pierwszych klientów. Na tym etapie wiadomo, że aspekty takie jak ochrona przed awarią czy skalowalność będzie miała mniejsze znaczenie. W przypadkach, gdzie oprogramowanie nie jest rozbudowane oraz posiada nieskomplikowaną logikę biznesową, często wybór pada na rozwiązanie oparte na architekturze monolitycznej [8]. Stosując podejście „Najpierw Monolit" [9] można sprawdzić czy pomysł na produkt oraz rozwiązanie spełniają wymagania klientów. Dzięki temu, że istnieje jedna instancja aplikacji, łatwo przeprowadzać testy integracyjne [10] oraz ciągle wdrażać rozwiązanie [11].

Wraz z rosnącą popularnością aplikacji wśród klientów, nieunikniona jest ewaluacja projektu, dostosowując swoją strukturę do zmieniających się wymagań biznesowych.
Niesie to za sobą kilka konsekwencji:

- Kolejne funkcjonalności mogą wpływać na inne komponenty, przez co łatwo można spowodować błąd podczas implementacji.
- W jednym miejscu jest przechowywana cała logika biznesowa, która z czasem osiągnie rozmiary znacznie utrudniające wdrożenie nowego członka zespołu.
- Rosnąca ilość obsługiwanych operacji, spowoduje zwiększenie się zapotrzebowania na zasoby (CPU, RAM), co zacznie generować wyższe koszty utrzymania.
- Zwiększająca się liczba programistów generuje konflikty w organizacji pracy oraz powoduje problemy w komunikacji przy zmianach w implementacji.
- Wprowadzane modyfikacje wymagają wdrożenia nowej wersji całego systemu.

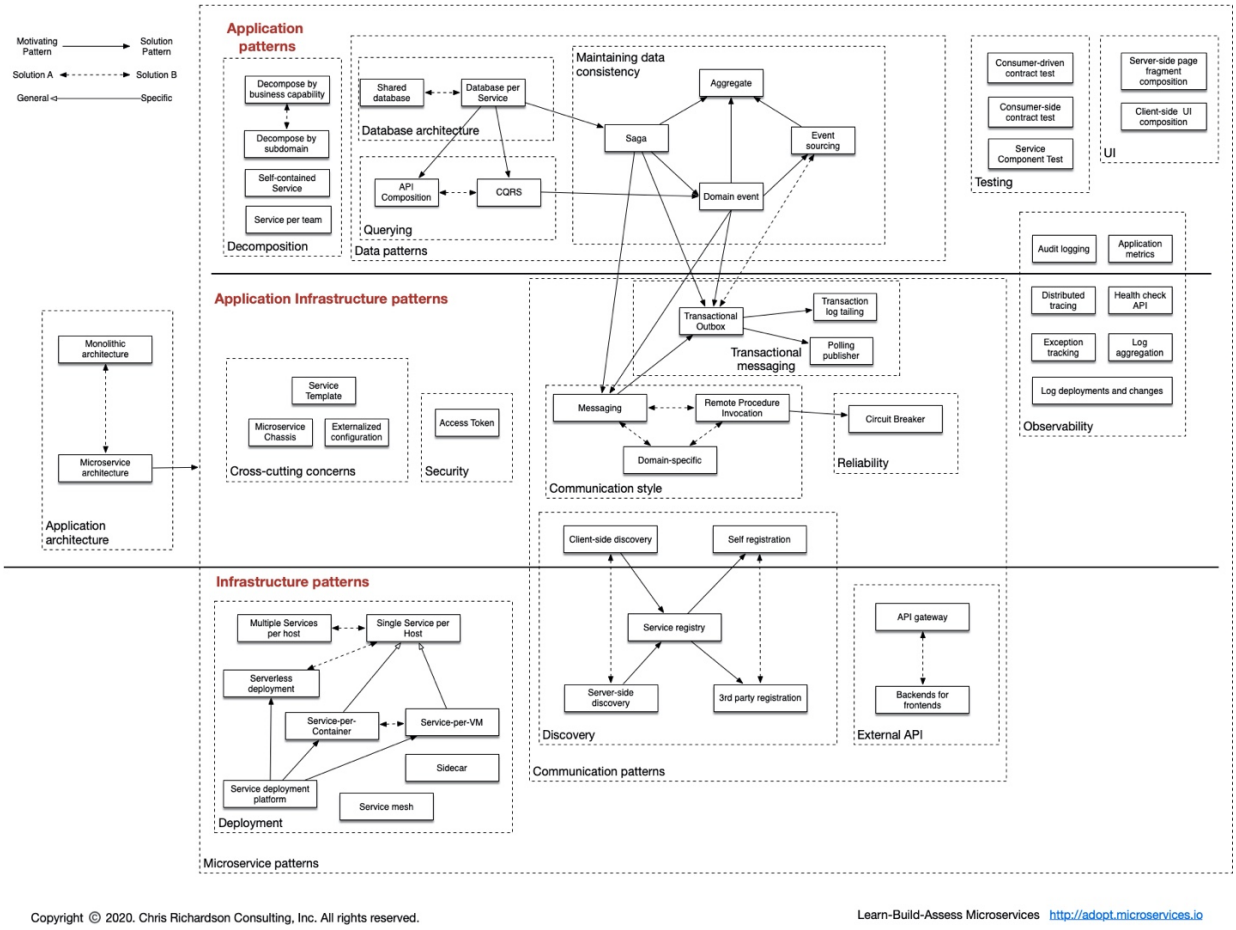
W wyniku wymienionych problemów każda kolejna zmiana w oprogramowaniu może generować większy nakład prac oraz dług techniczny [12].

Problem można rozwiązać na wiele sposobów, jednym z nich jest rozproszenie monolitu na luźno połączone komponenty [13], które są samodzielnymi składnikami sytemu połączonymi za pomocą udostępnionych interfejsów [14]. Każdy komponent powinien posiadać ograniczony zestaw funkcjonalności dotyczący jedynie odizolowanej części całego systemu. W ten sposób można wprowadzić wzorzec architektoniczny zorientowany na usługi [15]. Wymaga on przemyślanego planu organizacji zespołów, zasad komunikacji pomiędzy usługami i wsparcia rozbudowanej infrastruktury. Wdrożenie można rozwiązać poprzez zastosowanie konteneryzacji [16] usług oraz zastosowanie możliwości jakie udostępniają platformy PaaS (Platform as a Service) [17]. Ze względu na rozproszenie aplikacji na mniejsze serwisy trudno jest przeprowadzić testy. Sprawdzenie poprawność działania pojedynczego przypadku użycia wymaga uruchomienia części lub całego systemu.

Jeżeli grupa osób odpowiedzialnych za projekt poradzi sobie się z trudnościami technologicznymi oraz organizacyjnymi, wtedy powyższa zmiana przyniesie wiele korzyści:

- Skalowanie pojedynczych komponentów zamiast całego systemu pozwala oszczędzić zasoby infrastruktury.
- Wdrażane zmiany dotyczą tylko konkretnych elementów a nie całego systemu.
- Podział na mniejsze zespoły, które zajmują się funkcjonalnością danego serwisu. Skutkuje brakiem trudności z wdrożeniem nowej osoby do zespołu oraz mniej skomplikowaną logiką biznesową.
- Serwisy są niezależne technologicznie. Muszą jedynie spełniać interfejs oraz zasady komunikacyjne do integracji z resztą systemu.
- Ograniczenie awarii całego systemu.

Warto również zaznaczyć, że architektura zorientowana na usługi umożliwia zastosowanie wielu wzorców. Ich zbiór przedstawia Rysunek 1. Pozwala to stosować generyczne rozwiązania do zdefiniowanych typów problemów.



Rysunek 1 - Diagram przedstawia możliwe wzorce dla aplikacji zorientowanych na usługi [18]

1.2 Cel i zakres pracy

Głównym celem pracy jest zbadanie efektywności, skalowalności oraz możliwości architektury zorientowanej na serwisach z wykorzystaniem wzorców na przykładzie wybranego problemu.

W rozdziale 2 jest omówiona część teoretyczna, w której zostaną przedstawione zagadnienia oraz technologie.

W rozdziale 3 przedstawiono problem, który zawiera zarys wymagań funkcjonalnych i niefunkcjonalnych do spełnienia.

W rozdziale 4 znajduje się sposób rozwiązania problemu przez autora z wykorzystaniem wiedzy z rozdziału 2.

W rozdziale 5 są wyniki badań rozwiązania, wykonanych w celu weryfikacji opisanych założeń.

2. Dziedzina problemu

2.1 Architektura zorientowana na usługi (SOA)

Architektura zorientowana na usługi (SOA) jest to koncepcja, która gromadzi w sobie zagadnienia organizacyjne oraz techniczne, stosowane w celu połączenia bazy wiedzy informatycznej z biznesową. Umożliwia zwiększenie zwinności przy podejmowaniu decyzji związanych z utrzymaniem i rozwojem systemu. Dla standaryzacji oprogramowania zorientowanego na usługi istnieje baza wiedzy z opisem dobrych praktyk tworzenia aplikacji SaaS (software as a service) – *12 Factor App* [19].

Poprzez definiowanie elementów systemu, wprowadzany jest proces dekompozycji, którego wynikiem są podzielone usługi. Przykładowo aplikacja społecznościowa może zostać podzielona na osobne usługi: użytkownicy, wpisy i notyfikacje. Każda z nich może działać niezależnie od pozostałych oraz wykorzystuje inne interfejsy usługowe. Interfejsy muszą spełniać wyznaczone standardy komunikacji, aby istniała możliwość łatwego wiązania ze sobą usług. Usługi są ze sobą luźno powiązane, czyli nie wymagają wiedzy na temat jaka logika odbywa się za interfejsami innych usług.

Organizacja pracy i komunikacji organizacji powinna być prowadzona w taki sposób, aby odzwierciedlała strukturę projektowanego systemu, zgodnie *z prawem Conwaya* [20].

„Każda organizacja, która projektuje system (zdefiniowany szeroko), stworzy projekt, którego struktura jest kopią struktury komunikacyjnej organizacji.”

Melvin E. Conway

Dlatego praktykowane jest podejście, gdzie pracownicy zostają podzieleni na zespoły odpowiedzialne za jeden serwis. Grupy programistów komunikują się wzajemnie w celu wypracowania kontraktów, spełniających założenia biznesowe. Tak jak serwisy, zespoły nie muszą mieć pełnej wiedzy domenowej na temat logiki biznesowej enkapsulowanej przez inną usługę.

W ostatnich latach podejście budowania aplikacji zorientowanych na usługi znacznie zyskało na popularności. Ma to duży związek z rozwojem konteneryzacji (znacznie ułatwia proces tworzenia i zarządzania infrastrukturą) oraz narzędzi do implementacji posiadającej gotowe rozwiązania problemów architektonicznych (np. biblioteki, platformy programistyczne). Zrezygnowano również z Korporacyjnej Magistrali Usług (ang. Enterprise Service Bus), czego skutkiem jest utworzenie nowego terminu – mikrousługi.

Występują rozbieżności w zdefiniowaniu różnic pomiędzy SOA a mikrousługami. Warto zaznaczyć, że mikrousługi nie są niczym rewolucyjnym w odniesieniu do SOA, jest to jedynie modyfikacja podejścia, gdzie wykluczono ESB a wprowadzono drobnoziarniste usługi [21]. Termin mikrousługi został stworzony, aby zaznaczyć zmianę oraz promować podejście jako nowocześniejsze niż SOA (SOA zostało spopularyzowane o wiele wcześniej niż mikrousługi). W niniejszej pracy jest wykorzystywana terminologia związana ściśle z SOA, natomiast sposób realizacji bazuje na mikrousługach.

2.2 Przegląd istniejących rozwiązań

Niniejsza praca przedstawia zdefiniowany kontekst problemu stworzony przez autora, który jest dopasowany do tematyki. Do rozwiązania problemu został wykorzystany wzorzec architektoniczny SOA. Tak jak każdy inny wzorzec SOA posiada grupę cech, które opisują jedynie inne podejście realizacji. Poniżej przedstawiono wybrane wzorce architektoniczne wraz z opisem:

- Wzorzec MVC – podział aplikacji na 3 niezależne komponenty: model (przechowywanie i pobieranie danych), widok (wizualizacja modelu przez zdefiniowany interfejs użytkownika oraz odbieranie komunikatów) i kontroler (warstwa spajająca model i widok, odpowiada za komunikację i zarządza logiką). Jest to jeden z popularniejszych stosowanych wzorców, ze względu na prosty podział odpowiedzialności oraz stosowanie w najpopularniejszych bibliotekach oraz platformach programistycznych (np. Ruby on Rails, Spring, Django). [22]
- Prezentacja-abstrakcja-kontrola (PAC) – inspirowany wzorcem MVC. Zawiera agentów, którzy zawierają w sobie komponenty: prezentacja (widok), abstrakcja (model) i kontrola (kontroler). Agenci tworzą wzajemnie strukturę, która komunikuje się poprzez część kontrolną. [23]

2.3 Przegląd literatury

Do przygotowania pracy dyplomowej autor głównie korzystał z poniższych pozycji z literatury:

- *Mikrousługi: wdrażanie i standaryzacja systemów w organizacji inżynierskiej* [24]
- *Tao mikrousług : projektowanie i wdrażanie* [25]

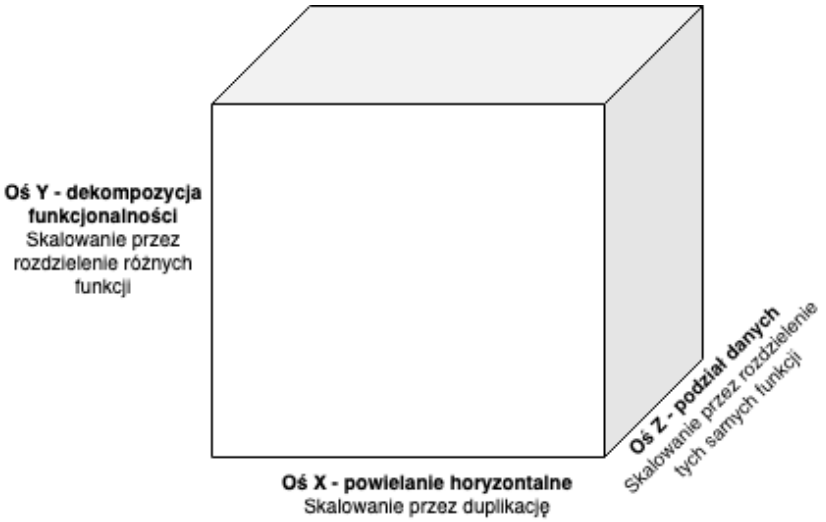
- *Microservices patterns* [26]

2.4 Aplikacje rozproszone

W celu wprowadzenia zintegrowanego środowiska, które zawiera szereg powiązanych komponentów, należy stosować sprawdzone rozwiązania i przestrzegać reguły, które zostały poniżej opisane.

2.4.1 Skalowalność

Ważnym aspektem aplikacji rozproszonych jest skalowalność, czyli właściwość systemu opisująca zachowanie przy zmieniających się zasobach. Termin ten można przedstawić w postaci trójwymiarowego sześcianu (sześcian skali) [27] ze zdefiniowanymi osiami X, Y i Z opisanymi na Rysunek 2.



Rysunek 2 - sześcian skali

Każdą z osi można zdefiniować w następujący sposób:

- Oś X – polega na uruchomianiu kopii tej samej instancji, które są połączone z serwerem równoważącym obciążenie. Redukuje czas obliczeniowy poprzez rozprowadzanie żądań od klientów według ustalonego algorytmu. Dla usług, które nie wykonują skomplikowanej logiki, a jedynie przekazują wykonanie operacji do bazy danych, nie wniesie to znacznego wzrostu wydajności systemu.
- Oś Y – proces podziału funkcjonalności systemu na mniejsze komponenty. Podejście jest stosowane dla aplikacji zorientowanych na usługi, gdzie

wprowadzane są granice pomiędzy funkcjonalnościami systemu. Dla każdej usługi dostarczane są zasoby obliczeniowe, przez co możliwe jest niezależne dostosowywanie wydajność instancji.

- Oś Z – podobny do skalowania X, uruchamia kopię instancji z wyłączeniem części funkcjonalności (najczęściej względem podzbioru danych). Stosowany do skalowania baz danych poprzez dzielenie partycji na zestaw serwerów. Zapytania mogą być przypisane do konkretnej partycji przez przekazywanie klucza głównego podmiotu lub rozpoznanie typu klienta.

2.4.2 Komunikacja

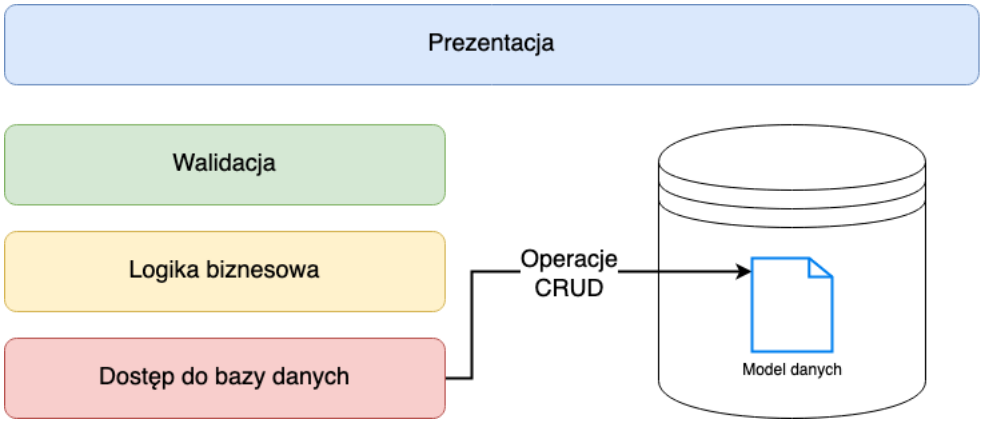
Komunikacja to kluczowy element systemu zorientowanego na usługi, ponieważ wpływa on na skalowalność rozwiązania. Wyróżnia się dwa style komunikacji:

- Synchroniczny – stosuje protokoły komunikacyjne, które wymagają dla każdego zapytania klienta odpowiedzi odbiorcy. Wymaga, aby w trakcie procesu komunikacji zarówno klient jak i odbiorca byli dostępni.
- Asynchroniczny – wymaga zastosowania brokera wiadomości, pośredniczącego pomiędzy klientem a odbiorcą w celu przechowywania komunikatów do dostarczenia. Dzięki temu podejściu klient nie musi oczekiwać na odpowiedź ze strony odbiorcy i może przejść do przetwarzania kolejnych operacji. Wadą tego rozwiązania jest stosowanie jednego punktu awarii, który zatrzymuje działanie systemu (można temu zapobiec przez replikowanie brokera wiadomości, np. w trybie master/slave). Może realizować style komunikacji asynchronicznej publikowanie/subskrypcja i kolejkowanie oraz synchronicznej żądanie odpowiedź.

2.5 Zastosowane wzorce architektoniczne

2.5.1 CQRS

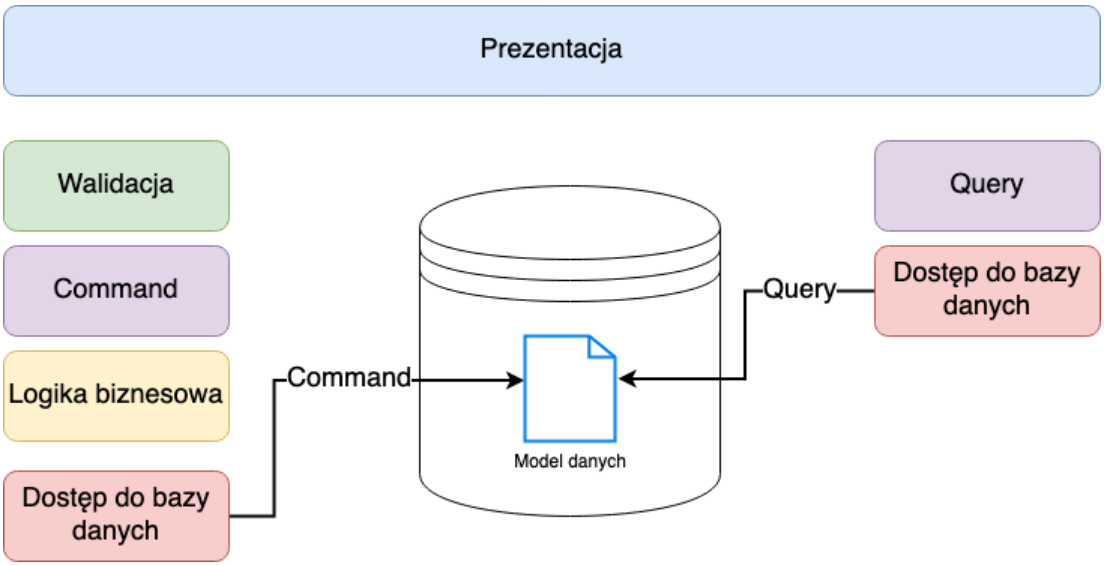
Powszechnie do tworzenia aplikacji z warstwą dostępu do bazy danych stosowane jest podejście tworzenia metod CRUD. CRUD zapewnia dostęp do trzech operacji modyfikujących stan danych (C – create, U – update, D - delete) oraz jednej odczytu (R - read).



Rysunek 3 - Wzorzec CRUD

Rysunek 3 przedstawia przepływ danych z wykorzystaniem wzorca CRUD z uwzględnieniem warstwy walidacji, logiki biznesowej oraz dostępu do bazy danych.

Nie zawsze jest to optymalne podejście, a w szczególności, gdy narzucone metody nie odzwierciedlają rzeczywistego zamiaru. Za nazwą metody może się znajdować skomplikowana logika. Kolejną wadą rozwiązania jest wspólny model dla odczytu i zapisu. Przykładowo model odczytu może mieć różne struktury, co jest rozwiązywane przez mapowanie obiektów na DTO (Data Transfer Object [28]). Z czasem mapowanie obiektów może stać się trudne do modyfikacji oraz kosztowne przy operacjach łączenia tabel. To samo może stać się z logiką biznesową dla modelu odczytu, w wyniku podejmowanych kompromisów w budowaniu wspólnego schematu. W zależności od charakterystyki aplikacji dochodzi do asymetrycznego obciążenia operacji odczytu i zapisu. Jeżeli stosowane jest homogeniczne podejście budowania warstwy dostępu do bazy danych, nie można skalować wydajności systemu na podstawie obciążenia operacji. Przykładowo istnieje duże prawdopodobieństwo, że portal z ogłoszeniami będzie miał większe zapotrzebowanie na odczyt niż zapis danych, co implikuje w potrzebę skalowania konkretnych komponentów systemu. Ostatnim problemem jest zarządzanie bezpieczeństwem dostępu do danych. Wspólny model nie daje możliwości tworzenia osobnych reguł bezpieczeństwa.



Rysunek 4 - Wzorzec CQS

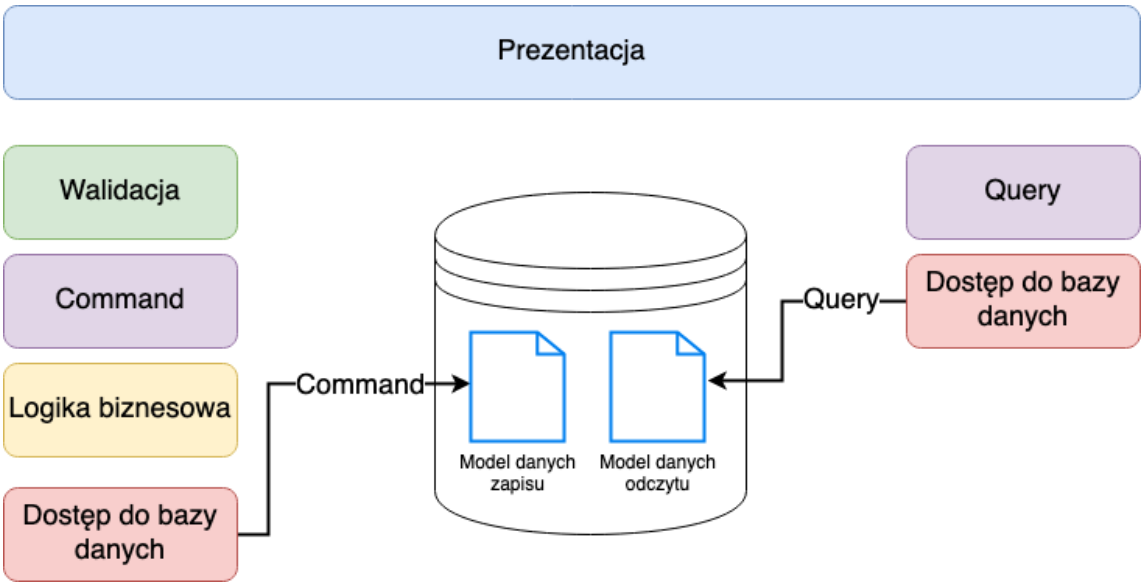
Jako alternatywę wprowadzono wzorzec CQS (Command Query Separation). Rysunek 4 przedstawia ogólny przepływ danych dla tego wzorca.

Charakteryzuje się on podziałem metod dostępu na dwie odpowiedzialności:

- Command (Polecenie) – modyfikacja stanu bazy danych.
- Query (Zapytanie) – odczyt bazy danych.

CQS wprowadza jasną informację, za co odpowiedzialna jest dana metoda. Nie rozwiązuje jednak problemu ze wspólnym modelem danych.

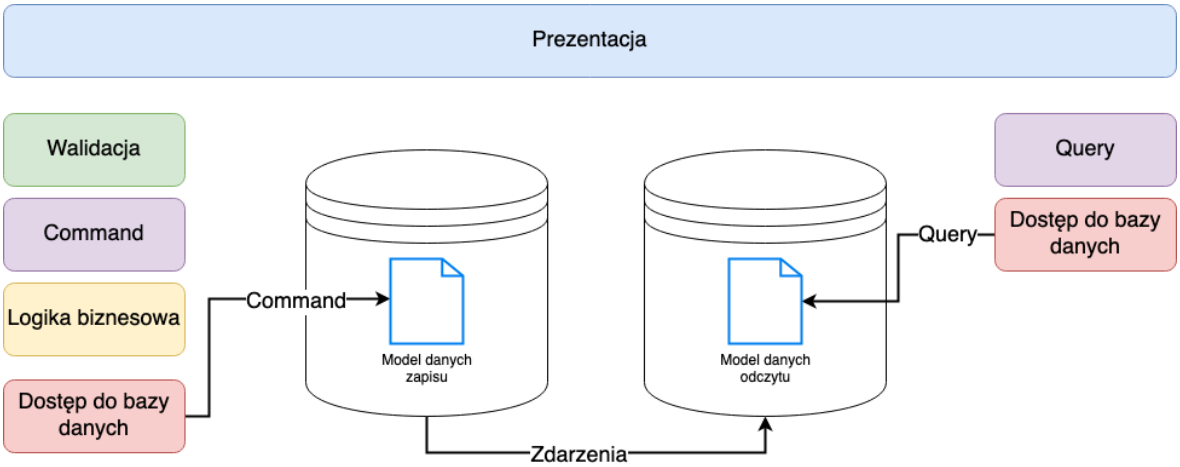
W podziale odpowiedzialności dokonano kolejnego kroku, wprowadzając wzorzec CQRS (Command and Query Responsibility Segregation).



Rysunek 5 - Wzorzec CQRS

Zgodnie z tym co jest przedstawione na Rysunek 5, CQRS rozdziela modele danych na osobne schematy, co daje następujące korzyści:

- Skalowanie - dzięki temu podejściu można tworzyć osobne instancje aplikacji, które odzwierciedlają podział CQRS. Wtedy możliwe jest skalowanie w osi X względem zapotrzebowania na operację odczytu czy zapisu.
- Optymalizacja zapytań bazy danych – często stosowaną techniką jest tworzenie w bazie danych zmaterializowanego widoku, na podstawie rzeczywistych danych. Ogranicza to powstawanie złożonych połączeń dla zapytań, które korzystają ze zdefiniowanego schematu. Innym rozwiązaniem jest całkowita separacja bazy danych na osobne instancje. Wtedy źródłem rzeczywistych danych jest baza danych dla operacji zapisu, a baza danych operacji odczytu jest traktowana jako pamięć podręczna, która jest ostatecznie spójna. Baza danych przeznaczona do odczytu musi być aktualizowana według przyjętej strategii. Jednym z rozwiązań jest tworzenie zdarzeń przez bazę danych zapisu, które następnie wpływają na bazę danych odczytu, aktualizując jej stan, co opisuje Rysunek 6. Zdarzenia zawierają informacje, które są następnie denormalizowane [29] w celu dostosowania do przyjętego schematu.



Rysunek 6 - Wzorzec CQRS dla separowanych baz danych

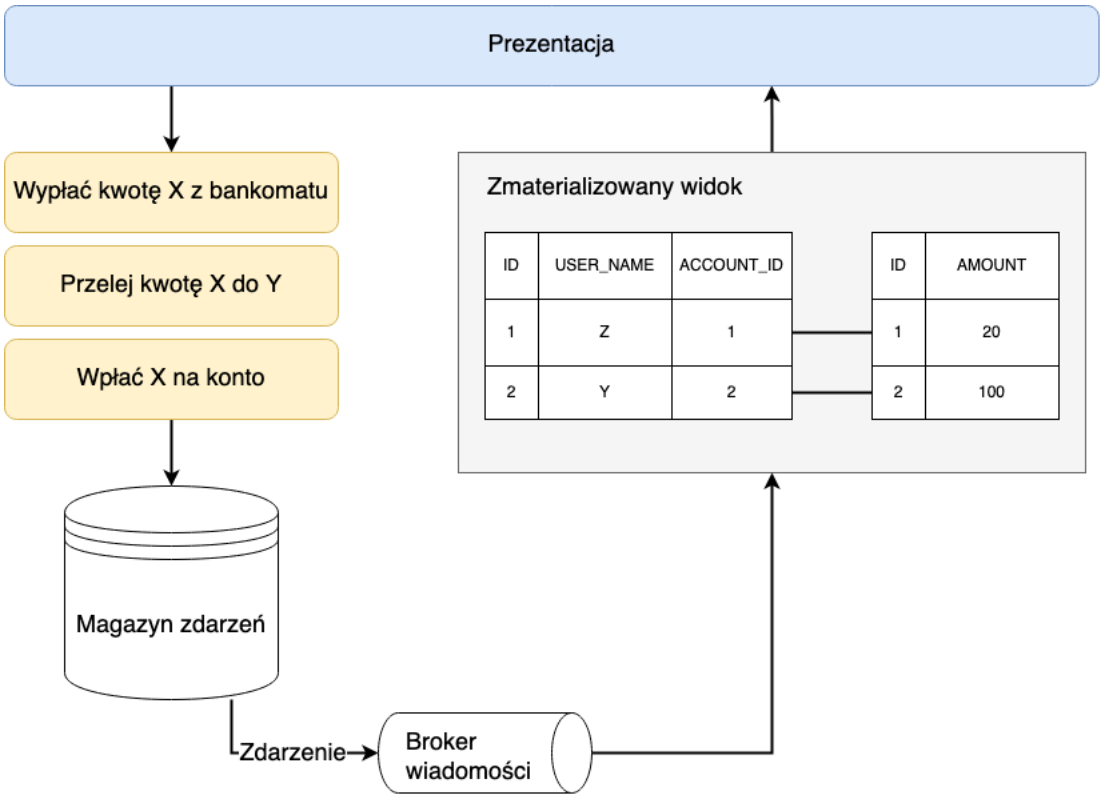
- Bezpieczeństwo danych – wprowadzenie separacji modeli danych daje możliwość elastycznego zarządzania dostępem do danych. Można definiować reguły dostępowe, które są przypisane do konkretnych jednostek.

- Dekompozycja problemów – wzorzec CQRS upraszcza rozwój aplikacji, oddzielając logikę związaną z zapisem od odczytu.

Wzorzec CQRS wymaga zaimplementowania mechanizmu reakcji na operacje zapisu, który może być złożony. System powinien być zaprojektowany z myślą o reakcji na komunikaty. Należy wdrożyć rozwiązanie, które będzie dystrybuować komunikaty pomiędzy komponentami. Kolejnym problemem jest ostateczna spójność danych. Wzorzec nie jest zalecany dla funkcjonalności, która musi podawać rzeczywiste dane zgodne ze stanem bazy danych. Dodatkowo, jeżeli logika biznesowa jest łatwa do zarządzania, zalecanym rozwiązaniem jest wykorzystanie wzorca CRUD.

2.5.2 Wzorzec określania źródła zdarzeń (Event Sourcing)

Dla powszechnie stosowanego podejścia CRUD baza danych przedstawia jedynie obecny stan. Przykładowo zmiana stanu rekordu w bazie danych sprawi, że nie można określić jaki był poprzedni stan obiektu. Do rozwiązania tego problemu stosowany jest wzorzec określania źródła zdarzeń w połączeniu z CQRS.



Rysunek 7- Event Sourcing

Event sourcing wprowadza do wzorca CQRS dziennik zdarzeń. Baza danych zamiast przechowywać rzeczywisty stan danych zawiera szereg zdarzeń z informacjami o modyfikacjach obiektów. To rozwiązanie daje następujące korzyści:

- Możliwość odtworzenia stanu systemu z dowolnego punktu w czasie. Jest to cenna funkcjonalność dla systemów, które muszą monitorować zachowania użytkowników z możliwością dokładnego określenia rzeczywistych danych.
- Zwiększenie szybkości zapisu danych. Kolejne modyfikacje nie blokują dostępu do bazy danych a są kolejkowane. Możliwe jest asynchroniczne wykonywanie zapisu w tle, bez oczekiwania na zakończenie przetwarzania.
- Analiza zdarzeń. Zdarzenia mogą być wykorzystane do wskazania trendów zachowania użytkowników.
- Akcje kompensacyjne, które wycofują zmiany wprowadzone przez zdarzenie, poprzez odwołanie się do modyfikacji z przeszłości.

  Korzystanie z tego wzorca wymaga również rozwiązania następujących problemów:

  - W celu uzyskania zmaterializowanego widoku danych należy wprowadzić mechanizm projekcji. Projekcje dla systemu z dużą ilością danych są czasochłonne, stąd trzeba ograniczać ich częstotliwość. Można to rozwiązać poprzez wykonywanie projekcji systematycznie co określoną liczbę zdarzeń.
  - Minimalizacja ilości przechowywanych danych. W przeciwieństwie do tradycyjnego podejścia, gdzie przechowywany jest jedynie rzeczywisty stan, Event sourcing składuje znacznie więcej informacji. Generuje to większe zapotrzebowanie zasobów dla bazy danych, które może być redukowane poprzez ograniczenie informacji w zdarzeniu.
  - Modyfikacja schematu. Nie istnieje możliwość migracji schematu bazy do nowej wersji. Jedynym rozwiązaniem jest modyfikacja mechanizmu denormalizacji danych.
  - Usunięcie informacji o zdarzeniach generowanych przez użytkownika jest trudne lub niemożliwe do realizacji. Przykładowo użytkownik chce, żeby system nie przechowywał żadnych powiązanych z nim informacji zgodnie z rozporządzeniem o ochronie danych. Taka modyfikacja musiałaby zmieniać dane każdego zdarzenia w systemie, w celu zachowania spójności. Łamie to

reguły korzystania ze wzorca – usunięcie danych powinno być realizowane jedynie poprzez dodanie kolejnego zdarzenia.

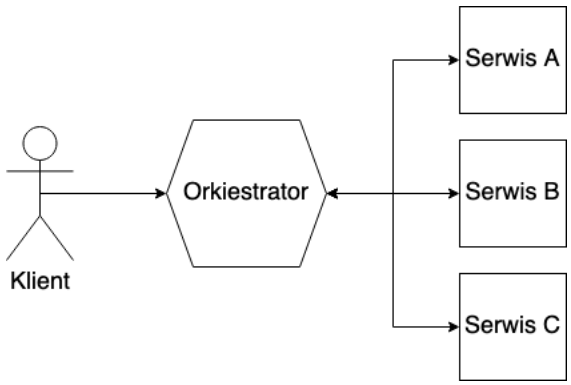
Event Sourcing jest stosowany w przypadkach, gdzie wymagany jest dziennik zdarzeń. Przykładowo znajduje swoje zastosowanie w systemach bankowych, który informuje użytkowników o aktualnym stanie konta oraz musi dokonywać jego modyfikacji. W razie awarii lub niepowodzenia operacji, system jest w stanie przywrócić stan konta do dowolnego punktu w czasie z przeszłości. Dodatkowo spełnia wszystkie warunki, żeby przedstawiać dane statystyczne, które mogą zostać przedstawione użytkownikowi lub poddane analizie przez pracowników firmy. Rysunek 7 opisuje przykład, gdzie dane z magazynu zdarzeń generują komunikaty, których zawartość wpływa na stan zmaterializowanego widoku użytkowników banku.

2.5.3 Saga

Wzorzec projektowy, który wprowadza zarządzanie transakcjami systemu rozproszonego, zawierającego szereg transakcji lokalnych. Każda zrealizowana transakcja lokalna publikuje zdarzenie, które następnie wywołuje kolejną operację. Zdarzenie może rozpocząć następną transakcję lokalną lub sekwencję akcji kompensacyjnych w celu przywrócenia stanu sprzed rozpoczęcia Saga.

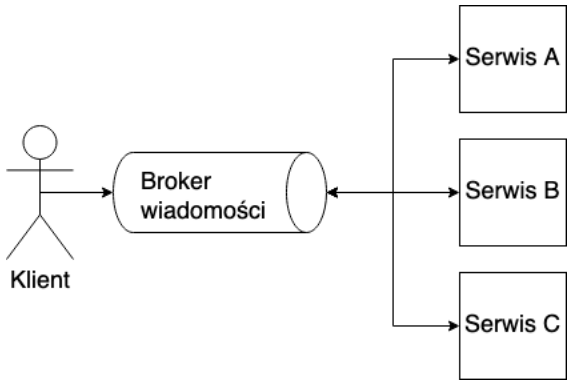
Można wyróżnić dwa style korzystania z Saga:

- Aranżacja – koordynacja zdarzeń odbywa się w scentralizowanym punkcie. Orkiestrator odpowiada za nadzór i obsługę wszystkich operacji, które mają nastąpić w ramach jednej transakcji. Wprowadza znaczne uproszczenie dla usług biorących udział. Usługi nie muszą wiedzieć jaka logika jest zawarta w orkiestratorze. Wadą jest wprowadzenie pojedynczego punktu awarii.



Rysunek 8 - Saga aranżacja

- Choreografia – brak scentralizowanej obsługi zdarzeń. Uczestnik Saga po przetworzeniu lokalnej transakcji publikuje komunikat, który jest konsumowany przez innego uczestnika. Każdy z uczestników musi wiedzieć jaką operacje wykonać dla zdarzenia oraz jakie zdarzenia przekazać do reszty. Główną zaletą tego rozwiązania jest brak wąskiego gardła systemu w postaci orkiestratora, co sprawdza się dla prostych przepływów pracy. Natomiast korzystanie z tego rozwiązania może być trudne do zarządzania przy zmianach organizacji pracy.



Rysunek 9 - Saga choreografia

2.5.4 Brama interfejsu

Wzorzec, który jest stosowany w celu wprowadzenia zunifikowanego interfejsu dostępu do aplikacji. Stosowany w aplikacjach rozproszonych, aby zapobiec sytuacji, gdzie klient musi komunikować się z interfejsem każdej z usług osobno. Poza przekazaniem zapytania klienta do odpowiedniej usługi może spełniać dodatkowe funkcjonalności. Brama komunikuje się z klientem z zastosowaniem protokołu SSL, a następnie przekazuje żądania do usług z zabezpieczonej sieci wewnętrznej po nieszyfrowanym protokole. Optymalizuje to czas potrzebny na obsługę żądania. Stosowanie bramy pomoże w łatwym dodaniu mechanizmu przechowywania odpowiedzi dla klienta w pamięci podręcznej. Wadą bramy interfejsu jest pojedynczy punkt awarii, który można rozwiązać przez skalowanie.

3. Opis problemu

3.1 Ogólny opis

W ramach pracy zostanie przedstawiona przykładowa realizacja aplikacji do głosowania, wykorzystująca opisane techniki oraz spełniająca założone wymagania. System powinien być w stanie stworzyć ankiety do głosowania, które będą dostępne do wypełnienia przez zaproszonych użytkowników.

3.2 Wymagania niefunkcjonalne

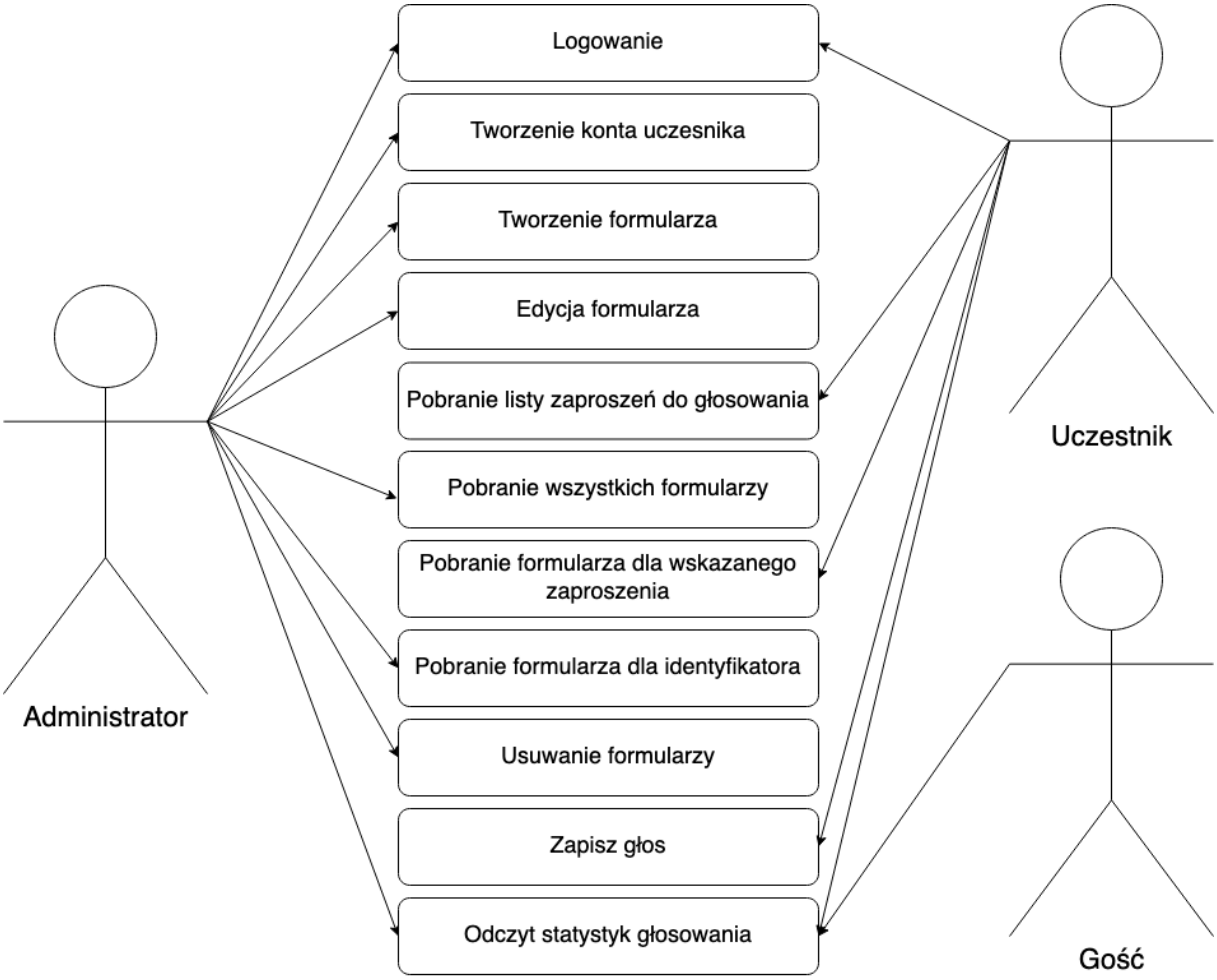
Wymagania niefunkcjonalne systemu:

- Wdrożenie za pomocą obrazów Docker na platformę Docker Compose.
- Skalowanie komponentów aplikacji.
- Prowadzenie dziennika zdarzeń dla głosów.
- Ograniczenie dostępu na podstawie roli użytkownika (opisane przez Rysunek 10).

3.3 Wymagania funkcjonalne

Wymagania funkcjonalne systemu:

- Logowanie.
- Tworzenie konta uczestnika.
- Tworzenie ankiety.
- Edycja ankiety.
- Pobranie ankiety dla zaproszenia.
- Pobranie ankiety dla identyfikatora.
- Pobranie wszystkich ankiet.
- Usuwanie ankiety.
- Pobranie listy zaproszeń do głosowania.
- Zapis głosu.
- Odczyt statystyk głosowania.
- Głosowanie jest dostępne tylko dla opublikowanych ankiet.
- Uczestnik ma dostęp do ankiety poprzez zaproszenie.
- Po odpowiedzi nie można ponownie zagłosować.



Rysunek 10 - Diagram przypadków użycia dla ról dostępnych w systemie

4. Realizacja pracy

4.1 Stos technologiczny

Do realizacji implementacji projektu wykorzystano następujący stos technologiczny.

4.1.1 TypeScript

Nadzbiór języka JavaScript stworzony przez Microsoft, który ukazał się po raz pierwszy w 2012 roku. Jego główną cechą jest wprowadzenie silnego typizowania do JavaScript, co zyskało uznanie wśród programistów i aktualnie jest niepodważalnie liderem wśród rozwiązań tego typu [30]. Silne typizowanie umożliwia walidowanie typów, co znacznie przyśpiesza pracę przy rozbudowanych aplikacjach. W celu uruchomienia programu napisanego w TypeScript na środowisku NodeJS [31] należy najpierw transpilować [32] kod źródłowy do JavaScript za pomocą TypeScript Checker lub można skorzystać z kompilatora Babel.

4.1.2 NestJS

Framework, który umożliwia tworzenie skalowalnych oraz łatwych w utrzymaniu aplikacji sieciowych. Platforma korzysta z funkcjonalności środowiska NodeJS. Istnieje możliwość stosowania paradygmatów: programowania obiektowego (OOP), funkcyjnego (FOP) oraz reaktywnego (FRP). Enkapsuluje w sobie popularny framework ExpressJS i jeżeli zajdzie potrzeba, zapewnia bezpośrednio dostęp do jego funkcjonalności. Dzięki czemu programista może elastycznie modyfikować domyślne działanie narzucone przez platformę programistyczną. Za pomocą narzędzi dostarczanych przez NestJS można zaimplementować aplikacje oparte na mikrousługach, stosować wzorce takie jak CQRS, Event Sourcing i Saga oraz zapewniać komunikację pomiędzy usługami za pomocą protokołu TCP, brokera wiadomości lub gRPC. Domyślnie wspieranym językiem jest TypeScript, co umożliwia wprowadzanie klas DTO oraz encji z określonymi typami.

4.1.3 Redis (Remote Dictionary Server)

Nierelacyjna baza danych NoSQL, która magazynuje dane w przestrzeni pamięci RAM. Umożliwa obsłużenie stu tysięcy zapytań w jednej sekundzie, co jest przydatne dla części aplikacji, które wymagają szybkiego odczytu danych. Wspiera wiele struktur: ciągi, hasze, listy, zestawy, sortowane zestawy z zapytaniami o zakres, bitmapy, hiperlogi, indeksy geoprzestrzenne i strumienie [33].

Redis jest wysoko dostępny i skalowalny. Dane mogę by przechowywane na wielu serwerach replikacyjnych, co umożliwia zwiększenia wydajności dla zapytań obsługiwanych na wielu instancjach. W momencie, gdy jeden z serwerów ulegnie awarii, dane mogą zostać odzyskane z kopii zapasowej. Redis znajduje swoje zastosowanie między innymi jako magazyn pamięci podręcznej. Dodatkowo umożliwia określenie parametru TTL (czas ważności dokumentu) dla konkretnych danych przechowywanych pod zdefiniowanym kluczem.

4.1.4 PostgreSQL

System RDBMS (Relational Database Management System [34]) rozwijany od 1986 przez społeczność otwartego oprogramowania. Jeden z najpopularniejszych silników relacyjnych baz danych [35] przygotowany do pracy w środowisku produkcyjnym oraz dla środowisk deweloperskich. PostgreSQL jest zgodny ze standardem SQL, dzięki czemu stosując podstawową funkcjonalność istnieje możliwość zmiany silnika bazy bez skomplikowanych migracji danych [36].

4.1.5 TypeORM

Narzędzie ORM gotowe do pracy w środowisku NodeJS z bazami danych: MySQL, MariaDB, Postgres, CockroachDB, SQLite, Microsoft SQL Server, Oracle, SAP Hana i sql.js. Umożliwia korzystanie z dwóch następujących wzorców architektonicznych dotyczących dostępu do danych składowanych w bazie danych [37]:

- Aktywny rekord (Active record) – cała logika dostępu do bazy danych jest enkapsulowana razem z obiektem encji. Interfejs dostępu jest dostarczany przez platformę programistyczną, co znacznie ogranicza ilość kodu potrzebnego do wprowadzenia podstawowych funkcjonalności CRUD. Jest uznawany za antywzorzec, ze względu na łamanie SRP (Single Responsibility Principle), nie powinien być stosowany dla rozwiązań z rozbudowaną logiką biznesową.
- Mapowania danych (Mapowanie danych) – wprowadza warstwę pośrednią, pomiędzy warstwą domenową aplikacji, a warstwą składowania danych, która jest odpowiedzialna za dwukierunkowy transfer danych, za pomocą funkcji mapujących.

4.1.6 Docker oraz Docker Compose

Docker jest to otwarta platforma służąca do zarządzania „lekkimi maszynami wirtualnymi" - kontenerami. Kontenery są to odizolowane środowiska, na których można uruchomić przygotowaną instancję aplikacji. Docker umożliwia elastyczne zarządzanie infrastrukturą:

- Skalowanie – umożliwia replikowanie pojedynczych instancji kontenerów, które mogą być zarządzane ręcznie lub przez orkiestrator.
- Definiowanie sieci kontenerów – wprowadzenie własnej topologii sieci, która może być izolowana od dostępu zewnętrznego.
- Globalna baza obrazów kontenerów – możliwość korzystania z gotowych kontenerów, przygotowanych przez społeczność oraz dostawców oprogramowania. Ułatwia i przyspiesza proces wdrożenia narzędzia, dzięki zewnętrznej konfiguracji z poziomu Docker.

Środowisko może być przenoszone zarówno na inne maszyny lokalne oraz rozwiązanie chmurowe.

4.1.7 RabbitMQ

Broker wiadomości zgodny ze standardem AMQP, otwarty protokół oprogramowania pośredniczącego, zorientowany na niezawodne trasowanie komunikatów. Pozwala na asynchroniczne przekazywanie komunikatów pomiędzy oddzielnymi komponentami systemu, które muszą jedynie posiadać bibliotekę kliencką, spełniającą jeden z wielu wspieranych protokołów [38, 39]. Rozwiązanie jest stosowane w aplikacjach zorientowanych na usługach ze względu na następujące cechy:

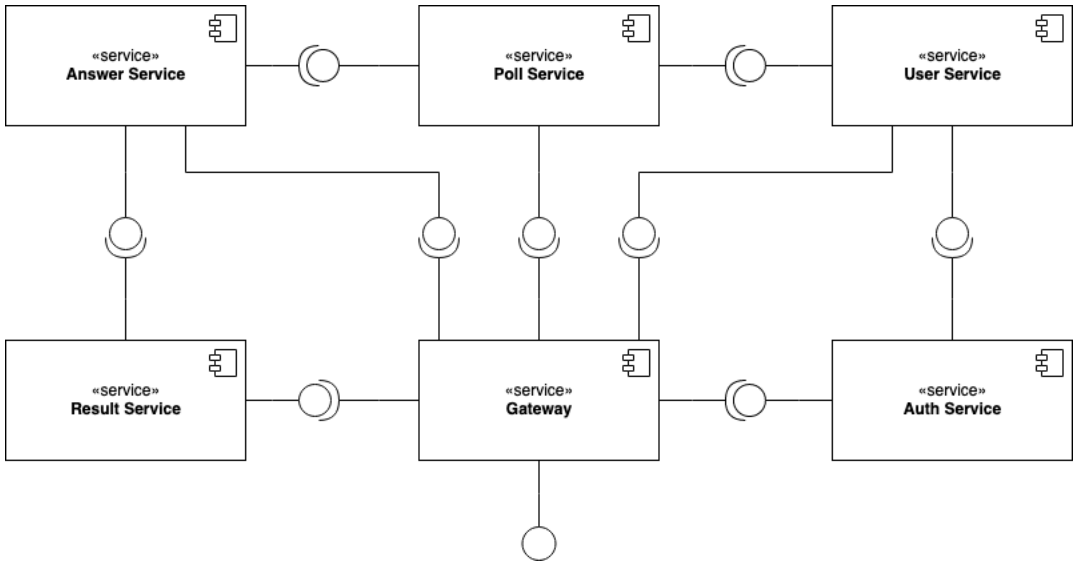
- Wsparcie dla wielu klientów, dla których każdy z komunikatów jest po równo dystrybułowany dla domyślnie ustawionego algorytmu rotacyjnego.
- Możliwość wdrożenia klastra w celu zwiększenia wydajności obsługi wielu komunikatów i połączeń.
- Oprogramowanie jest domyślnie dostarczane z graficznym interfejsem użytkownika, co znacznie ułatwia i przyspiesza diagnostykę przesyłanych komunikatów.

4.1.8 NGINX

Serwer internetowy, który może pełnić również funkcjonalność dystrybutora żądań. Jest to stabilne, modułowe i proste w użyciu narzędzie, które aktualnie jest najpopularniejszym rozwiązaniem. Umożliwia stosowanie statycznych i dynamicznych algorytmów równoważenia obciążenia, co można wykorzystać na przykład przy komponencie systemu, który wymaga pracy na wielu kopiach. Dzięki temu następuje redukcja opóźnień, zwiększenie odporności na błędy oraz zwiększenie przepustowości. [40]

4.2 Struktura systemu

4.2.1 Diagram komponentów



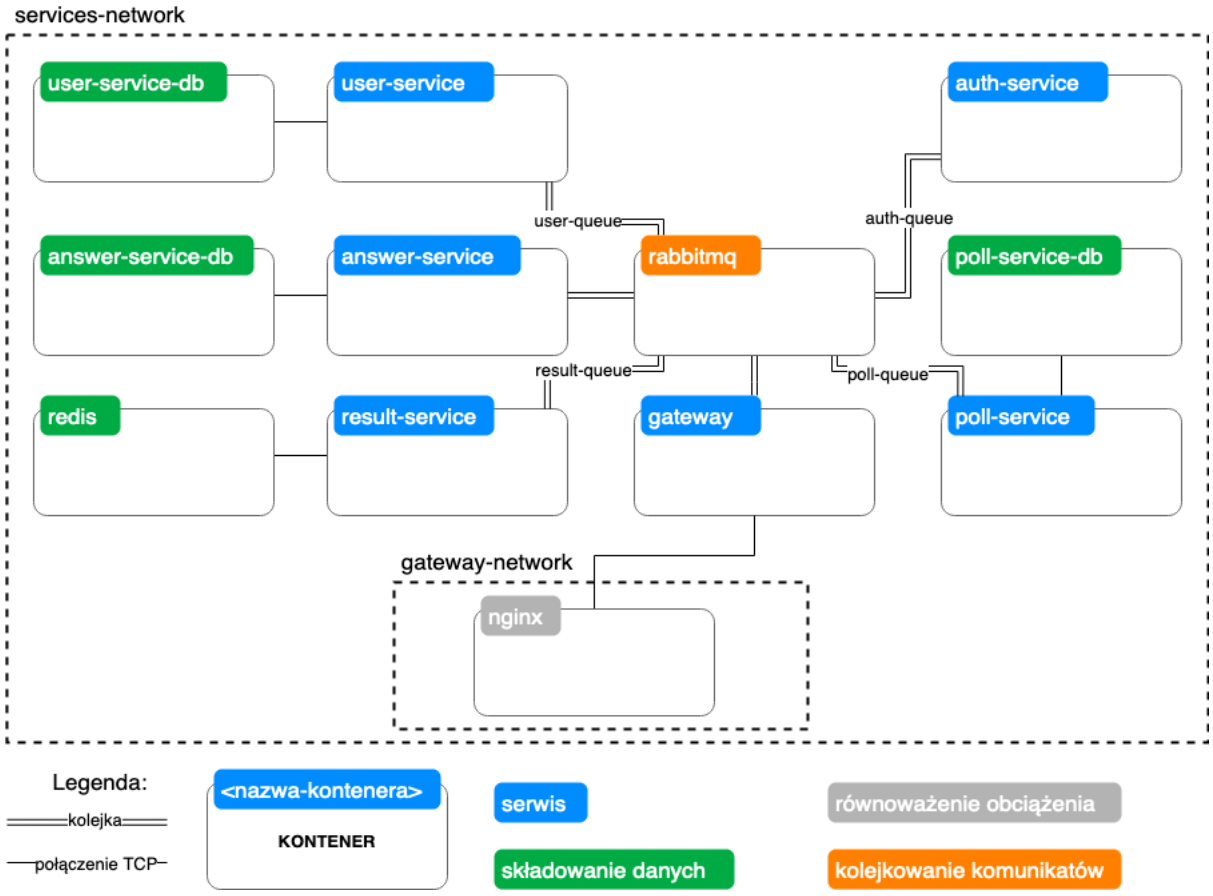
Rysunek 11 - Diagram komponentów systemu

Rysunek 11 przedstawia diagram komponentów. Każdy z serwisów posiada wyodrębnioną logikę biznesową, która nie pokrywa się z resztą usług. Poniżej jest przedstawiona lista funkcjonalności realizowanych przez każdą z usług:

- Gateway – stworzony w celu realizacji wzorca API Gateway [41], który udostępnia pojedynczy punkt dostępu do zbioru drobnoziarnistych interfejsów usług. Dodatkowo określa dla każdego kontrolera obsługującego zapytanie poziom dostępu na podstawie roli użytkownika.
- Auth Service – odpowiada za uwierzytelnienie oraz autoryzację użytkownika do systemu.

- User Service – zapewnia dostęp do zarządzania użytkownikami w systemie.
- Poll Service – realizuje operacje CRUD dla ankiet oraz walidację głosu dla ankiety powiązanej z użytkownikiem przez zaproszenie.
- Answer service – odpowiada za zapis głosu. Wykonuje operacje Command, bazując na wzorcu CQRS. Przekazuje głos do zapisu w bazie przechowującej zdarzenia, wprowadza ochronę przed awarią zapisu zdarzenia oraz wykonuje projekcję bazy danych.
- Result service – odpowiada za odczyt statystyk głosowania. Wykonuje operacje Query, bazując na wzorcu CQRS. Reaguje na zdarzenia, które przetwarza do postaci zdenormalizowanej, a następnie zapisuje w bazie danych. Umożliwia również ręczne stworzenie zapytania o wszystkie zdarzenia dla wskazanej ankiety.

4.2.2 Diagram infrastruktury



Rysunek 12 - Diagram infrastruktury systemu

Powyższy Rysunek 12 przedstawia wszystkie kontenery wykorzystane do implementacji systemu do ankiet.

Na diagramie są zaznaczone kolejki komunikatów obsługiwane przez RabbitMQ razem z ich nazwami. Większość przypadków wymaga zastosowania komunikacji dwukierunkowej pomiędzy serwisami, dlatego zastosowano rozwiązanie „Direct Reply-to" [42]. Dla klienta tworzona jest osobna kolejka z nagłówkiem „reply-to", która będzie zwracała komunikaty odbiorcy.

4.2.3 Lista komunikatów

W celu zobrazowania funkcjonowania systemu stworzono Tabela 1, która zawiera pełną listę komunikatów. Komunikaty są budowane według następującej konwencji:

- *role* – usługa obsługująca komunikat.
- *cmd* – identyfikator komunikatu.

Tabela 1 posada następujące kolumny:

- *Nazwa usługi* – identyfikuje usługę, która przetwarza komunikat.
- *Komunikat przychodzący* – komunikat, który rozpoczyna wykonywanie operacji w usłudze.
- *Komunikat wychodzący* – komunikat, który jest przekazywany do brokera wiadomości po zakończeniu przetwarzania.
- *Czy wymagana jest odpowiedź?* – określa czy zakończenie operacji wymaga poinformowania innych komponentów systemu o rezultacie.

Dla zwiększenia czytelności pominięto komunikaty przetwarzane przez Gateway, ponieważ jego jedyna odpowiedzialność to przekazywanie przetwarzania do wskazanej usługi.

Tabela 1 - komunikaty systemu

Nazwa usługi	Komunikat przychodzący	Komunikat wychodzący	Czy wymagana jest odpowiedź?
Poll Service	`role: 'poll', cmd: 'get-polls'`	-	Tak
Poll Service	`role: 'poll', cmd: 'create-poll'`	-	Tak
Poll Service	`role: 'poll', cmd: 'validate-answer'`	-	Tak
Poll Service	`role: 'poll', cmd: 'get-poll'`	-	Tak
Poll Service	`role: 'poll', cmd: 'update-poll'`	-	Tak
Poll Service	`role: 'poll', cmd: 'delete-poll'`	-	Nie
Poll Service	`role: 'poll', cmd: 'check-invitations'`	-	Tak
Poll Service	`role: 'poll', cmd: 'get-invitation-poll'`	-	Tak
Poll Service	`role: 'poll', cmd: 'restore-invitation'`	-	Nie
Answer Service	`role: 'answer', cmd: 'send-answer'`	1. `role: 'poll', cmd: 'validate-answer'` 2. `role: 'poll', cmd: 'restore-invitation'` 3. `role: 'result', cmd: 'result-answer'`	Nie
Answer Service	`role: 'answer', cmd: 'answer-poll-projection'`	-	Tak
Result Service	`role: 'result', cmd: 'get-results'`	-	Tak
Result Service	`role: 'result', cmd: 'result-answer'`	-	Nie
Result Service	`role: 'result', cmd: 'poll-projection'`	`role: 'answer', cmd: 'answer-poll-projection'`	Tak
Auth Service	`role: 'user', cmd: 'auth'`	`role: 'user', cmd: 'get'`	Tak
Auth Service	`role: 'user', cmd: 'check'`	`role: 'user', cmd: 'get'`	Tak
User Service	`role: 'user', cmd: 'get'`	-	Tak
User Service	`role: 'user', cmd: 'add'`	-	Tak

4.2.4 Wdrożenie

Infrastruktura jest wdrożona za pomocą kontenerów Docker zarządzanych przez Docker Compose. Technologia ta umożliwia deklaratywną konfigurację kontenerów za pomocą pliku zapisanego w notacji YAML lub JSON.

```
version: '3'
services:
  nginx:
    image: nginx
    volumes:
      - ./nginx:/etc/nginx/templates
    ports:
      - ${NGINX_PORT_IN}:${NGINX_PORT_OUT}
    networks:
      - services-network
      - gateway-network
    deploy:
      resources:
        limits:
          cpus: '0.1'
          memory: 256M
  redis:
[…]
  rabbitmq:
[…]
  gateway:
[…]
  #MICROSERVICES
  user-service:
    build: ./user-service
    restart: always
    hostname: user-service
    env_file:
      - .env
    networks:
      - services-network
    deploy:
      resources:
        limits:
          cpus: '0.1'
          memory: 256M
    depends_on:
      - rabbitmq
  auth-service:
[…]
  poll-service:
[…]
  answer-service:
[…]
  result-service:
[…]
#DATABASES
  user-service-db:
[…]
  poll-service-db:
[…]
  answer-service-db:
[…]
networks:
```

```
services-network:
  internal: true
gateway-network:
  driver: bridge
```

Listing 1 - plik konfiguracyjny docker-compose.yml

Konfiguracja systemu jest przedstawiona w Listing 1, gdzie zawarte są następujące informacje:

- Nazwa hosta - zamiast korzystać z przypisanego do kontenera adresu IP, poprawniej jest się odnosić do niezmiennej nazwy hosta, która jest przechowywana w wewnętrznym serwerze DNS dostarczanym przez Docker.
- Nazwa obrazu Docker. Dla serwisów występuje odwołanie do dokumentu Dockerfile, w których opisano instrukcje do stworzenia środowiska uruchomieniowego dla aplikacji NestJS. Listing 2 przedstawia kroki budowania kontenera. To podejście jest stosowane, gdy istnieje potrzeba zmodyfikowania innego obrazu. W poniższym przykładzie wprowadzono zmiany dla obrazu zawierającego środowisko NodeJS.

```
FROM node:15.1.0-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
RUN ls
CMD ["node", "dist/main"]
```

Listing 2 - plik Dockerfile dla auth-service

- Zmienne środowiskowe współdzielone przez wszystkie kontenery, które zapewniają jeden punkt konfiguracji systemu i ukrywają wrażliwe informacje np. dane logowania do bazy danych.
- Sieci do których należy każdy z kontenerów. Sieć wewnętrzna *services-network* jest odizolowana od dostępu zewnętrznego w celu zabezpieczenia wrażliwych informacji przekazywanych pomiędzy usługami. Sieć zewnętrzna *gateway-network* obsługuje zapytania pochodzące od klienta, które trafiają do kontenera zawierającego serwer *ngnix*. Następnie za pomocą wbudowanego narzędzia równoważenia obciążenia, zapytania są przekazywane według zdefiniowanych w pliku konfiguracyjnym reguł.

```
upstream loadbalancer {
server gateway:8081 weight=5;
server gateway:8082 weight=5;
server gateway:8083 weight=5;
server gateway:8084 weight=5;
server gateway:8085 weight=5;
server gateway:8086 weight=5;
server gateway:8087 weight=5;
server gateway:8088 weight=5;
server gateway:8089 weight=5;
}
server {
listen 8080;
location / {
proxy_pass http://loadbalancer;
}}
```

Listing 3 - Konfiguracja serwera nginx

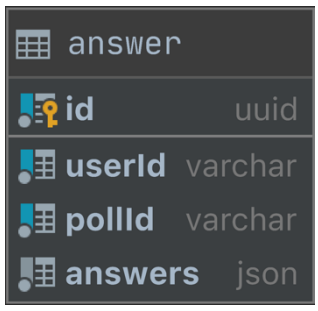
Listing 3 zawiera konfigurację serwera *nginx*, który przekazuje zapytania do wskazanych hostów, stosując algorytm rotacyjny według ustalonych wag.

- Zasoby dla kontenerów. Zostały wprowadzone, w celu zapewnienia jednakowego dostępu do jednostki CPU oraz pamięci RAM dla każdego serwisu – pozwoli to uzyskać dokładniejsze wyniki skalowania podczas badań.
- Woluminy tworzone i zarządzane przez Docker.

4.3 Struktura danych

Każdy serwis posiada własną przestrzeń składowania danych, co jest zgodne z wzorcem „Baza danych dla każdego serwisu” [43]. Poniżej przedstawiono strukturę przechowywania danych dla każdej bazy danych:

- Answer Service – schemat bazy danych



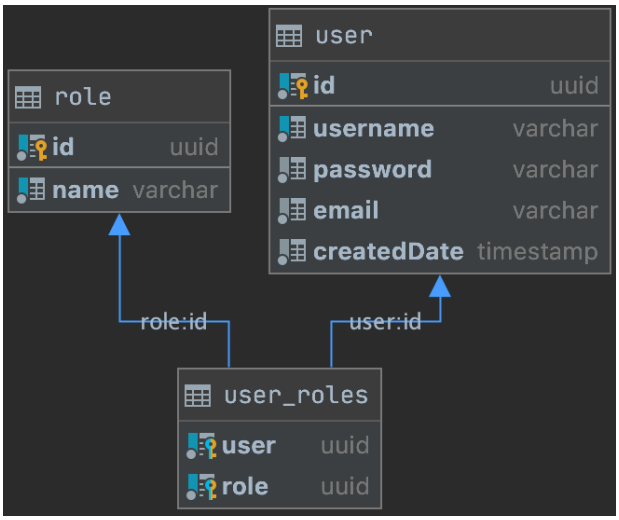
Rysunek 13 - Schamat bazy danych dla Answer Service

Opis dla Rysunek 13:

- id – identyfikator odpowiedzi
- userId – identyfikator użytkownika, który jest autorem odpowiedzi
- pollId – identyfikator ankiety

- answers – obiekt zawierający odpowiedzi na pytania

- User Service – schemat bazy danych



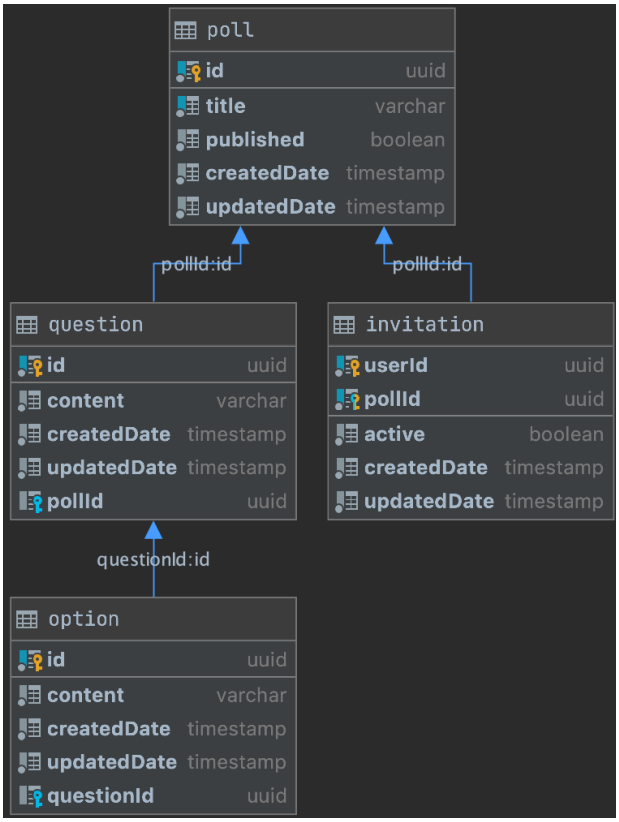
Rysunek 14 - Schemat bazy danych dla User Service

Opis dla Rysunek 14:

- id – identyfikator użytkownika/roli
- name – nazwa roli
- username – nazwa użytkownika
- password – hasło użytkownika do logowania
- email – adres email użytkownika
- createdDate – moment stworzenia użytkownika

Zastosowano tabelę pośredniczącą *user_roles*, aby umożliwić przypisywanie wielu użytkownikom wiele ról.

- Poll Service – schemat bazy danych



Rysunek 15- Schamat bazy danych dla Poll Service

Opis dla Rysunek 15:

- id – identyfikator ankiety/pytania/zaproszenia/opcji
- published – flaga, która określa czy ankieta jest opublikowana
- createdDate – moment stworzenia ankiety/pytania/zaproszenia/opcji
- updatedDate - moment stworzenia ankiety/pytania/zaproszenia/opcji
- content – treść pytania/opcji
- active - flaga, która określa czy zaproszenie jest aktywne

- Result Service – schemat dokumentu

```
{
  [pollId]: {
    [questionId],
    answers: [
        [answerId]: [count]
    ]
  }
}
```

Listing 4 - Schamat dokumentu dla Result Service

```
{
  "1": {
    "1",
    answers: [
        "1": 10,
        "2": 15,
    ]
  }
}
```

Listing 5 - Przykładowa zawartość dokumentu dla Result Service

Opis dla Listing 4:

- pollId – identyfikator ankiety
- pollId – identyfikator pytania
- pollId – identyfikator odpowiedzi
- count – liczba oddanych głosów na daną odpowiedź

Kosztem czytelności wprowadzono strukturę, która zawiera minimalną ilość informacji. Listing 5 przedstawia przykładową zawartość statystyk, gdzie na odpowiedź o *id 1* oddano 10 głosów a *2* 15 głosów.

4.4 Kontrola dostępu

W celu wprowadzenia kontroli dostępu stworzono mechanizm autoryzacji, która określa czy zalogowany użytkownik spełnia wymagania na podstawie jego roli.

Każda metoda domyślnie jest chroniona przed dostępem publicznym. Dla metod z adnotacją *@Public* dostęp ma każdy klient aplikacji (nie jest wymagane uwierzytelnienie).

```
[…]
@Public()
login(@Body() data: CredentialsDto): Observable<string> {
  return this.client.send<string>(AUTH_USER_PATTERN, { body: { ...data }
});
}
[…]
```

Listing 6 - Metoda z dostępem publicznym

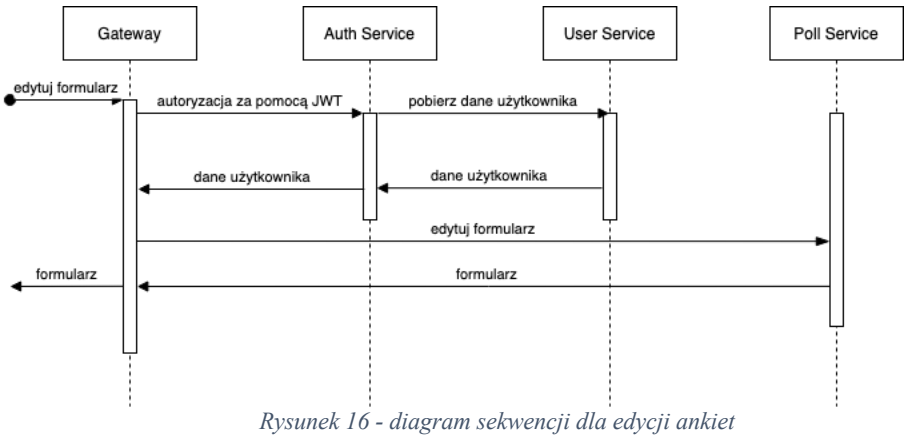
W celu zdefiniowania dostępu na podstawie roli, należy wprowadzić adnotację *@Roles()*, która jako argument przyjmuje tablicę wybranych ról, spośród dostępnych dla systemu.

```
[…]
@Post(':id')
@Roles([Role.Participant])
sendAnswer(
  @CurrentUser() user,
  @Param('id') pollId: string,
  @Body() answerDto: AnswerDto,
) {
  answerDto.pollId = pollId;
  answerDto.userId = user.id;
  return this.clientProxy.send(SEND_ANSWER_PATTERN, { ...answerDto });
}
[…]
```

Listing 7 - Metoda z dostępem dla wskazanej roli

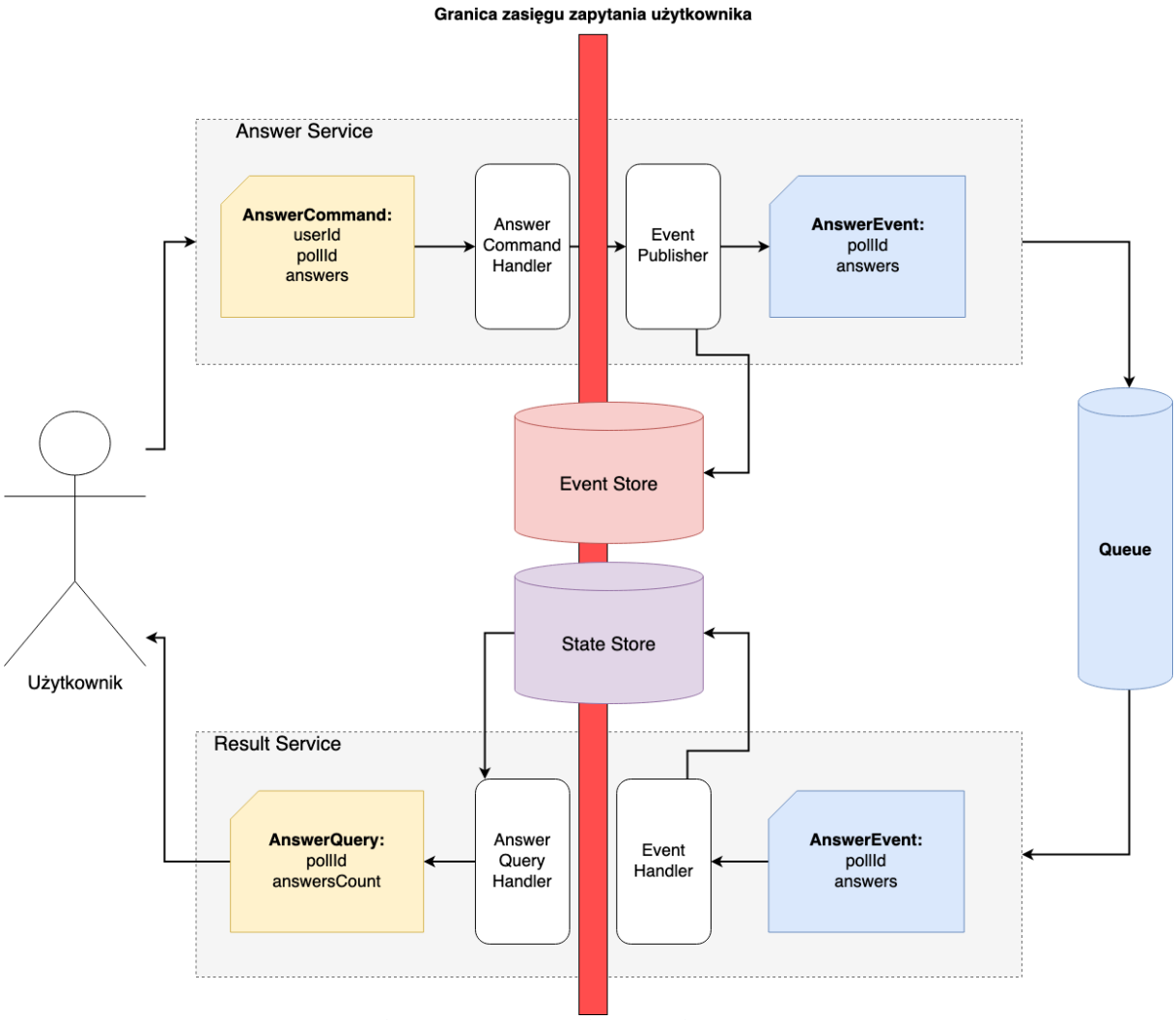
Rysunek 16 opisuje działanie autoryzacji. Wprowadzona adnotacja wymusza autoryzację użytkownika w Auth Service (weryfikacja JWT). Następnie token JWT jest dekodowany, na jego podstawie zostaje pobrany użytkownik z bazy danych. Wymagane role są porównywane do pobranych danych użytkownika, jeżeli użytkownik posiada wymaganą rolę *Gateway* zostaje poinformowany o poprawnej próbie autoryzacji do systemu.

Brama interfejsu agreguje dane [44] o uwierzytelnionym użytkowniku. Stosuje się to rozwiązanie, aby niwelować przypadki, gdzie dane użytkownika są wykorzystywane przez inne usługi (pobranie danych użytkownika następuje tylko raz przy autoryzacji).



Rysunek 16 - diagram sekwencji dla edycji ankiet

4.5 Implementacja CQRS oraz Event sourcing



Rysunek 17 - Uproszczony schamat realizacji CQRS i EventSourcing

Rysunek 17 przedstawia uproszczoną wizualizację realizacji wzorca CQRS oraz Event Sourcing. Przepływ operacji zapisu głosu jest następujący:

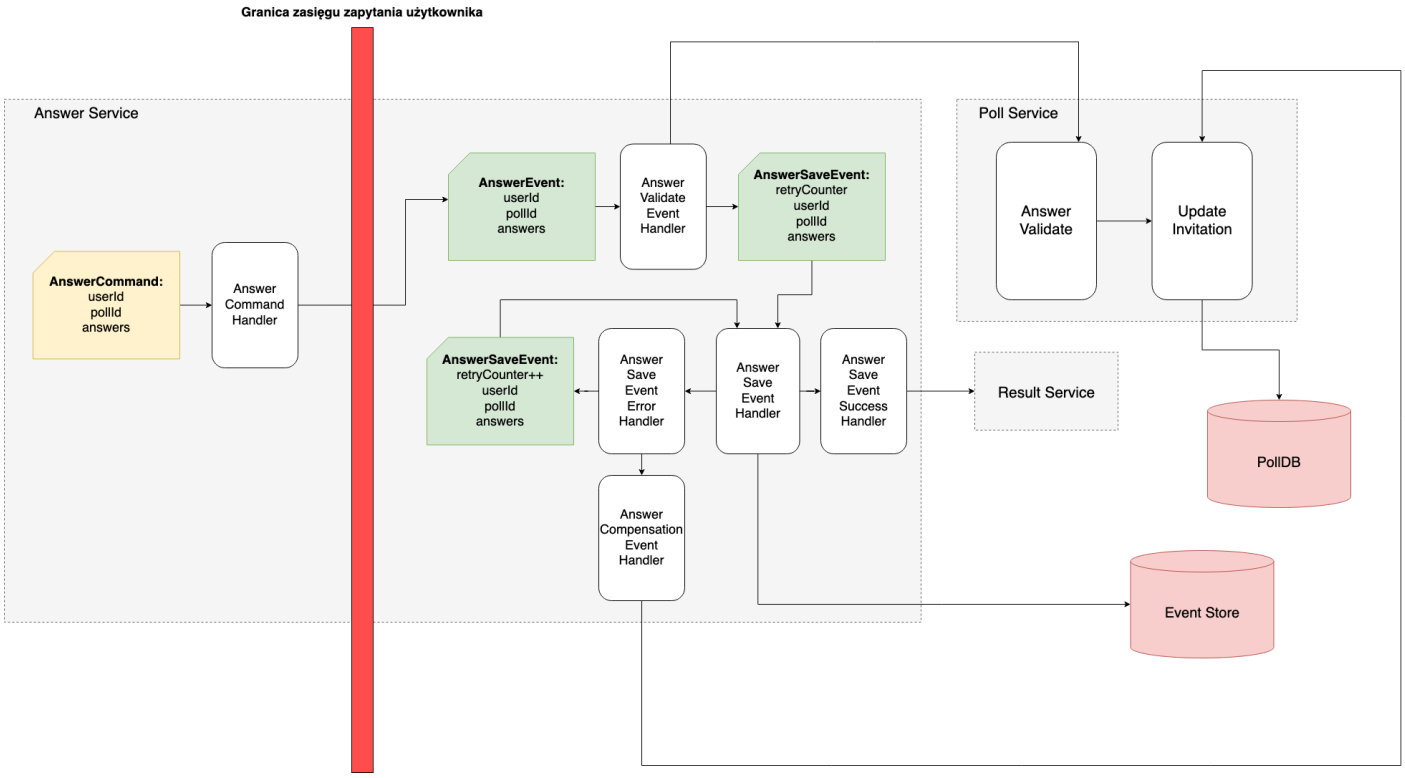
- Na podstawie danych zapytania tworzony jest obiekt *Answer Command*, który zostaje przekazany do *Answer Command Handler*.
- W tym momencie kończy się zasięg zapytania użytkownika. Dalsze operacje są wykonywane asynchronicznie, a do klienta jest zwracana informacja o dalszym przetwarzaniu danych.
- Kolejnym krokiem jest walidacja i próba zapisu zdarzenia do *Event Store*, który przechowuje dane w relacyjnej bazie danych *PostgreSQL*. Szczegółowe omówienie tej części znajduje się w opisie Rysunek 18.

- Po udanej próbie zapisu zdarzenia do bazy danych broker wiadomości otrzymuje komunikat, który jest przekazywany do kolejki zdarzeń.
- Komunikat zostaje skonsumowany przez jedną z instancji *Result Service*. *State Store* przechowuje statystyki głosowań w nierelacyjnej bazie danych *Redis*. W celu aktualizacji statystyk następuje odczyt aktualnych danych, a następnie denormalizacja komunikatu zdarzenia i zapis.

Główną wadą powyższego rozwiązania jest ewentualna spójność danych. Użytkownik po sprawdzeniu czy jego głos został zapisany może otrzymać informację, że wciąż ma możliwość zagłosowania, a statystyki nie zostały zaktualizowane. Jest to jednak kompromis, który umożliwia szybką odpowiedź serwera na próbę zagłosowania, które (jeżeli nie dojdzie do błędu) ostatecznie zostanie zapisane. Cały proces reagowania aplikacji na potencjalne błędy oraz zabezpieczenia są przedstawione w rozdziale 4.6.

Przekazanie statystyk do klienta jest realizowane poprzez odczyt bazy danych *State Store*. Ze względu na to, że w trakcie zapisu mogły nastąpić błędy, których nie da się automatycznie rozwiązać, (np. błąd sieci) statystki mogą być nieaktualne. Problem jest rozwiązany poprzez projekcje danych zawartych w *Event Store*, które zawsze zawierają rzeczywisty stan głosowania.

4.6 Zabezpieczenie przed awarią za pomocą Saga



Rysunek 18 - Zabezpieczenie przed awarią operacji zapisu głosu

Rysunek 18 przedstawia przepływ operacji wykonywanych w celu zapisu głosu z uwzględnieniem zabezpieczeń przed niepowodzeniem. Dla czytelności pominięto zapis po stronie *Answer Service* oraz komunikacje przez broker komunikatów, w celu zaprezentowania miejsc w usłudze *Poll Service* do których odnoszą się zdarzenia w *Answer Service*.

W celu zabezpieczenia zapisu głosu są wprowadzone następujące mechanizmy:

- Wykorzystanie wzorca Saga w połączeniu z choreografią kolejnych kroków, których logika jest zawarta w kolejnych zdarzeniach. Powodzenie każdego z kroków zależy od lokalnych transakcji w *Answer Service* oraz *Poll Service*.
- Kompensacja danych po nieudanej próbie zapisu do *Event Store*.
- Zabezpieczenie po stronie bazy danych. Wprowadzenie ograniczenia unikalności dla każdego głosu.
- Powtarzanie transakcji zapisu zdarzenia w *Event Store.*

Opis zdarzeń:

- *Answer Validate Event Handler* – odpowiada za przekazanie komunikatu o próbie zapisu głosu do *Poll Service*. Usługa *Poll Service* sprawdza czy dla

podanego użytkownika istnieje zaproszenie do wskazanej ankiety oraz waliduje głosy. Po walidacji zaproszenie zostaje zaktualizowane (blokada ponownego głosowania) oraz do *Answer Validate Event Handler* zwracana jest informacja o powodzeniu operacji.

```
//Klasa obsługuje zdarzenie walidacji głosu.
@EventsHandler(ValidateAnswerEvent)
export class ValidateAnswerEventHandler
  implements IEventHandler<ValidateAnswerEvent> {
  constructor(
    @Inject(POLL_SERVICE) private clientProxy: ClientProxy,
    private readonly publisher: EventBus,
  ) {}
  async handle(event: ValidateAnswerEvent) {
    const { userId, pollId, answers } = event;
    try {
        Logger.log(`AnswerEvent => Start validate vote`);
        //Przesłanie komunikatu do Poll Service, w razie błędu zwracany
        //jest wyjątek.
      await this.clientProxy
        .send(VALIDATE_ANSWER_PATTERN, { userId, pollId, answers })
        .pipe(timeout(2000))
        .toPromise();
      Logger.log(`AnswerEvent => End with success validate vote`);
        //W usłudze publikowane jest zdarzenie o powodzeniu walidacji.
      this.publisher.publish(new SaveAnswerEvent(userId, pollId, answers));
    } catch (err) {
      Logger.log(err);
      Logger.log(`AnswerEvent => End with error validate vote`);
    }
  }
}
```

Listing 8 – Implementacja Answer Validate Event Handler

Listing 8 przedstawia klasę *ValidateAnswerEventHandler,* która implementuje opisane zachowanie *Answer Validate Event Handler.*

- *Answer Save Event Handler* – wykonuje próbę zapisu zdarzenia do *Event Store* i przekazuje zdarzenie do *Answer Save Event Success Handler* lub informuje *Answer Save Event Error Handler* o niepowodzeniu.

```
//Klasa obsługuje zdarzenie zapisu głosu.
@EventsHandler(SaveAnswerEvent)
export class SaveAnswerHandler implements IEventHandler<SaveAnswerEvent> {
  constructor(private readonly publisher: EventBus) {}
  async handle(event: SaveAnswerEvent) {
    const { userId, pollId, answers, retryCounter } = event;
    try {
      Logger.log(`SaveAnswerEvent => Start save vote ${[…]}`);
        //Próba zapisu głosu
      […]
```

```
      await Answer.save(answer);
      Logger.log(`SaveAnswerEvent => End with success save vote ${[…]}`);
        //W usłudze publikowane jest zdarzenie o powodzeniu zapisu.
      this.publisher.publish(new SaveAnswerSuccessEvent(pollId, answers));
    } catch (err) {
      Logger.log(`SaveAnswerEvent => End with error save vote ${[…]}`);
        //W usłudze publikowane jest zdarzenie o niepowodzeniu zapisu.
      this.publisher.publish(
        new SaveAnswerErrorEvent(
          userId,
          pollId,
          answers,
          retryCounter,
          err.code,
        ),
      );
    }
  }
}
```

Listing 9 - Implementacja Answer Save Event Handler

Listing 9 przedstawia klasę *SaveAnswerHandler,* która implementuje opisane zachowanie *Answer Save Event Handler.*

- *Answer Save Event Success Handler* – przekazuje informację o zdarzeniu do brokera wiadomości.

```
//Klasa obsługuje zdarzenie poprawnego zapisu głosu.
@EventsHandler(SaveAnswerSuccessEvent)
export class SaveAnswerSuccessEventHandler
  implements IEventHandler<ValidateAnswerEvent> {
  constructor(
    @Inject(RESULT_SERVICE) private clientProxy: ClientProxy,
    private readonly publisher: EventBus,
  ) {}
  async handle(event: SaveAnswerSuccessEvent) {
    const { pollId, answers } = event;
    try {
      Logger.log(`SaveAnswerSuccessEvent => Start send result ${[…]}`);
        //Przesłanie komunikatu z informacją o głosie do Result Service.
      await this.clientProxy
        .send(SEND_RESULT_PATTERN, { pollId, answers })
        .pipe(timeout(2000))
        .toPromise();
      Logger.log(`SaveAnswerSuccessEvent => End with success send res[…]`);
    } catch (err) {
      Logger.log(err);
      Logger.log(`AnswerEvent => End with error send result ${[…]}`,
      );
    }
  }
}
```

Listing 10 - Implementacja Answer Save Event Success Handler

Listing 10 przedstawia klasę *SaveAnswerSuccessEventHandler,* która implementuje opisane zachowanie *Answer Save Event Success Handler.*

- *Answer Save Event Error Handler* – sprawdza ilość powtórzeń operacji zapisu, jeżeli nie została ona przekroczona lub nie naruszono ograniczenia unikalności to inkrementuje ilość powtórzeń i ponawia próbę zapisu. W innym przypadku wywołuje *Answer Compensation Event Handler*.

```
//Klasa obsługuje zdarzenie niepoprawnego zapisu głosu.
@EventsHandler(SaveAnswerErrorEvent)
export class SaveAnswerErrorHandler
  implements IEventHandler<SaveAnswerErrorEvent> {
  constructor(private readonly publisher: EventBus) {}
  handle(event: SaveAnswerErrorEvent): any {
    const { userId, pollId, answers, retryCounter, errorCode } = event;
        //Jeżeli liczba powtórzeń nie zostanie przekroczona lub błąd nie
        //wynika z naruszeń ograniczenia unikalności głosu to następuje
        //ponowne wykonanie próby zapisu.
    if (retryCounter >= 2) {
      Logger.log(`SaveAnswerEventError => Dropping vote ${[…]}`,);
      if (errorCode !== ANSWER_UNIQUE_VIOLATION)
        //W innym przypadku jest publikowane zdarzenie kompensacji danych.
        this.publisher.publish(new CompensationAnswerEvent(userId,pollId));
    } else {
      Logger.log(`SaveAnswerEventError => Retry vote ${[…]}`);
      this.publisher.publish(new SaveAnswerEvent(userId, pollId, answers,
      retryCounter + 1),
      );
    }
  }
}
```

Listing 11- Implementacja Answer Save Event Error Handler

Listing 11 przedstawia klasę *SaveAnswerErrorHandler,* która implementuje opisane zachowanie *Answer Save Event Error Handler.*

- *Answer Compensation Event Handler* – jest odpowiedzialny za kompensację danych po nieudanej próbie zapisu. Przekazuje komunikat do *Poll Service*, który cofa edycję zaproszenia do głosowania.

```
//Klasa obsługuje zdarzenie kompensacji danych.
@EventsHandler(CompensationAnswerEvent)
export class CompensationAnswerHandler
  implements IEventHandler<CompensationAnswerEvent> {
  constructor(
    @Inject(POLL_SERVICE) private clientProxy: ClientProxy,
    private readonly publisher: EventBus,
  ) {}
  async handle(event: CompensationAnswerEvent) {
    const { userId, pollId } = event;
```

```
    try {
      Logger.log(`SaveAnswerEvent => Start save vote ${[…]}`);
        //Do usługi Poll Service przekazana jest informacja o kompensacji
        //danych.
      await this.clientProxy
        .send(RESTORE_INVITATION_PATTERN, { userId, pollId })
        .pipe(timeout(2000))
        .toPromise();
      Logger.log(
        `SaveAnswerEvent => End with success save vote ${[…]}`,
      );
    } catch (err) {
      Logger.log(`SaveAnswerEvent => End with error save vote ${[…]}`);
    }
  }
}
```

Listing 12 - Implementacja Answer Compensation Event Handle

Listing 12 przedstawia klasę *SaveAnswerErrorHandler,* która implementuje opisane zachowanie *Answer Save Event Error Handler.*

5. Wyniki badań

5.1 Skalowanie

W celu zbadania możliwości skalowania rozwiązania, została wykorzystana opisana infrastruktura wdrożona na Docker Compose oraz stworzono prostą implementację klientów aplikacji w środowisku NodeJS. Każdy klient loguje się do systemu, a następnie oddaje głos. Taka operacja jest wykonywana przez określoną liczbę powtórzeń. Po zakończeniu pracy klienta, zwracany zostaje czas wykonywania opisanych operacji. Poniższa implementacja jest możliwa do wykorzystania jedynie dla danych testowych.

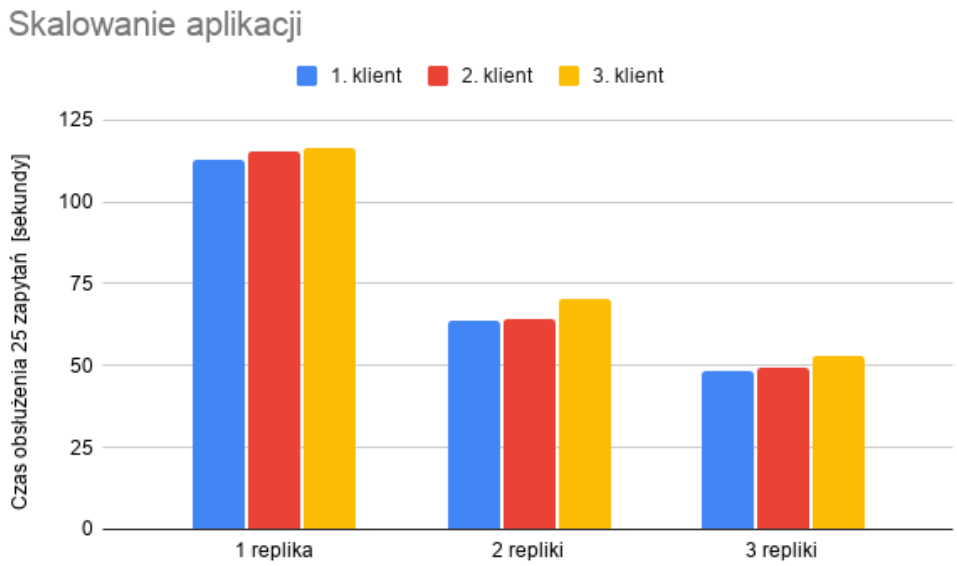
```
const axios = require("axios");

async function auth(username, password = "1234") {
  const data = await axios.post("http://localhost:8080/auth", {
    username,
    password,
  });

  return data.data.accessToken;
}

async function vote(token, pollId, answers) {
  const data = await axios.post(
    `http://localhost:8080/answer/${pollId}`,
    {
      answers,
    },
    { headers: { Authorization: `Bearer ${token}` } }
  );
  return data;
}

async function singleVote(from = 0, to = 25) {
  for (let i = from; i < to; i++) {
    let token = await auth(`participant-${i}`);
    await vote(token, "1", [
      {
        questionId: "1",
        answerId: "2",
      },
    ]);
  }
}

async function main() {
  await singleVote();
  console.timeEnd("vote");
}

console.time("vote");
main();
```

Listing 13 - implementacja klienta testującego skalowanie aplikacji

Skalowanie aplikacji odbywa się za pomocą dwóch poleceń terminalowych:

- Budowa oraz uruchomienie kontenerów:
  `docker-compose up --build`
- Skalowanie wskazanych usług bez ponownego tworzenia kontenerów:
  `docker-compose up --scale user-service=2 --no-recreate`

Wynikiem powyższych kroków jest stworzenie kopii instancji usługi *User Service* (jest to tylko przykład, w calach badawczych skalowano wszystkie usługi). Wdrożona aplikacja automatycznie łączy się z brokerem wiadomości *RabbitMQ*, bez potrzeby dodatkowej konfiguracji. Dla tego rozwiązania jedyną granicą skalowania są zasoby systemu, w badaniu przyznano pulę 4 jednostek przetwarzania oraz 7.5 GB pamięci RAM oraz ograniczono zasoby dla każdego kontenera, aby zapobiec rywalizacji o zasoby (opisane w Listing 1).



Rysunek 19 - wyniki badania skalowania aplikacji

Rysunek 19 przedstawia wykres kolumnowy, który opisuje zależność czasu przetwarzania żądań klientów od ilości kopii instancji usług. Można na jego podstawie wysunąć następujące wnioski:

- Skalowanie serwisów w górę, z 1 repliki na 2 repliki, skróciło czas obsługi zapytań klientów prawie o połowę.
- Skalowanie serwisów w górę, z 2 replik na 3 repliki nie przyniosło już tak efektywnego obniżenia czasu oczekiwania. Czynnikiem wpływającym na takie zachowanie może być ograniczony dostęp do współdzielonych jednostek przetwarzania.

- Czas zapytań klientów jest do siebie zbliżony. Powodem takiego zachowania jest zastosowanie równoważenia obciążenia dla serwera *nginx* oraz brokera wiadomości z algorytmem karuzelowym.

Powyższa analiza daje obiecujące wyniki dla tego rozwiązania. Elastyczne skalowanie umożliwia dostosowanie działania aplikacji w zależności od natężenia ruchu sieciowego. Zachowanie jest zgodne z wprowadzoną konfiguracją, dzięki czemu łatwiej jest wprowadzać modyfikację parametrów w celu uzyskania lepszych wyników.

W celu weryfikacji zużycia procesora dla każdego przypadku testu z Rysunek 19, skorzystano z narzędzia *top* [45], które prezentuje aktualne zużycie zasobów systemu przez procesy.

```
PID    COMMAND      %CPU   […]
710    com.docker.h 40.7   […]
[…]
```

Listing 14 - Wykorzystanie procesora dla testu z jedną repliką usług

```
PID    COMMAND      %CPU   […]
710    com.docker.h 81.0   […]
[…]
```

Listing 15 - Wykorzystanie procesora dla testu z dwiema replikami usług

```
PID    COMMAND      %CPU   […]
710    com.docker.h 124.5  […]
[…]
```

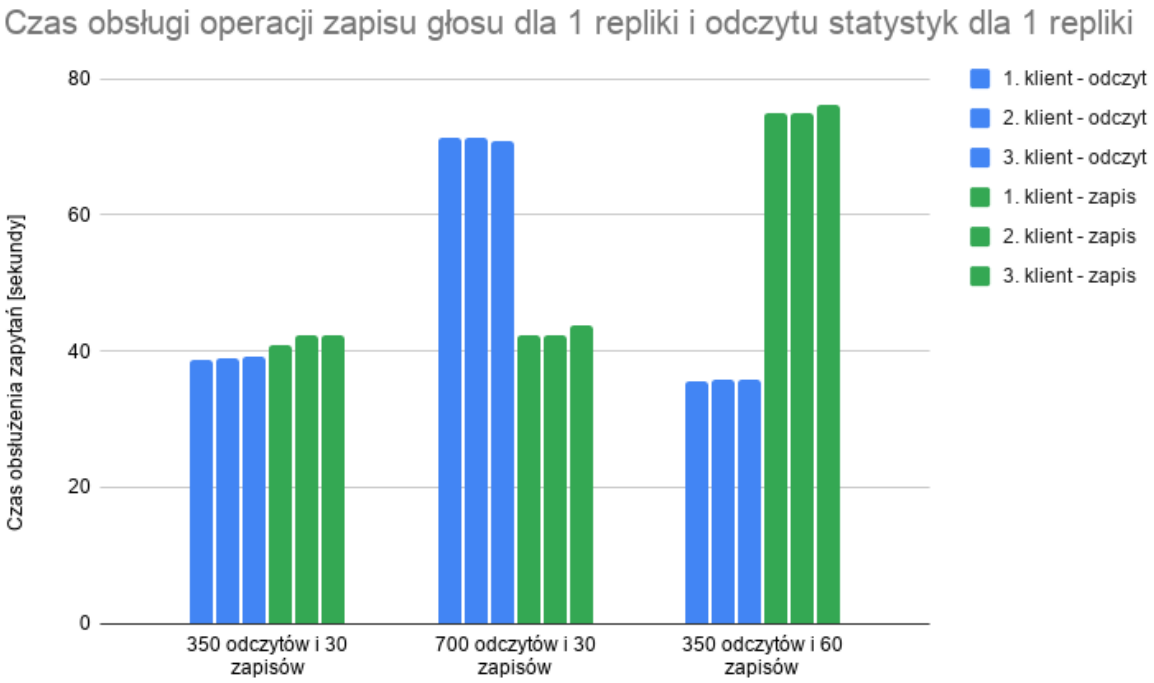
Listing 16 - Wykorzystanie procesora dla testu z trzema replikami usług

Dla lepszej czytelności w listingach pozostawiono jedynie informację o wykorzystaniu procesora przez proces platformy Docker w kolumnie *%CPU*. Wyniki przedstawione w Rysunek 19 wskazują na brak wystarczających zasobów jednostki przetwarzania dla testu z trzema replikami usług, co potwierdza Listing 16.

Badanie wykazało również problem. Brakuje monitorowania aplikacji, które umożliwiłoby analizę pracy systemu na bieżąco, a także narzędzia do automatycznego podejmowania decyzji związanych ze skalowaniem oraz przyznawaniem zasobów. Problem można rozwiązać przez wdrożenie orkiestratora kontenerów *Kubernetes* [46].

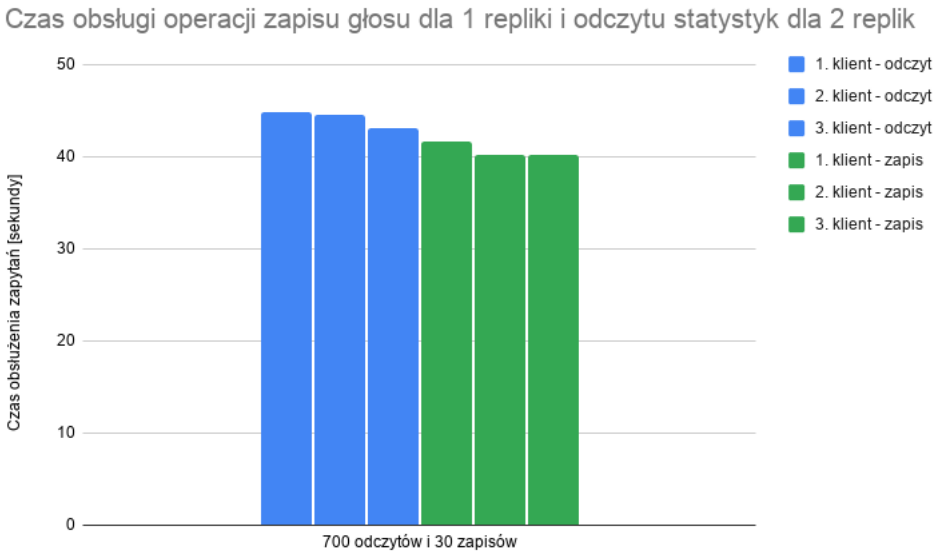
W celu zbadania skalowalności zapisu i odczytu przeprowadzono test, gdzie równocześnie wykonywano operację:

- odczytu statystyk – 3 klientów
- zapisu głosu – 3 klientów

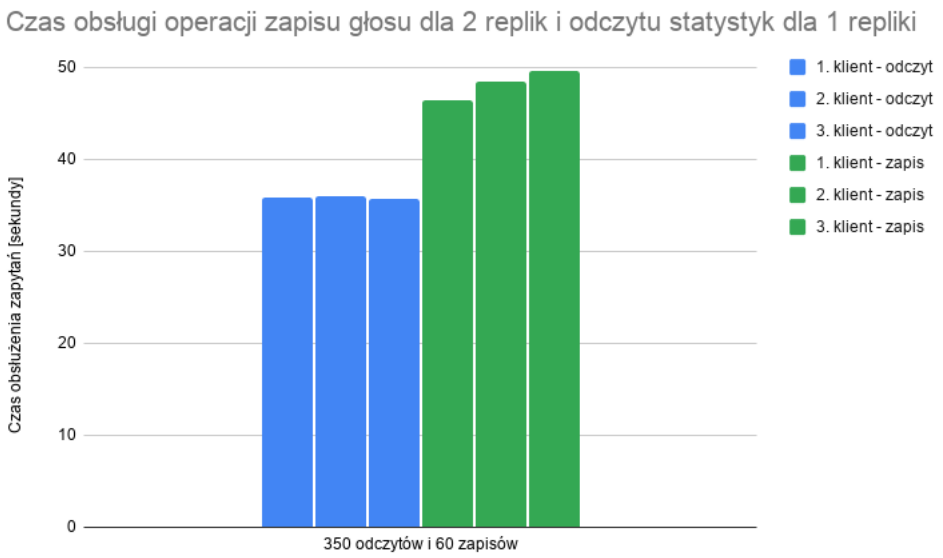


Rysunek 20 - Wykres czasu obsługi operacji zapisu głosu dla 1 repliki i odczytu statystyk dla 1 repliki

Rysunek 20 przedstawia wpływ ilości zapytań odczytu i zapisu na czas obsługi. Dla pierwszego przypadku dobrano takie ilości, aby czas zapisu i odczytu był do siebie zbliżony. Przypadek drugi i trzeci przedstawia wyniki dla podwojonej ilości operacji odczytu, a następnie zapisu. Dzięki zastosowaniu izolacji każdej z usług na platformie Docker, można dokonać skalowania w celu wyrównania czasu oczekiwania na zapis i odczyt.



Rysunek 21 – Wykres czasu obsługi operacji zapisu głosu dla 1 repliki i odczytu statystyk dla 2 replik



Rysunek 22 – Wykres czasu obsługi operacji zapisu głosu dla 2 replik i odczytu statystyk dla 1 repliki

Rysunek 21 i Rysunek 22 przedstawiają wyniki skalowania, gdzie podwojono instancje usług odpowiedzialnych za bardziej obciążoną operację. Czas obsługi żądań zapisu i odczytu dla przypadków, które różniły się w Rysunek 20, są teraz do siebie zbliżone. Rezultaty potwierdzają, że stosowanie architektury zorientowanej na usługi daje możliwość skalowania rozwiązania w zależności od zapotrzebowania.

5.2 Awaria asynchronicznego zapisu głosu

W tej części zostanie przedstawione działanie zabezpieczenia przed błędem zapisu głosu do bazy danych opisanego w rozdziale 4.6. W celu wywołania przypadku, gdzie operacja zapisu zostanie powtórzona dwa razy, a następnie głos zostanie zapisany dodano do implementacji następującą zmianę.

```
@EventsHandler(SaveAnswerEvent)
export class SaveAnswerHandler implements IEventHandler<SaveAnswerEvent> {
  constructor(private readonly publisher: EventBus) {}
  async handle(event: SaveAnswerEvent) {
    const { userId, pollId, answers, retryCounter } = event;
    try {
      […]
      if (retryCounter < 2) throw new Error(); //Dodana linia kodu
      […]
    } catch (err) {
      […]
  }
}
```

Listing 17 - Zmodyfikowana implementacja Save Answer Handler

Listing 17 zawiera modyfikację metody *SaveAnswerHandler*, do której dodano warunek, wymuszający powtórzenie operacji minimum dwa razy. Następnie następuje próba zagłosowania.

```
curl --location --request POST
'http://localhost:8080/answer/1' \
--header 'Authorization: Bearer […]\
--header 'Content-Type: application/json' \
--data-raw '{
    "answers": [
        {
            "questionId": "1",
            "answerId": "2"
        }
    ]
}'
CHECK_YOUR_VOTE
```

Listing 18 - Oddanie głosu z użyciem narzędzia curl

Powyżej kolorem zielonym zaznaczono polecenie, które przedstawia oddanie głosu przez użytkownika. W odpowiedzi zwracany jest kod *CHECK_YOUR_VOTE* zaznaczony kolorem żółtym, który informuje użytkownika o sprawdzeniu głosu poprzez kolejne żądanie. Implementacja może wykorzystać technologię *WebSockets* [47], w celu ograniczenia potrzeby wykonywania dodatkowych zapytań dla weryfikacji statusu głosu. Kolejnym rozwinięciem implementacji może być wprowadzenie rozbudowanego statusu głosu, który informowałby klienta o błędzie zapisu.

```
curl --location --request GET
'http://localhost:8080/poll/invitation' \ --header
'Authorization: Bearer […]
[{"userId":"27","pollId":1,"active":false,"createdDate":"2021-
01-19T09:42:26.733Z","updatedDate":"2021-01-
21T18:01:24.934Z"}]
```

Listing 19 - Sprawdzenie oddanego głosu

```
curl --location --request GET 'http://localhost:8080/result/1'
{"1":{"2":2}}
```

Listing 20 - Sprawdzenie statystyk głosowania

Powyższe listingi przedstawiają komendy, które sprawdzają czy dany głos został zapisany. Listing 19 weryfikuje zaproszenie, które straciło ważność po oddaniu głosu. Listing 20 przedstawia zaktualizowane statystyki.

1. **gateway_1 Start request from ::ffff:172.19.0.5**
2. answer-service_1AnswerCommandHandler => **{"userId":27,"pollId":"1","answers":[{"questionId":"1","answerId":"2"}]}**
3. answer-service_1 AnswerEvent => **Start validate vote**
4. answer-service_1 AnswerEvent => **End with success validate vote**
5. answer-service_1 SaveAnswerEvent => **Start save vote {[…],"retryCounter":0}**
6. answer-service_1 SaveAnswerEvent => **End with error save vote {[…],"retryCounter":0}**
7. answer-service_1 SaveAnswerEventError => **Retry vote {[…],"retryCounter":0}**
8. answer-service_1 SaveAnswerEvent => **Start save vote {[…],"retryCounter":1}**
9. answer-service_1 SaveAnswerEvent => **End with error save vote {[…],"retryCounter":1}**
10. answer-service_1 SaveAnswerEventError => **Retry vote{[…],"retryCounter":1}**
11. answer-service_1 SaveAnswerEvent => **Start save vote {[…],"retryCounter":2}**
12. answer-service_1 SaveAnswerEvent => **End with success save vote {[…],"retryCounter":2}**
13. answer-service_1 SaveAnswerSuccessEvent => **Start send result** {"pollId":"1","answers":[{"questionId":"1","answerId":"2"}]}

```
14.     result-service_1 ResultService => Start save answer
   1, [{"questionId":"1","answerId":"2"}]
15.     answer-service_1 SaveAnswerSuccessEvent => End with
   success send result
```

Listing 21 - Zapis dziennika zdarzeń systemu dla powtarzanej operacji zapisu głosu

Dziennik zdarzeń z Listing 21 przedstawia zapis operacji wykonanych w ramach zapisu głosu do bazy danych. Zostały tylko uwzględnione zdarzenia z usług *Answer Servic*e i *Result Service* z zaznaczeniem opisu stanu zapytania dla zwiększenia czytelności.

W szeregu zdarzeń zawartych w Listing 21 jest odzwierciedlony założony przypadek, gdzie operacja zapisu zostaje powtórzona dwa razy, a następnie zatwierdzona. Dodatkowo wykazano moment walidacji zapytania oraz przekazanie zdarzenia do *Result Service.*

6. Zakończenie

6.1 Podsumowanie i wnioski

Architektura zorientowana na usługi została stworzona w celu zwiększenia wydajności rozwijania i utrzymania aplikacji. W niniejszej pracy zostało przedstawione, że to rozwiązanie można udoskonalić, korzystając z wzorców architektonicznych, takich jak: *CQRS, Event Sourcing* i *Saga*. Po zaimplementowaniu modułów niwelujących występowanie błędów, system jest wiarygodny i może zostać wdrożony na środowisko produkcyjne. Zapewniając infrastrukturę, która zapewnia skalowanie instancji serwisów, aplikacja jest w stanie obsłużyć dowolną ilość klientów, ograniczając się jedynie do dostępnych zasobów. Odizolowanie środowisk pracy każdego z serwisów jest zaletą przy wprowadzaniu osobnych zespołów dla każdej z usług. Stosując się do ustalonych zasad komunikacji można niezależnie modyfikować usługę pod względem użytej technologii czy praktyk tworzenia oprogramowania.

Wadą wprowadzenia tej architektury jest zdecydowanie nakład pracy jaki trzeba poświęcić do przygotowania planu działania systemu oraz czas na przygotowanie środowiska. Osoba odpowiedzialna za decyzje architektoniczne musi przewidywać potencjalne błędy, które wpływają na pracę każdego zespołu odpowiedzialnego za usługę. Nie jest to zdecydowanie wzorzec, który powinien być zastosowany dla małych projektów lub małego zespołu programistów. Infrastruktura wymaga wysokich kwalifikacji w zakresie zarządzania wieloma aplikacjami na szeroką skalę oraz zwiększa początkowe koszty wdrożenia systemu.

6.2 Realizacja efektów kształcenia

- Krytycznie analizować teksty naukowe, dobierać i kompletować literaturę źródłową dotyczącą wybranego problemu informatycznego – Rozdział 1 i 2
- Identyfikować oraz formułować specyfikację nietypowego problemu informatycznego – Rozdział 3 i 5
- Formułować i testować hipotezy związane z wybranym problemem informatycznym – Rozdział 6
- Integrować potrzebną wiedzę z różnych dziedzin i dyscyplin oraz zastosować podejście systemowe, uwzględniające także aspekty pozatechniczne – Rozdział 4 i 6
- Oceniać przydatność i możliwość wykorzystania nowych technologii informatycznych Rozdział 4

- Projektować poszczególne etapy samodzielnego rozwiązania nietypowego problemu informatycznego – Rozdział 5
- Proponować ulepszenia/usprawnienia istniejących rozwiązań informatycznych – Rozdział 5 i 7
- Konstruować i redagować pracę o charakterze naukowo-technicznym – Rozdział 2
- Posługiwać się jasnym i precyzyjnym językiem używanym w informatyce Rozdział 1 i 2

6.3 Dalszy rozwój

Przedstawiona praca jest jedynie propozycją rozwiązania, która nie posiada kilku istotnych elementów:

- Brak zabezpieczenia łącza pomiędzy klientem, a bramą dostępu.
- Nie zastosowano tokenów, które byłyby walidowane przez każdy serwis w celu potwierdzania autoryzacji żądania w sieci wewnętrznej.
- Optymalizacja projekcji. Przy wzrastającej ilości odpowiedzi operacja projekcji będzie wykonywała się coraz dłużej. Można zastosować migawki, które będą minimalizowały ilość danych poddawanych projekcji.

7. Bibliografia

[1] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown i R. Metz, „Reference Model for Service Oriented Architecture 1.0," 2006. [Online]. Available: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html.

[2] IBM Cloud Education , „ Message Brokers," 23 1 2020. [Online]. Available: https://www.ibm.com/cloud/learn/message-brokers.

[3] A. B. Bondi, „Characteristics of scalability and their impact on performance," 2000.

[4] H. C. Yichang i R. Chen, „The Implementation of Database High Availability Infrastructure in TGPMS," w *2010 International Conference on Computational and Information Sciences*, 2010.

[5] J. Peters, J. M. E. v. d. Werf i J. Hage, „Architectural Pattern Definition for Semantically Rich Modular Architectures," w *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016.

[6] M. Rahman, S. Iqbal i J. Gao, „Load Balancer as a Service in Cloud Computing," w *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, 2014.

[7] L. Hohmann, Beyond Software Architecture: Creating and Sustaining Winning Solutions 1st Edition, Addison-Wesley Professional, 2003.

[8] C. Richardson, „microservices.io," 2018. [Online]. Available: https://microservices.io/patterns/monolithic.html.

[9] M. Fowler, „MonolithFirst," 3 6 2015. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html.

[10] M. Fowler, „IntegrationTest," 16 1 2018. [Online]. Available: https://martinfowler.com/bliki/IntegrationTest.html.

[11] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck i M. S. , „Continuous Deployment at Facebook and OANDA," w *International Conference on Software Engineering Companion (ICSE-C)*, 2016.

[12] M. Fowler, „Technical Debt," 21 5 2019. [Online]. Available: https://www.martinfowler.com/bliki/TechnicalDebt.html.

[13] S. Mankovskii, „Loose Coupling,” 2009. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_1187.

[14] K. M. Goertzel, T. Winograd i B. A. Hamilton, Safety and Security Considerations for Component-Based Engineering of Software-Intensive Systems, Naval Ordnance Safety and Security Activity, 2011.

[15] C. Zhu, M. Chai, Y. Lu i Y. Guo, „Service Oriented Architecture Design of Energy Consumption Information System about Petroleum Enterprise,” w *nternational Conference on Computational and Information Sciences*, Shiyang, 2013.

[16] I. C. Education, „Containerization,” 15 5 2019. [Online]. Available: https://www.ibm.com/cloud/learn/containerization.

[17] I. C. Education, „PaaS (Platform-as-a-Service),” 29 10 2019. [Online]. Available: https://www.ibm.com/cloud/learn/paas.

[18] R. Chris, „A pattern language for microservices,” 2020. [Online]. Available: https://microservices.io/patterns/.

[19] A. Wiggins, „The Twelve-Factor App,” 2017. [Online]. Available: https://www.12factor.net.

[20] M. E. Conway, „How doo committees invent?,” *F. D. Thompson Publications, Inc.,* 1968.

[21] E. Haddad, „Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow,” Uber, 8 9 2015. [Online]. Available: https://eng.uber.com/service-oriented-architecture/.

[22] A. Singh, P. Chawla, K. Singh i A. K. Singh, „Formulating an MVC Framework for Web Development in JAVA,” w *018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, 2018.

[23] N. Levy i F. Losavio, „Analyzing and comparing architectural styles,” w *Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society*, 1999.

[24] S. J. Fowler, Mikrousługi : wdrażanie i standaryzacja systemów w organizacji inżynierskiej, Helion, 2017.

[25] R. Rodger, Tao mikrousług : projektowanie i wdrażanie, Helion, 2019.

[26] C. Richardson, Microservices patterns, Manning, 2018.

[27] C. Richardson, „The Scale Cube," 2020. [Online]. Available: https://microservices.io/articles/scalecube.html.

[28] M. Fowler, „Data Transfer Object," 2020. [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html.

[29] G. Sanders i S. Shin, „Denormalization effects on performance of RDBMS," w *Proceedings of the 34th Annual Hawaii International Conference on System Sciences, Maui*, 2001.

[30] S. Greif i R. Benitte, „JavaScript Flavors," 2020. [Online]. Available: https://2020.stateofjs.com/en-US/technologies/javascript-flavors/.

[31] O. Foundation, „About Node.js," [Online]. Available: https://nodejs.org/en/about/.

[32] „Transpiler," Devopedia, 2019. [Online]. Available: https://devopedia.org/transpiler.

[33] H. Jiang, M. Shi i S. Li, „Redis-based web server cluster session maintaining technology," w *13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*, 2017.

[34] K. Riede, „RDBMS -an introduction to Relational Database Management Systems," Groms, 2002.

[35] „DB-Engines Ranking of Relational DBMS," solid IT , 2021. [Online]. Available: https://db-engines.com/en/ranking/relational+dbms.

[36] T. P. G. D. Group, „What is PostgreSQL?," 2021. [Online]. Available: https://www.postgresql.org/about/.

[37] M. Fowler i D. Rice, Patterns of Enterprise Application Architecture, Addison-Wesley Professiona, 2003.

[38] „Which protocols does RabbitMQ support?," VMware Inc., 2020. [Online]. Available: https://www.rabbitmq.com/protocols.html.

[39] I.-C. Donca, C. Corches, O. Stan i L. Miclea, „Autoscaled RabbitMQ Kubernetes Cluster on single-board computers," w *2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2020.

[40] NGINX, „HTTP Load Balancing,” F5, 2021. [Online]. Available: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

[41] R. Chris, „Pattern: API Gateway / Backends for Frontends,” 2020. [Online]. Available: https://microservices.io/patterns/apigateway.html.

[42] „ Direct Reply-to,” VMware Inc., 2020. [Online]. Available: https://www.rabbitmq.com/direct-reply-to.html#usage.

[43] C. Richardson, „Pattern: Database per service,” [Online]. Available: https://microservices.io/patterns/data/database-per-service.html.

[44] D. Martin, „Korzystanie z bram interfejsu API w mikrousługach,” 23 10 2018. [Online]. Available: https://docs.microsoft.com/pl-pl/azure/architecture/microservices/design/gateway.

[45] M. Kerrisk, „top(1) — Linux manual page,” 9 2020. [Online]. Available: https://man7.org/linux/man-pages/man1/top.1.html.

[46] „Kubernetes Documentation,” The Linux Foundation, 20 11 2020. [Online]. Available: https://kubernetes.io/docs/home/.

[47] S. Arora, J. Maini, P. Mallick, P. Goel i R. Rastogi, „Efficient E-learning management system through web socket,” w *3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2016.

8. Spis ilustracji

9. Spis listingów

10. Spis tabel