Spis treści

1 Wstęp

1.1 Wprowadzenie

W ostatnich latach w świecie IT można było zauważyć trend polegający na promowaniu budowania dużych systemów jako aplikacji rozproszonych składających się z wielu serwisów. Prelegenci na różnych konferencjach programistycznych przepowiadali schyłek wszelkich problemów podczas budowania wielkich systemów wykorzystywanych w biznesie. W dniu dzisiejszym rozwiązanie to określane jest architekturą opartą na **mikroserwisach**. Koncepcja ta nie jest czymś zupełnie nowym i możemy określić ją jako rozwinięcie architektury SOA (Service Oriented Architecture). Idea tworzenia oprogramowania w architekturze mikroserwisowej polega na budowaniu autonomicznych komponentów z których każdy odpowiada za konkretne zadanie. Elementy te współpracując ściśle ze sobą pozwalają na dostarczenie wymaganej logiki biznesowej w systemach rozproszonych.

Wielu architektów oraz zespołów programistów wizja rozbicia swojej monolitycznej aplikacji na wiele mikroserwisów zachęciła do prób budowania takich rozwiązań. Pomimo początkowego entuzjazmu popartego niezaprzeczalnymi zaletami mikroserwisów takimi jak:

- możliwości stosowania różnych technologii
- skalowalności
- odporności na awarie

dały o sobie znać wady, które spowodowały, że projektowanie, implementacja oraz utrzymanie takiej architektury stało się olbrzymim wyzwaniem dla firm produkujących oprogramowanie. Jednym z największych problemów okazał się sposób komunikacji pomiędzy mikroserwisami. Tym samym projektowanie interfejsów API [7] wymagało od architektów oraz programistów rozwiązania problemów w następujących kwestiach:

- wyboru formatu wymiany danych (JSON, XML itp.)
- zaprojektowania punktów końcowych wywołania serwisów
- obsługi błędów

- wydajności (ilość danych przy jednym wywołaniu serwisu, czas oczekiwania na odpowiedź)
- implementacji uwierzytelniania
- wybór technologii w której zostaną utworzone mikroserwisy

Mimo tych przeszkód korporacje pokroju Amazon, Netflix czy Google budują swoje olbrzymie systemy rozproszone w oparciu o mikroserwisy wykorzystując zróżnicowany stos technologii. Architektura mikroserwisowa przyczyniła się do spopularyzowania takich technologii jak chmura obliczeniowa (Amazon Web Services, Azure itp.) oraz rozwiązań opartych na kontenerach (Docker, Kubernetes).

Technologiczny potentat jakim jest niewątpliwe firma Google opierając swój biznes na usługach sieciowych stworzyła w tym celu technologię opartą na protokole RPC (Remote Procedure Call), która miałaby być panaceum na wyżej wymienione problemy w stosunku do „klasycznego" podejścia opartego na technologii REST.

1.2 Cel i założenia pracy

Celem niniejszej pracy jest zaprojektowanie oraz implementacja rozwiązania, które pozwoli rozbić prosty system monolityczny na grupę niezależnych usług sieciowych. Aplikacja zostanie zbudowana przy użyciu *„klasycznego"* podejścia, gdzie wykorzystuje się komunikację opartą na technologii REST oraz przy udziale komunikacji opartej na zdalnym wywołaniu procedur(RPC). Logika każdego z serwisów zostanie oddzielona od warstwy transportu tak, aby w przyszłości było możliwie rozbudowanie systemu o dodatkowe funkcjonalności. Implementacja pozwoli opisać typowe problemy z jakimi spotykamy się podczas tworzenia architektury mikroserwisowej do których należą:

1. Wykorzystanie różnych technologii do budowy poszczególnych komponentów systemu. Zakłada się wykorzystanie co najmniej dwóch języków programowania. Natura systemów rozproszonych sprawia, że nie są budowane przez jeden zespół programistów. Usługi, które stanowią część większego systemu realizują różne zadania. Nie istnieje jeden stos technologiczny, który byłby odpowiedni do każdego postawionego przed nim zadania więc należy używać odpowiednio dobranej technologii tam gdzie sprawdzi się najlepiej.
2. Definiowanie punktów końcowych w komunikacji między usługami. Problemem w tym przypadku jest poprawne zdefiniowanie jakie dane oraz w jaki sposób będą przekazywane do poszczególnych usług. W przypadku podejścia wykorzystującego styl architektury REST są to dobrze skonstruowane adresy URI wraz z wersjonowaniem wersji API. W przypadku zdalnego wywołania procedur przejrzysta dokumentacją zdalnych metod wraz dokładny opis struktur przekazywanych w żądaniu oraz zwrotnych w odpowiedzi będzie istotnym wyzwaniem mającym znaczący wpływ na użyteczność mikroserwisów.

3. Zabezpieczenie danych przed nieuprawnionym przechwyceniem i odczytem. Informacje przekazywane pomiędzy usługami należy odpowiednio zabezpieczyć, nie może to być czysty tekst. W tym celu serwery HTTP oraz RPC będą implementować protokół SSL/TLS. Na potrzeby aplikacji działającej w obrębie jednej maszyny zostanie wygenerowany klucz prywatny podpisany przez własne główne centrum certyfikacji CS(ang. certificate authority). W rozwiązaniach, które będą funkcjonowały świecie rzeczywistym należy używać niezależnego oraz zaufanego centrum certyfikacji.

4. Wydajność przesyłania oraz przetwarzania danych. W przypadku aplikacji monolitycznej przekazywanie danych pomiędzy poszczególnymi modułami odbywa się w pamięci lokalnej. W przypadku architektury mikroserwisowej dochodzi dodatkowa warstwa transportowa. Od formatu danych oraz użytego protokołu zależy ile czasu każda z usług będzie potrzebowała na przyjęcie informacji ewentualną serializację oraz przesłanie informacji zwrotnej. Pomiar wydajności zostanie zaimplementowany w kliencie i będzie obejmował wielokrotne wywołanie wybranych metod. Czas jaki upłynie od chwili przesłania żądania do momentu otrzymania informacji zwrotnej dla każdego udanego wywołania będzie sumowany dla kilku wybranych wartości powtórzeń w pętli.

Powyższe zagadnienia to nie jedyne problemy z jakimi muszą się zmierzyć projektanci mikroserwisów. Dochodzi do tego wiele aspektów związanych ze skalowaniem, obsługą błędów, monitorowaniem, ciągłej integracji źródeł kodu(Continuous Integration) itp. jednak są to zadania, którymi trzeba się zająć w dalszym etapie projektowania systemu i nie są bezpośrednio związane z metodami transportu danych.

2 Teoretyczne podstawy integracji mikroserwisów

2.1 Wstęp

W systemach rozproszonych zbudowanych z kilkudziesięciu lub nawet kilku tysięcy mikroserwisów komunikacja pomiędzy nimi jest jednym z najważniejszych aspektów. Od poprawnego zaprojektowania API oraz doboru właściwych technologii zależy, czy system będzie elastyczny jeśli chodzi o integrację z zewnętrznymi odbiorcami oraz wykorzysta większość zalet jakie oferują mikroserwisy.

2.2 Technologie oraz protokoły używane do komunikacji pomiędzy mikroserwisami

2.2.1 HTTP oraz HTTP/2

Z początkiem lat dziewięćdziesiątych XX wieku na skraju powstania ogólnoświatowego medium informacji jakim jest internet opracowano prosty protokół **HTTP** (***(ang. Hypertext Transfer Protocol)***. Będąc umiejscowionym w najwyższej warstwie (aplikacji) modelu TCP/IP służy do obsługi żądań oraz odpowiedzi w relacji klient-serwer [5]. Każde żądanie oraz odpowiedź mają postać wiadomości, która jest zapisana w postaci tekstu przez co jest prosta w odczycie przez człowieka. Wiadomości składają się z następujących elementów:

- **Linia początkowa**-podczas żądania w tej linii określa się z jakiej metody HTTP chce się skorzystać (m.in. GET, POST, PUT, DELETE) oraz wersję protokołu (najczęściej HTTP/1.1). Odpowiedź zawiera wersję protokołu oraz kod statusu odpowiedzi (np. 200, 404, 500)
- **Nagłówki HTTP**-nagłówki zawierają dane w postaci *nazwa-nagłówka: wartość-nagłówka*, które określają m.in. adres serwera, informacje na temat połączenia czy format danych jakie są akceptowane [5]
- **Ciało wiadomości**-opcjonalny element wiadomości zwierający dane w dowolnym formacie uzgodnionym w nagłówku

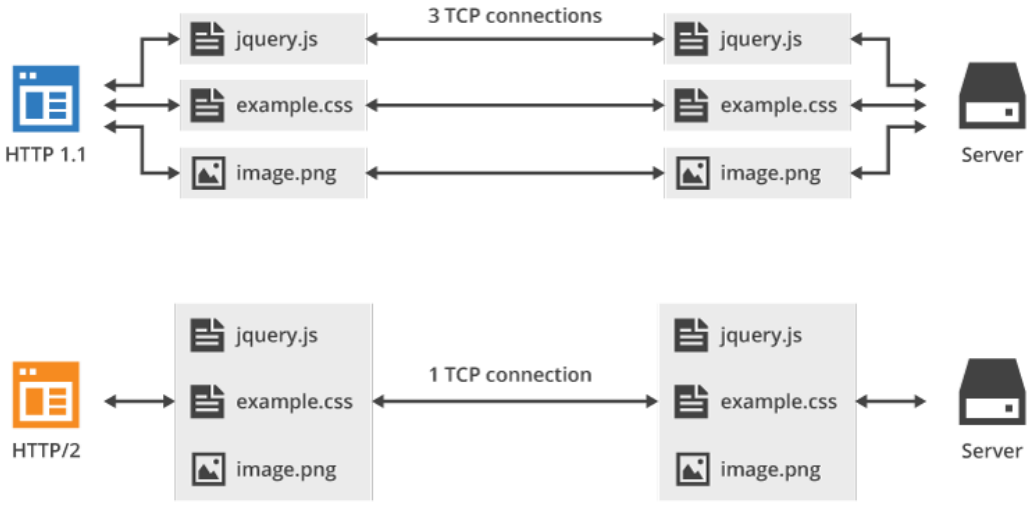
Protokół HTTP jest bezstanowy co oznacza, że serwer nie może zapisać stanu żądania od klienta i użyć go gdy przyjdzie następne. Każdy zasób sieciowy do którego odwołuje się klient jest identyfikowany za pomocą **URI** ***(ang. Uniform Resource Identifier)*** [2]. Jest to unikalny identyfikator składający się z sekwencji znaków będący referencją do konkretnego zasobu umieszczonego na serwerze HTTP. Prostota oraz całkiem jawny sposób przesyłania wiadomości powoduje, że protokół HTTP jest całkowicie niezabezpieczony umożliwiając tym samym proste przechwycenie wiadomości. Panaceum na tą dolegliwość jest rozszerzana wersja **HTTPS** ***(ang. Hypertext Transfer Protocol Secure)***. Wiadomości przenoszone za pomocą tego protokołu są dodatkowa szyfrowane za pomocą certyfikatów SSL.

Przez lata HTTP/1.1 był najbardziej popularnym protokołem używanym w sieci internetowej oraz podczas korzystania z usług sieciowych. Jednak zwiększające zasoby na stronach internetowych powodowały, że do wczytania wszystkich treści potrzeba setek żądań. Powoduje to wolniejsze wczytywanie bogatych w treści i multimedia portali. Dlatego w roku 2015 organizacja **IETF** ***(Internet Engineering Task Force)*** opracowała standard HTTP/2[1]. Jednymi z głównych zmian względem protokołu HTTP/1.1 są:

1. Dane przekazywane za pomocą protokołu są zapisane binarnie przez co wielkość wiadomości ulega znacznemu zmniejszeniu oraz zwiększa wydajność parsowania danych,

2. Transmisja danych odbywa się za pomocą jednego połączenia TCP eliminując tym samym obciążenie serwera związane z ciągłym odpytywaniem przez klienta o kolejne zasoby [9],

Rysunek 2.1: Porównanie działania połącznia protokołu HTTP/1.1 oraz HTTP/2



3. Klient ma możliwość ustawienia priorytetu dla zwracanych przez serwer danych poprzez ustawianie wag dla każdego z elementów strony [9],

4. Nagłówki wiadomości są kompresowane. Do serwera wysyłane są tylko części unikalne. Cała reszta nagłówka jest odrzucana [9],

5. Serwer ma możliwość wypychania danych zanim klient o nie zapyta [9].

2.2.2 REST

Pojęcie REST (*(Representational State Transfer)*) pojawiło się w roku 2000 w pracy doktorskiej Roya Fieldinga [15] i oznacza styl architektoniczny dla systemów rozproszonych. Usługa REST będąc oparta na protokole HTTP wykorzystuje udostępniane przez niego metody (*GET, POST, PUT, DELETE*) do manipulacji na zdalnych zasobach do których dostęp jest wskazywany przy użyciu unikalnych wzorców identyfikatorów URI (*Uniform Resource Identifier*). Metody te odwzorowują funkcje **create (utwórz)**, **read (odczytaj)**, **update (aktualizuj)** oraz **delete (usuń)** znane pod kryptonimem *CRUD*. W swojej pracy Fielding nie tylko przedstawił definicję usługi REST, ale także zapisał wiele wytycznych, które służyły przyszłym projektantom przy budowie swoich serwisów. Zbiór tych reguł wraz z późniejszymi pracami rozwijającymi tą dziedzinę kryje się pod pojęciem **HATEOAS (*Hypermedia As The Engine Of Application State*)**. Interfejs API, który jest zbudowany z wykorzystaniem zasad HATEOAS wymaga od klienta wykrycia funkcji jakie udostępnia. Klient który chciałby wykorzystać interfejs musi wykonać zapytanie GET na głównym identyfikatorze URI. W ten sposób otrzyma szczegółową listę identyfikatorów, które będą mogły zostać wykorzystane do obsługi innych operacji. Identyfikatory te nie mogą być przechowywane przez klienta wymuszając dynamiczną analizę. Wymaga to od programisty po stronie klienta zaimplementowanie mechanizmów, które będą umożliwiały dostosowanie się do zmian po stronie serwera. Reguła ta umożliwia projektantom interfejsów dowolną ich modyfikację bez obawy, że po stronie klienta wystąpią problemy z ich użytkowaniem. Pomimo faktu, iż rozwiązanie to przekłada się na lepszą skalowalność oraz rozszerzalność klientów oraz serwerów upraszczało pracę tylko projektantom po stronie serwera. Programiści po stronie klienta musieli pilnować, aby nie umieścić identyfikatora na stałe co przy rosnącym poziome skomplikowania API prowadziło do wielu pomyłek. Tym samym narodził się bardziej pragmatyczny wariant REST. Pozostawiono najlepsze cechy starszej specyfikacji wprowadzając znaczne ułatwienia dla programistów po stronie klienta. Poniższe zasady mają za zadanie uporządkować standard REST tak, aby ułatwić integrację usług [7]:

- działania na pojedynczych obiektach powinny korzystać z wszystkich dostępnych metod udostępnionych przez HTTP. Do tej pory najczęściej wykorzystywana była metoda PUT,
- należy stosować kody powrotu dla odpowiedzi serwera,
- programista po stronie klienta powinien jakiego formatu danych ma się spodziewać wywołując metodę,
- każda kolejna wersja API powinna zawierać w identyfikatorze URI numer wersji,
- identyfikatory URI powinny być zaprojektowane według ustalonego wzorca,
- nagłówki HTTP powinny zawierać jedynie wymagane informacje

Tablica 2.1: Przykład użycia metod HTTP wraz z opisem operacji oraz identyfikatorami URI

Operacja	Metoda	Identyfikator URI
Umieszczenie produktu w koszyku	POST	`http://api.v1.eshop/bucket/bucketName`
Wyświetlenie zawartości koszyka	GET	`http://api.v1.eshop/bucket/bucketName`
Pobranie produktu z koszyka	GET	`http://api.v1.eshop/bucket/bucketName/product/productName`
Zastępowanie produktu w koszyku	PUT	`http://api.v1.eshop/bucket/bucketName/product/productName`
Usuwanie produktu z koszyka	DELETE	`http://api.v1.eshop/bucket/bucketName/product/productName`
Usuwanie całego koszyka	DELETE	`http://api.v1.eshop/bucket/bucketName`

W swojej pierwotnej formie REST był zaprojektowany do wymiany informacji przy użyciu protokołu XML. Format ten posiada olbrzymie możliwości jeśli bierze się pod uwagę tworzenie skomplikowanych obiektów z zagwarantowaną spójnością. Biorąc pod uwagę, że usługi REST charakteryzują się prostotą oraz lekkością pobieranie znacznych ilości danych w postaci XML stało się niewygodne. Godnym następcą okazał się format **JSON** ***JavaScript Object Notation***. Format ten został opracowany przez Douglasa Crockforda, który wykorzystał elementy języka JavaScript budując tym samym lekki oraz prosty język definicji danych. Największą zaletą formatu JSON jest możliwości translacji bezpośrednio w używanych językach programowania, bez konieczności konwersji na obiekty tak jak ma to miejsce w przypadku formatu XML. Format XML wymusza również implementację mechanizmów analizowania składni ze względu na implementację atrybutów, przestrzeni nazw lub wariantów kodowania tekstu.

Listing 2.1: Dane zapisane w formacie XML

```
<?xml version="1.0" encoding="iso-8859-1"?>
<users>
  <user>
    <firstname>Paul</firstname>
    <surname>Sajnog</surname>
    <address>Wolczanska 12</address>
    <city>Lodz</city>
    <country>Poland</country>
    <contact>
      <phone>666 777 000</phone>
      <email>167686@edu.p.lodz.pl</email>
    </contact>
  </user>
</users>
```

Listing 2.2: Dane zapisane w formacie JSON

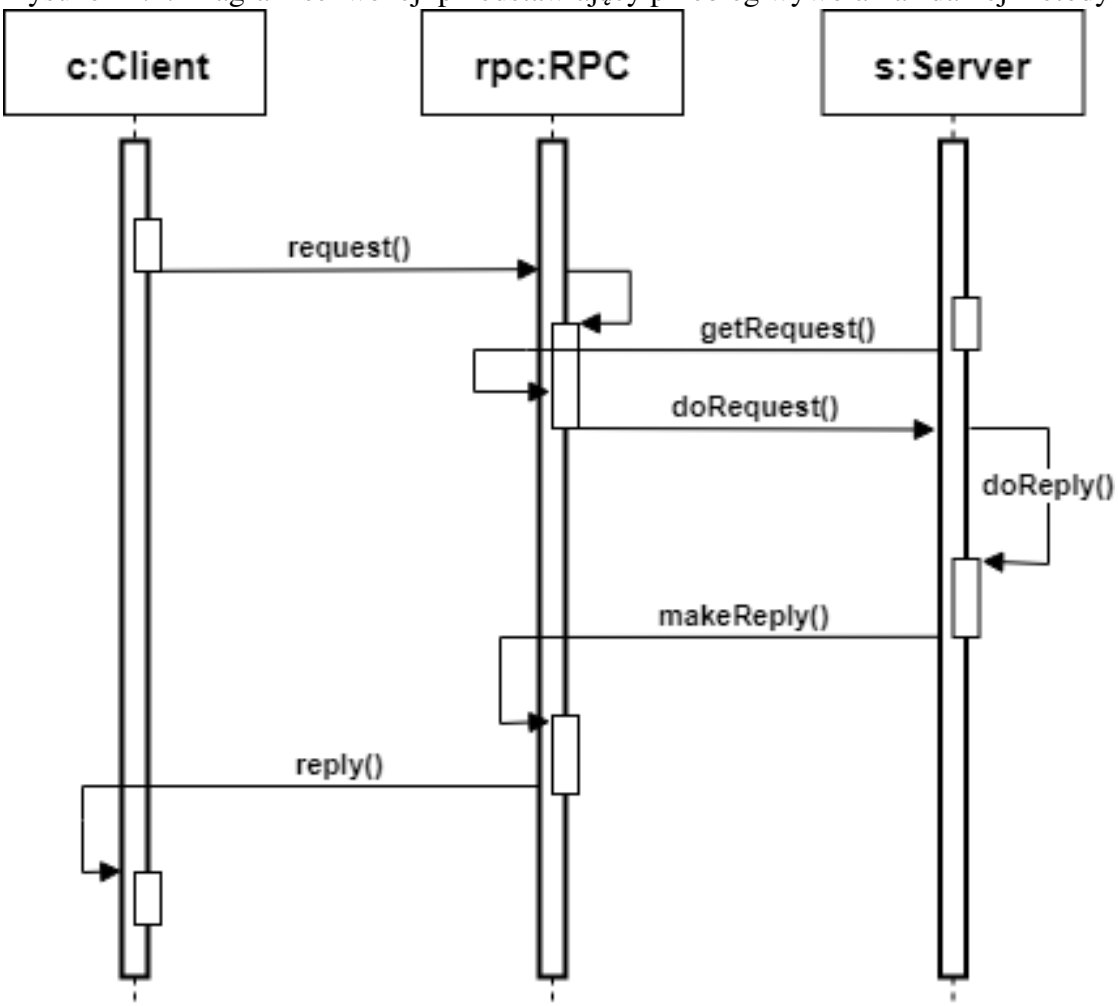
```
{
    "users": {
        "user": {
            "firstname": "Paul",
            "surname": "Sajnog",
            "address": "Wolczanska 12",
            "city": "Lodz",
            "country": "Poland",
            "contact": {
                "phone": "666 777 000",
                "email": "167686@edu.p.lodz.pl"
            }
        }
    }
}
```

2.2.3 RPC

Koncepcja RPC (*Remote Procedure Call*) jest znacznie starsza niż wcześniej przedstawiona REST. Już w roku 1981 dokładnie została opisana w publikacji inżynierów pracujących w laboratoriach Xerox [10]. Architektura takiego rozwiązania opiera się na modelu klient-serwer. W momencie gdy zachodzi potrzeba wywołania zdalnej program kliencki przekazuje parametry do metody lokalnej tak jak się to odbywa w przypadku normalnych aplikacji. W tym momencie bez jakiejkolwiek wiedzy po stronie klienta na temat połącznia sieciowego środowisko uruchomieniowe RPC przekazuje dane do usługi uruchomionej po stronie serwera, który znajduje się w innej adresacji sieciowej. Gdy już się zakończy przetwarzanie żądania serwer zwraca tą samą drogą odpowiedź z wynikiem. Cały proces przebiega w sposób synchroniczny więc klient oczekuje do czasu, aż serwer zwróci mu odpowiedź. Główną ideą RPC jest ukrywanie wszelkich informacji na temat sieciowej warstwy transportowej oraz mechanizmów serializacji oraz deserializacji przesyłanych danych. Ma to na celu odciążyć programistów, którzy nie muszą już implementować kodu odpowiedzialnego za przygotowanie danych do wysyłki oraz podejmowania decyzji o doborze warstwy transportowej.

W większości technologii opartych na protokole RPC proces tworzenia usługi RPC rozpoczyna się od zdefiniowania udostępnionych interfejsów w specjalnym języku **IDL *Interface Description Language***. W przypadku technologii XML-RPC oraz JSON-RPC funkcjonalność tą można uzyskać za pomocą dodatkowych bibliotek. Język ten nie ma ustalonego standardu dla wszystkich rozwiązań dostępnych na rynku. W zależności od użytej technologii różni się on składnią oraz definiowaniem konkretnych abstrakcji danych używanych w procesie wymiany pomiędzy serwerem oraz klientem. Następnie z tak przygotowanych definicji wygenerowane zostają specjalne części kodu(*ang. stubs*) osobno dla klienta i serwera. Każda ze stron implementuje wygenerowany kod. Po stronie serwera pozostaje dodać odpowiednią logikę, która zajmuje się przetwarzaniem odebranych danych. Różne rozwiązania stosują różne formaty danych w niektórych wypadkach jest to format binarny (Java RMI, Apache Thrift, gRPC) w innych XML lub JSON (SOAP, XML-RPC, JSON-RPC). Za warstwę transportową może posłużyć protokół TCP, UDP lub HTTP.

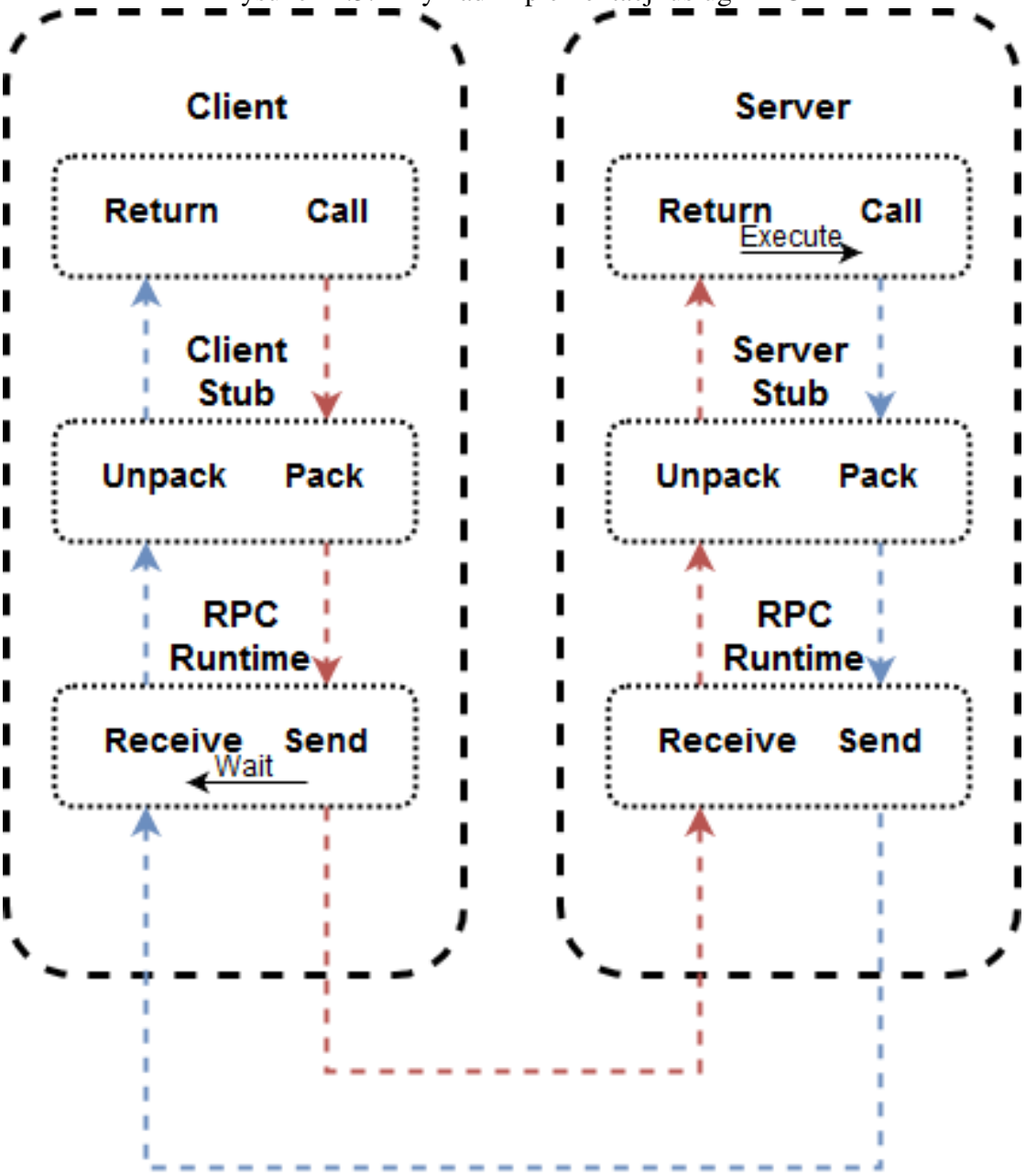
Rysunek 2.2: Diagram sekwencji przedstawiający przebieg wywołania zdalnej metody



2.3 Komunikacja synchroniczna i asynchroniczna

Gdy projektant rozpoczyna planowanie wdrożenia architektury mikroserwisowej musi podjąć decyzję, czy komunikacja pomiędzy mikroserwisami ma być synchroniczna, czy asynchroniczna. Decyzja ta ma olbrzymi wpływ na to jak będzie w przyszłości zachowywał się cały system. W podejściu synchronicznym klient po wysłaniu żądania, będzie oczekiwał na odpowiedź serwera. Asynchroniczna komunikacja umożliwia klientowi wykonywanie innych działań po wysłaniu żądania. Oba podejścia niejako wymuszają różne modele współpracy. Synchroniczny model komunikacji jest dużo łatwiejszy do zamodelowania jednak nie sprawdzi się w przypadku, gdy zadania do wykonania są długotrwałe lub gdy oczekuje się niewielkich opóźnień ze względu na swoją blokującą naturę. Różne tryby komunikacji mają wpływ na styl współpracy między mikroserwisami. Architektura oparta na stylu żądanie-odpowiedź działa w klasyczny sposób, gdzie to klient inicjuje żądanie i czeka na odpowiedź serwera. Innym stylem jest architektura oparta na zdarzeniach. W tym wypadku klient generuje zdarzenie, które zostaje zarejestrowane przez magistralę zdarzeń (np. RabbitMQ), a następnie przekierowane do odpowiedniej usługi. Tutaj klient nie jest powiadamiany, gdzie trafi jego żądanie.

Rysunek 2.3: Przykład implementacji usługi RPC

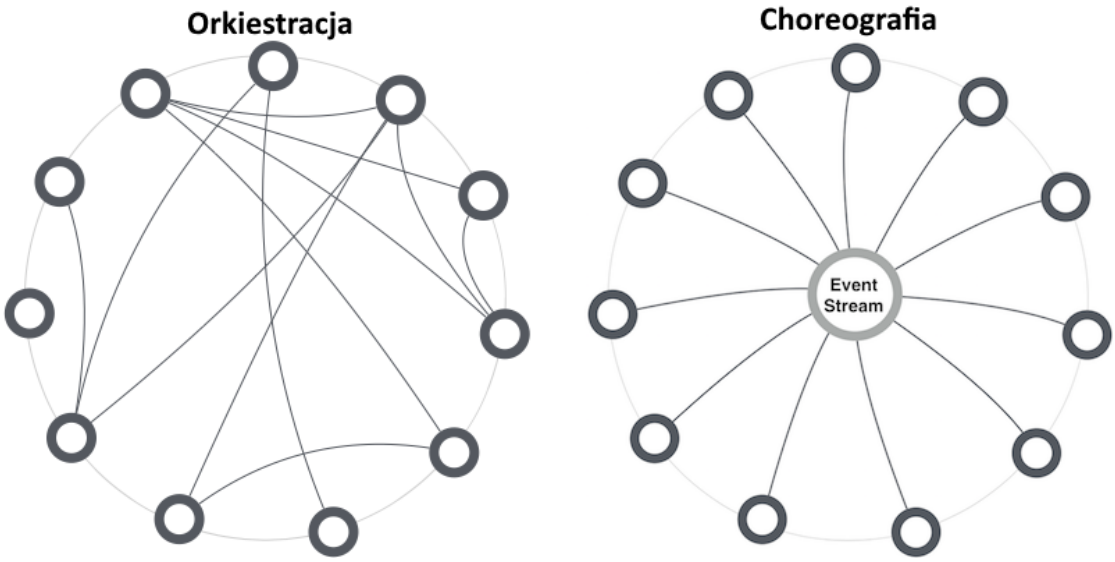


2.3.1 Orkiestracja i choreografia

W momencie, gdy procesy biznesowe wychodzą poza granicę pojedynczej usługi pojawia się problem zarządzania nimi. Szczególnie podatne na te zagrożenia są mikroserwisy. Aby temu zaradzić zaproponowano dwa różne podejścia. W pierwszym specjalnie do tego powołany koordynator aranżuje wysyłanie wielu żądań do odpowiednich usług oczekując na odpowiedź. Wzorzec oparty na choreografii pozwala usłudze klienckiej na wyemitowanie danego zdarzenia, które trafia do magistrali zdarzeń. Tam jest subskrybowane przez każdy dołączony serwis, który wykonuje swoje działania gdy zachodzi taka potrzeba. Pierwsze podejście opiera swoją komunikację na wywołaniach synchronicznych z każdym serwisem, który jest zaangażowany w przetwarzanie żądania. Dodatkowo zabierana jest w pewien sposób autonomiczność każ-

dego z mikroserwisów, które są uzależnione od poleceń koordynatora. Podejście oparte na choreografii pozwala działać w sposób asynchroniczny, gdyż każda usługa działa niezależnie wykonując swoje zadanie.

Rysunek 2.4: Porównanie grafów zależności dla orkiestracji oraz choreografii w architekturze mikroserwisowej



3 Przegląd istniejących rozwiązań

3.1 Wstęp

Zarówno technologia REST jak i RPC nie są czymś nowy w świecie wymiany informacji pomiędzy rozproszonymi systemami informatycznymi. Na rynku istnieje wiele lepszych lub gorszych rozwiązań, które można użyć przy projektowaniu i budowie architektury mikroserwisowej. Rozdział ten ma na celu przybliżenie popularnych rozwiązań dostępnych w chwili tworzenia pracy.

3.2 Rozwiązania oparte na technologii REST

3.2.1 Spark Java

Powstała w roku 2014 biblioteka dedykowana językom Java oraz Kotlin. Założeniem autora było stworzenie technologii, która ułatwiałaby budowanie serwisów sieciowych bez zbędnego narzutu dodatkowych modułów tak jak ma to miejsce w rozwiązaniach typu Spring lub Jersey. W skład biblioteki wchodzą klasy odpowiedzialne za routing, ciasteczka, sesje, filtrowanie, obsługę błędów itp. Niewielkie rozmiary oraz podstawowa obsługa żądań REST sprawia, że idealnie nadaje się do tworzenia niewielkich serwisów, a wykorzystanie dobrodziejstw jakie przynosi ósme wydanie języka Java (funkcje lambda) pozwala pisać przejrzysty i kompaktowy kod.

Listing 3.1: Przykład metody zwracającej tekst w Spark Java

```
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello␣World");
    }
}
```

3.2.2 Flask

Python jest jednym z języków, który jest polecany do tworzenia mikroserwisów ze względu na minimalizm oraz olbrzymią dostępność bibliotek wspomagających

budowę API. Przeglądając dostępne rozwiązania dla tego języka najczęściej można natrafić na projekty wykorzystujące bibliotekę Flask, która posiada wbudowany router oraz pozwala pisać aplikacje składające się z modułów za pomocą obiektów zwanych Blueprints [6]. Ponadto mamy do dyspozycji cały wachlarz modułów odpowiedzialnych za zarządzenie sesją, logowanie, uwierzytelnianie oraz obsługę baz danych.

Listing 3.2: Prosty model aplikacji z użyciem Flask

```
from flask import Flask, jsonify


# instantiate the app
app = Flask(__name__)


@app.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

3.2.3 HapiJS

Postanie platformy nodejs, która zbudowana jest na podstawie silnika popularnej przeglądarki *Chromium* zrewolucjonizowało świat aplikacji sieciowych. Pozwala on na budowanie asynchronicznego kodu serwerowego oraz aplikacji po stronie klienta w tym samym języku, czyli JavaScript. W chwili obecnej żadna inna technologia nie posiada tak wielu bibliotek oraz kompleksowych rozwiązań w swojej bazie. Jednym z popularniejszych jest stworzony w laboratoriach firmy Walmart HapiJS. W odróżnieniu od innych podobnych rozwiązań HapiJS dysponuje bogatą biblioteką wtyczek rozszerzających funkcjonalność[4] oraz co ważniejsze został gruntowanie przetestowany w środowisku produkcyjnym.

Listing 3.3: Przykład uruchomienia serwera http w HapiJS

```
'use strict';

const Hapi = require('hapi');

const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

server.route({
  method: 'GET',
  path: '/',
  handler: (request, h) => {
    return 'Hello, world!';
  }
});

server.route({
  method: 'GET',
```

```
  path: '/{name}',
  handler: (request, h) => {
    return 'Hello, ' + encodeURIComponent(request.params.name) + '!'
    }
});

const init = async () => {
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
};

process.on('unhandledRejection', (err) => {
  console.log(err);
  process.exit(1);
});

init();
```

3.2.4 ASP NET Core Web API

W roku 2016 Microsoft zaprezentował platformę net core. Jest to odświeżony, w pełni modułowy zestaw bibliotek wraz z środowiskiem służącym do jego uruchomienia przygotowany do współpracy z systemami z rodziny Windows jak i dostępnymi na systemy Linux oraz OSX. Modułowa budowa pozwala na pobranie lub wykorzystanie tylko tych zależności jakie są wymagane do uruchomienia aplikacji zmniejszając tym samym wielkość gotowej aplikacji. Programiści do wyboru mają między innymi biblioteki odpowiedzialne za obsługę baz danych, logowania zdarzeń, uwierzytelniania itp. Każda biblioteka dostępna jest jako oddzielny moduł w repozytorium NuGet, skąd bardzo łatwo można ją dołączyć do projektu rozszerzając[12] jego funkcjonalności.

Listing 3.4: Przykład definicji adresów URI dla metod http w kontrolerze

```
[HttpGet]
public ActionResult<List<TodoItem>> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public ActionResult<TodoItem> GetById(long id)
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return item;
}
```

3.3 Rozwiązania oparte na technologii RPC

3.3.1 CORBA

CORBA (*Common Object Request Broker Architecture*) jest ogólną specyfikacją zaproponowaną w roku 1991 przez stowarzyszenie OMG (*Object Management Group*). Definiuje ona standard komunikacji w systemach rozproszonych oparty o paradygmat obiektowy. Do podstawowych wymagań[3] systemów implementujących standard CORBA należą:

- **Orientacja obiektowa**-wszystkie zdalne operacje są grupowane w interfejsach podobnie jak ma to miejsce w obiektowych językach programowania takich jak C++ lub Java. Instancją każdego interfejsu jest tzw. obiekt CORBA. Jeśli klient chce wywołać zdalnie musi znać dokładne informacje o zdalnym obiekcie. W tym wypadku, każdy obiekt powinien mieć zdefiniowaną referencje do obiektu.
- **Przezroczystość lokalizacji**-aby ułatwić używanie obiektów, każdy system zbudowany według specyfikacji CORBA musi zapewnić, że nie istotne w jakiej lokalizacji znajduje się obiekt. Dostęp do każdego obiektu musi zostać zapewniony bez względu, czy znajduje się on w przestrzeni adresowej tej samej aplikacji, czy też w innej. Obiekt może znajdować się nawet na innej maszynie zdalnej.
- **Neutralność względem języka programowania**-CORBA jest zaprojektowana w ten sposób, aby kod klienta i serwera mógł zostać utworzony w różnych językach oprogramowania. W tym wypadku całkowicie normalną jest sytuacja, gdyż klient napisany w języku Java wywołuje procedurę w obiekcie zaimplementowanym w języku Cobol.
- **Obsługa mostów**-organizacja OMG miała świadomość, że równolegle prowadzone są prace nad alternatywnymi systemami rozproszonymi takimi jak Microsoft DCOM lub RMI. W związku z tym zaistniała potrzeba integracji z tymi systemami. W konsekwencji powstało w późniejszym czasie wiele projektów umożliwiających łączenie ww. technologii z architekturą CORBA.

Interfejsy opisujące każdy obiekt definiuje się za pomocą specjalnie do tego stworzonego języka IDL (*Interface Definition Language*). Jest to język czysto deklaratywny przeznaczony wyłącznie do definiowania struktury interfejsów. Na definicję obiektów przypadają używane typy danych, interfejsy oraz opisy operacji. Przestrzenie nazw (C++) oraz pakiety (Java) są definiowane za pomocą modułów. Każda sygnatura wywoływanej operacji (metody) zawiera:

- identyfikator (nazwę),
- typ zwracanej wartości,
- parametry wywołania (typ oraz kierunek),
- wyjątki

Listing 3.5: Przykład zapisu definicji interfejsu w języku IDL

```
interface CustomerAccount {
    string get_name();
    long   get_account_no();
    boolean deposit_money(in float amount);
    boolean transfer_money(
        in float amount,
        in long  destination_account_no,
        out long confirmation_no
    );
};
```

Komunikacja pomiędzy klientem, a serwerem była oparta na zdalnym wywołaniu procedur. Początkowo wykorzystywano do tego specjalnie przygotowany protokół GIOP (*General Inter-ORB Protocol*) niezależny od sieciowej warstwy transportowej, a od wersji 2.0 IIOP (*Internet Inter-ORB Protocol*) oparty na warstwie transportowej TCP/IP. Budowa oraz zasady działania CORBA są bardzo złożone. Poniżej wymienione zostały podstawowe koncepcje opisujące architekturę technologii:

- **Kompilatory IDL**-każdą aplikację budowaną według wytycznych CORBA rozpoczyna się od zdefiniowania określonych interfejsów z których zostanie wygenerowany kod dla klienta jak i serwera.
- **Mapowanie języków programowania**-specyfikacja języka IDL opisuje dokładnie każdy typ danych zdefiniowany w interfejsach wraz z mapowaniem do typów danych języków do których zostaną interfejsy skompilowane. W dokumentacji zawarty jest również sposób implementacji obiektów po stronie serwera.
- **Kod klienta oraz serwera**-wygenerowany kod klienta (*stub*) pozwala wywoływać zdalne operacje w obiektach CORBA w taki sposób jak ma to miejsce w obiektach lokalnych. Natomiast kod serwera (*skeleton*) jest nadzbiorem kodu klienta pozwalając aplikacji serwerowej na implementację obiektu CORBA.
- **Referencja obiektu**-każdy klient odwołujący się do obiektu CORBA musi posiadać referencje do tego obiektu. W przypadku języków Java oraz C++ obiekt zawierający wywoływane metody sam w sobie posiada referencję. W momencie, gdy klient wywołuje zdalną operację zostaje przekierowany przez referencję do odpowiedniej lokalizacji. Następnie po stronie serwera wywołanie zostaje odebrane i przekierowane do odpowiedniego obiektu CORBA zaimplementowanego w kodzie serwera.
- **Adapter obiektu**-adapter jest pośrednikiem pomiędzy implementacją obiektu, a szyną ORB (*Object Request Broker*). Odpowiada on za generowanie referencji do obiektów, wywołania metod oraz zapewnienie bezpieczeństwa w komunikacji. W wczesnych wersjach CORBA zdefiniowany był jako BOA (*Basic Object Adapter*) jednak był on niedokładnie zdefiniowany co skutkowało brakiem kompatybilności pomiędzy różnymi implementacjami szyny ORB. W wersji 2.2 oraz późniejszych dodano adapter POA (*Portable Object Adapter*) z uszczegółowioną specyfikacją oraz wieloma rozszerzeniami umożliwiając już w pełna przenośność.
- **Pośrednik Zleceń Obiektowych**-ORB jest zasadniczą częścią technologii CORBA. Jest to zbiór oprogramowania umożliwiającego komunikację pomiędzy obiektami w sieci. Pośrednik zapewnia mechanizm lokalizowania oraz aktywowania

zdalnych serwerów oraz uzyskuje referencję do obiektów w aplikacji. Ponadto odpowiada za komunikację do innego pośrednika.

CORBA pomimo bardzo bogatej funkcjonalności została wyparta przez inne technologie opisane w dalszej części niniejszej pracy. Duża złożoność implementacji, problemy z bezpieczeństwem oraz brak wersjonowania to główne powody zaniechania stosowania tej technologii w systemach budowanych po roku 2000. Ponadto CORBA całkowicie pomijała platformę .NET, która jest jedną z głównych stosowanych w oprogramowaniu korporacyjnym.

3.3.2 Protokół XML-RPC

W roku 1998 utworzono protokół XML-RPC, którego zadaniem była komunikacja w konfiguracji klient-serwer. Protokół ten jest kombinacją architektury RPC, języka znaczników XML oraz protokołu HTTP co powoduje, że do jego implementacji można użyć istniejącej infrastruktury sieciowej oraz dostępnych technologii przesyłania danych[8]. XML-RPC jest technologią całkowicie niezależną od zastosowanej architektury oraz języka programowania umożliwiając w pełni działanie w systemach heterogenicznych. Zasada działania protokołu polega na synchronicznym przesyłaniu pakietów danych pomiędzy klientem, a serwerem w postaci żądań oraz odpowiedzi HTTP tak samo jak robią to przeglądarki internetowe z tą różnicą, że dane są reprezentowane w formacie XML. Bezstanowa natura protokołu HTTP sprawia, że protokół XML-RPC również posiada tą właściwość. Powoduje ona, że dwa identyczne wywołania zdalnej metody jedno po drugim są traktowane jako nie związane niczym zdarzenia, a żadna z wartości jakie przekazują nie jest nigdzie zapisywana. Poniżej został przedstawiony proces wywołania zdalnej procedury oraz odpowiedzi przez serwer:

1. aplikacja za pomocą klienta XML-RPC wykonuje zdalne wywołanie metody określając nazwę metody, parametry wywołania oraz adres serwera,
2. klient XML-RPC umieszcza wszystkie przekazane dane w strukturze xml. Aplikacja wysyła tak przygotowaną strukturę jako żądanie POST na wskazany adres serwera,
3. serwer HTTP przyjmuje żądanie od klienta przekazując dane ze struktury XML do słuchacza XML-RPC,
4. słuchacz parsuje odebrane dane i wywołuje odpowiednią metodę wraz z parametrami,
5. wywołana metoda zwraca wynik operacji do procesu XML-RPC, który przetwarza je do struktury XML,
6. serwer zwraca tak przygotowaną odpowiedź do klienta,
7. klient XML-RPC parsuje wynik z struktury XML przekazując dane do aplikacji, która w dalszej kolejności kontynuuje ich przetwarzanie

Serwer jak i klient mogą zamienić się miejscami, tylko ważne jest aby zachowany był podział na serwer-klient. Protokół XML-RPC definiuje reprezentację podstawowych typów danych przekazywanych w żądaniach oraz zwracanych w odpowiedzi: Za pomocą

Tablica 3.1: Podstawowe typy danych i ich reprezentacja w strukturze XML-RPC

Typ danych	**Znacznik XML**
Liczby Całkowite	<int>400</int>
Liczby Zmiennoprzecinkowe	<double>2.0</double>
Zmienne logiczne	<boolean>1</boolean>
Typy łańcuchowe	<string>Hello, World!</string>
Data i czas	<dateTime.iso8601>19030223T00:30:00</dateTime.iso8601>
Typ binarny	<base64>UGF3ZcWCIFNham7Ds2c=</base64>

notacji xml można również zapisać bardziej skomplikowane typy danych jakimi są struktury oraz tablice składające się z typów prostych. Definicje struktur oraz tablic przedstawiają się następująco:

Listing 3.6: Reprezentacja tablicy w XML-RPC

```
<array>
  <data>
    <value> Some Value</value>
      ...
    <value> Some Value </value>
  </data>
</array>
```

Listing 3.7: Reprezentacja struktury w XML-RPC

```
<struct>
  <member>
    <name>Name-1</name>
    <value>Value-1</value>
  </member>
    ...
  <member>
    <name>Name-n</name>
    <value>Value-n</value>
  </member>
</struct>
```

Każde żądanie XML-RPC składa się z nagłówka oraz danych właściwych przestawionych w notacji xml. W nagłówku podobnie jak w zwykłych żądaniu http znajdują się informacje o nazwie serwera, rodzaju wywoływanej metody, ścieżką wywoływanego skryptu oraz typu danych.

Listing 3.8: Przykład żądania POST

```
POST /rpchandlefunc HTTP/1.0
User-Agent: MySystemXMLRPC/1.0
Host: xmlrpc.server.com
Content-Type: text/xml
Content-Length: 165

<?xml version="1.0"?>
<methodCall>
```

```
<methodName>getCapitalCity</methodName>
<params>
<param>
<value><string>England</string></value>
</param>
</params>
</methodCall>
```

W przypadku odpowiedzi zwracany jest wynik wywołania metody lub wyjątek z błędem w przypadku nie powodzenia w postaci struktury nazwanej fault. Każdy błąd posiada swój kod z zakresu od -32768 do -32000. W nagłówku znajduje się kod odpowiedzi http, adres serwerem, który został przekazany jako parametr User-Agent w żądaniu przez klienta. W języku Java protokół został rozszerzony dodatkowo o specjalny typ *<nil/>* reprezentujący znany w javie null.

Listing 3.9: Przykład odpowiedzi serwera

```
HTTP/1.1 200 OK
Date: Sun, 29 Apr 2001 12:08:58 GMT
Server: Apache/1.3.12 (Unix) Debian/GNU PHP/4.0.2
Connection: close
Content-Type: text/xml
Content-length: 133

<?xml version="1.0"?>
<methodResponse>
<params>
<param>
<value><string>Michigan</string></value>
</param>
</params>
</methodResponse>
```

Pomimo wielu zalet protokół XML-RPC posiada również kilka znaczących wad, które sprawiają, że wykorzystanie go w komunikacji pomiędzy mikroserwisami nie będzie najbardziej optymalnym rozwiązaniem:

- w związku z tym, że żądania oraz odpowiedzi są przekazywane w języku XML ich wielkość jest znacząca,
- protokół sam w sobie nie posiada ważnych mechanizmów zapewniających bezpieczeństwo,
- protokół http na którym bazuje XML-RPC nie jest wydajny,
- użytkownik nie może zdefiniować swoich typów, musi polegać na tych wbudowanych

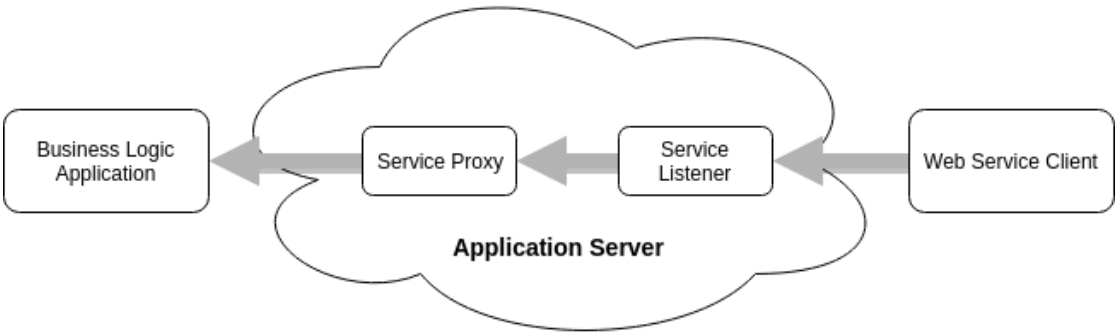
3.3.3 SOAP

Rosnąca na początku XXI wieku popularność systemów opartych na architekturze SOA (*Service Oriented Architecture*) spowodowała pojawienie się wielu rozwiązań opartych na modelu współpracujących ze sobą komponentów usługowych komunikujących

się za pomocą sieci Internet. Oprócz wspomnianych wcześniej technologii CORBA oraz DCOM pojawiło się również rozwiązanie o nazwie Web Services, które zdobyło bardzo dużą popularność. Każdego dnia miliony użytkowników na świecie korzysta z serwisów internetowych podczas zakupów w sklepach internetowych, zamawianiu biletów na środki komunikacji publicznej czy korzystając z komunikatora internetowego w telefonie komórkowym. Każdy komponent działający w technologii Web Service składa się z następujących części[14]:

- Aplikacji przetwarzającej logikę biznesową. W tym miejscu może to być koszyk przechowujący zakupy w sklepie internetowym, moduł cenowy itp.,
- Service Listener-serwis sieciowy nasłuchujący. Jego rolą jest odbieranie żądań przychodzących od klienta,
- Service Proxy-serwis dekodujący przychodzące żądanie z serwisu nasłuchującego przekazujący dane do aplikacji serwerowej,
- Serwer aplikacji-serwer HTTP służący jako kontener na serwisy

Rysunek 3.1: Ogólny zarys budowy typowej usługi sieciowej



Do najważniejszych rozwiązań wchodzących w skład technologii Web Services należą:

- **SOAP (*Simple Object Access Protocol*)**-protokół komunikacyjny służący do przekazywania zdalnych wywołań,
- **WSDL (*Web Service Description Language*)**-język oparty na XML służący do definiowania interfejsów usługi,
- **UDDI (*Universal Description, Discovery, and Integration*)**-rejestr wszystkich serwisów, które są dostępne zawierający ich metadane. Pobierając definicje z dokumentów WSDL służy do wyszukiwania udostępnionych usług, które mogą zainteresować potencjalnego konsumenta.

Protokół SOAP oparty na języku XML może korzystać z wielu dostępnych mechanizmów warstwy transportowej takich jak HTTP, HTTPS, SMTP lub JABBER. Wywoływanie usług następuje w dwóch trybach:

1. RPC (*Remote Procedure Call*)-w tym trybie XML reprezentuje listę parametrów wraz z wartościami, która zostaje przekazana komponentowi,
2. EDI (*Electronic Document Interchange*)-usługa otrzymuje tylko jeden parametr wywołania, którym jest cały dokument XML zawierający wszystkie dane składające się na encję biznesową.

Komunikaty SOAP zbudowane są ze znaczników XML. Najwyżej w hierarchii znajduje się znacznik **<Envelope>** oznaczający początek całego komunikatu. W dalszej kolejności znajduje się opcjonalny **<Header>** zawierający informacje nagłówkowe. Znacznik **<Body>** skupia wszystkie informacje o żądaniu oraz odpowiedzi. Uzupełnieniem powyższej listy jest znacznik **<Fault>** w którym znajdują się ewentualne błędy z kodami oraz opisami, jakie mogły pojawić się podczas przetwarzania żądania.

Listing 3.10: Przykład żądania SOAP

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="calculator">
      <m:multiply>
        <m:value_one>8</m:value_one>
        <m:value_two>5</m:value_two>
      </m:multiply>
    </soap:Body>
</soap:Envelope>
```

Powyższy listing przedstawia wywołanie metody *multiply*, która należy do komponentu *calculator*. Do wywołania metody są przekazane dwa parametry *value_one* oraz *value_two* z uzupełnionymi wartościami. W odpowiedzi serwer może zwrócić wynik tak jak przedstawia to listing poniżej.

Listing 3.11: Odpowiedź SOAP serwera na żadanie klienta

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="demo">
    <m:multiplyRes>
      <m:result>40</m:result>
    </m:multiplyRes>
  </soap:Body>
</soap:Envelope>
```

Wywołania usług zdalnych mogą przebiegać w sposób synchroniczny (protokół HTTP) lub asynchroniczny (JABBER, BEEP). Technologia SOAP jest bardzo złożona jednak bardzo popularna w rozwiązaniach korporacyjnych z uwagi na rozbudowany opis serwisów za pomocą WSDL oraz wykorzystanie znanego programistom protokołu XML.

3.3.4 Apache Thrift

Apache Thrift jest jedną z najnowszych technologii opartych na protokole RPC. Rozwiązanie to zostało opracowane w firmie Facebook w roku 2006, gdzie wykorzystywano je do komunikacji pomiędzy setkami serwisów tworzących ten portal

społecznościowy. Po roku zdecydowano się go przekazać fundacji Apache, gdzie stał się technologią open source. W dniu dzisiejszym wykorzystywany jest przy takich projektach jak Twitter, Pinterest lub Evernote. Podobnie jak wcześniej opisane technologie tak w tym przypadku wykorzystywany jest język IDL (*Interface Definition Language*) specjalnie przygotowany do współpracy z Apache Thrift.

Projekt powstał w czasie, gdy Facebook rozpoczął ekspansję na cały świat. Inżynierowie pracujący w projektach pobocznych zauważyli, że dotychczasowe rozwiązania oparte na zestawie LAMP nie są wystarczające[13]. Poszukiwania dostępnych rozwiązań nie przyniosły oczekiwanych efektów, gdyż na wielu płaszczyznach posiadały ograniczenia nie możliwe do zaakceptowania. W tym właśnie momencie podjęto decyzję o zaprojektowaniu od podstaw technologii, która nie tylko rozwiąże problemy z którymi nie poradziły sobie inne podobne projekty, ale również te nowe dopiero powstałe. Poniżej przedstawiono główne koncepcje Apache Thrift[11]:

- **Obsługa wielu typów danych**-oprócz podstawowych typów takich jak łańcuchy znaków lub liczby Thrift pozwala również definiować struktury, kolekcje (map, list, set), wyjątki oraz metody, które są w tym przypadku obiektami,
- **Warstwa transportu**-możliwość transmisji danych za pomocą różnych kanałów. Do dyspozycji zostają oddane gniazda sieciowe, protokół HTTP, systemy plików, obiekty w pamięci lub biblioteka zlib. Warstwa transportu jest całkowicie odseparowana od logiki co zapewnia bezproblemowe dopasowanie transmisji danych do danego rozwiązania.
- **Protokoły**-protokoły przygotowują dane do przesyłania. Apache Thrift zapewnia mechanizmy do automatycznej serializacji oraz deserializacji danych w postaci tekstu, zapisu binarnego, protokołu JSON itp. wszystko zależy od danego kontekstu. Za translację danych pomiędzy danymi protokołami odpowiada tak zwany *Procesor*.
- **Wersjonowanie**-zarządzenie zmianami w API jest bardzo trudnym wyzwaniem dla architektów. Niektóre zmiany są niewielkie i dotyczą tylko sygnatur pojedynczych metod, z drugiej strony zmiany mogą doprowadzić do przerwania kompatybilności wstecznej z poprzednią wersją. Jedną z funkcji Apache Thrift jest tzw. miękkie wersjonowanie. Oznacza to, że kolejne wersje API nie posiadają formalnych wymagań dla obsługi nowych lub zmienionych metod. W zamian programiści otrzymali narzędzia wspomagające utrzymywanie kompatybilności wstecznej:

  1. Argumenty metod są ponumerowane, nowa wersja metody może w pełni funkcjonować bez usuwania argumentów ze starszych wersji. Numery nadawane kolejnym wprowadzanym argumentom nie mogą się pokrywać z już przypisanymi. Nie wolno zmieniać numerów argumentów, które nie zostały usunięte.
  2. Każdemu nowemu argumentowi można przypisać domyślna wartość. Jest to pomocne, gdy klient operuje na starszej wersji, bez nowego pola i nie nadaje mu jakiejkolwiek wartości.
  3. Zmiana nazwy definicji metod lub serwisów jest zabroniona. Starsze wersje klientów nie będą mogły odnieść się do nazw, których nie znają.

- **Bezpieczeństwo**-w celu zabezpieczenia każdego z serwisów przed nie autoryzowanym dostępem Apache Thrift udostępnia obiekt *TSSLTransportFactory*, który zapewnia przechowywanie par kluczy RSA. Możliwe jest również tunelowanie SSH.
- **IDL *Interface Definition Language***-specjalnie przygotowany język opisu interfejsów w prosty sposób zapewnia programistom definiowanie serwisów. Definicja ta pozwala wygenerowanie kodu w każdym dostępnym języku o ile jest obsługiwany.

Listing 3.12: Przykład definicji metody w języku IDL

```
namespace py thrift.example1
namespace php thrift.example1

service SumService {
  i32 add(1: i32 a, 2: i32 b),
}
```

Przykład implementacji prostego serwisu w języku Java:

1. Aby móc skorzystać z Apache Thrift wymagana jest instalacja odpowiedniej biblioteki za pomocą, której możliwe będzie wygenerowanie kodu klienta oraz serwera.
2. W piku o rozszerzeniu *thrift* zostaje zapisana definicja serwisu z którego zostanie wygenerowany kod dla danego języka.

```
namespace java com.paul.thrift.service

exception MyException {
    1: i32 code,
    2: string exceptionDescription
}

struct Resource {
    1: i32 id,
    2: string name,
}

service MyService {

    Resource get(1:i32 id) throws (1:MyException ex),

    list <Resource> getResList() throws (1:MyException e),

}
```

3. Następny krok polega na wygenerowaniu kodu możliwego do implementacji zdefiniowanego serwisu. W linii komend należy wpisać następuje polecenie:
   $ thrift -r -out generated –gen java myservice.thrift Po jego wykonaniu w katalogu *generated*, który wskazano w poleceniu pojawią się następujące pliki z wygenerowanym kodem:
   - Resource.java,

- Service.java,
- MyException.java

4. W kolejnym kroku należy stworzyć implementację zdefiniowanego serwisu:

Listing 3.13: Implementacja kodu usługi

```
  public class MyServiceImpl implements MyService.Iface {

    @Override
    public Resource get(int id)
      throws MyException, TException {
        return new Resource();
    }

    @Override
    public List<Resource> getList()
      throws MyException, TException {
        return Collections.emptyList();
    }
}
```

5. Gdy już usługa jest zdefiniowana potrzeba serwera, który będzie nasłuchiwał na wybranym porcie. Przykładowy serwer korzysta z klasy *TThreadPoolServer*, która opakowuje pulę wątków ze standardowej biblioteki java, aby kod mógł być wykonywany asynchronicznie.

Listing 3.14: Kod serwera

```
public class MyServer {

  public static MyServiceImpl service;

  public static MyService.Processor processor;

  public static void myServer(MyService.Processor processor) {
    TServerTransport serverTransport = new TServerSocket(5000);
    TServer server = new TThreadPoolServer(new TThreadPoolServer.
    Args(serverTransport).processor(processor));
    server.serve();
  }

      public static void main(String [] args) {
        service = new MyServiceImpl();
        processor = new MyService.Processor(service);
        Runnable server = new Runnable() {
        public void run() {
          myserver(processor);
        }
      };
      new Thread(server).start();
    }
  }
```

6. Kod klienta można z powodzeniem zaimplementować jako moduł lub bibliotekę aplikacji, która będzie korzystać z usług zewnętrznego serwisu. W tym wypadku jest to niezależna aplikacja:

Listing 3.15: Kod klienta

```
public class MyClient {
  public static void main(String [] args) {

  TTransport transport = new TSocket("localhost", 5000);
  transport.open();

  TProtocol protocol = new TBinaryProtocol(transport);
  MyService.Client client = new MyService.Client(protocol);
      System.out.println(client.get());
      transport.close();
  }
}
```

3.3.5 gRPC-Google's Remote Protocol Call

gRPC jest rozwiązaniem opartym na RPC zaproponowanym przez firmę Google. Pomimo młodego wieku jak na technologie korporacyjne jest dobrze przetestowane produkcyjnie oraz wdrożone w wielu usługach wykorzystywanych wewnątrz tej firmy. Również tacy potentaci na rynku usług sieciowych jak Netflix oraz Cisco używają z powodzeniem tej technologii.

Zasada działania gRPC jest podobna do tej, którą wcześniej zaprezentowano w Apache Thrift. Dane wykorzystywane przez usługi oraz komunikaty są reprezentowane przez tzw. protobuf (ang. *protocol buffers*). Jako protokołu transportowego wykorzystano HTTP/2 będący następcą popularnego oraz szeroko stosownego protokołu HTTP/1.1. Protobuf są mechanizmem serializacji danych w postaci binarnej co powoduje, że zajmują znacznie mniej miejsca niż np. XML. Również procesy kodowania oraz dekodowania są znacznie szybsze, gdyż zasoby sprzętowe nie są zużywane na parsowanie danych tekstowych tak jak ma to miejsce w przypadku formatu XML lub JSON. Struktury danych wykorzystywane do tworzenia usług są definiowane za pomocą dedykowanego języka IDL w plikach o rozszerzeniu **.proto**. Następnie za pomocą specjalnego polecania wywoływanego np. z linii komend generowany jest kod serwera oraz klienta w wybranym języku programowania. W czasie pisania pracy wspieranych było 11 najpopularniejszych języków takich jak Java, Python czy C++. W buforach protokołu struktura danych składa się z dwóch podstawowych komponentów:

- Komunikatów (*messages*)-struktur przekazywanych oraz zwracanych podczas zdalnego wywołania metod składających się z pól o różnych typach danych,
- Usług (*services*)-struktur w których znajdują się definicje metod wywoływanych przez klientów

Pola w komunikatach zdefiniowane są według poniższego wzorca:

typ nazwa=pozycja

Typem może być każda wartość skalarna taka jak string, int, bool, float itp. W przypadku gdy pole musi być tablicą wartości przed typem należy użyć słowa kluczowego *repeated*. Istnieje również specjalny typ *enum*, którym można zdefiniować własny typ wyli-

czeniowy. Nazwa identyfikuje pole w sposób zrozumiały dla człowieka, natomiast pozycja to liczba określająca miejsce w strumieniu danych w którym znajduje się wartość pola. Pozycja jest istotną informacją, gdyż dzięki niej możliwe jest dekodowanie wartości kolejnych pól w strumieniu bajtów dokładnie w takiej kolejności jak zostały zdefiniowane w komunikacie. Restrykcja, która obowiązuje przy zachowaniu kolejności pól pozwala na bezproblemową zmianę nazw pól, gdy zostanie zachowana ich kolejność. W tym miejscu należy wspomnieć o jednej z najważniejszych decyzji projektowych protobuf, a mianowicie o możliwości dodawania kolejnych pól rozszerzających struktury komunikatów bez jakiejkolwiek wpływu na działanie poprzedniej już działającej wersji. Właściwość ta ułatwia to znacząco rozbudowę istniejącego API eliminując tym samym trudne w utrzymaniu wersjonowanie.

W bloku service odpowiadającym za definiowanie usługi znajdują się sygnatury metod wywoływanych przez klienta. Ogólny wzorzec przykładowej metody znajduje się poniżej:

rpc nazwa_metody (żądanie) returns (odpowiedź) {}

Słowo kluczowe *rpc* oznacza, że metoda jest przeznaczona do wywoływania zdalnego. Każda metoda posiada parę żądanie-odpowiedź, którymi są zdefiniowane wcześniej komunikaty. gRPC umożliwia cztery rodzaje komunikacji pomiędzy serwerem, a klientem:

- ***Unary***-jest to klasyczne podejście do wymiany informacji tak jak to ma miejsce w przypadku zwykłych żądań HTTP. Klient wysyła pojedyncze żądanie do serwera, który zwraca mu odpowiedź.
- ***Server streaming***- klient wysyłając pojedyncze żądanie w odpowiedzi otrzymuje sekwencję komunikatów. Komunikaty są wysyłane w sposób uporządkowany.
- ***Client streaming***- w tym wypadku to klient wysyła uporządkowaną sekwencję komunikatów, po czym serwer przetwarzając je zwraca jedną odpowiedź.
- ***Bidirectional streaming***- serwer jak i klient komunikują się ze sobą za pomocą strumieni. Oba strumienie działają niezależnie więc w zależności od potrzeb serwer może każdorazowo zwracać odpowiedź na pojedyncze żądanie klienta lub także zaczekać, aż klient prześle wszystkie komunikaty i dopiero wtedy odpowiedzieć.

Listing 3.16: Przykład definicji usługi w pliku proto

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.mycompany.geeting";

package greeting;

// The greeting service definition.
service Greet {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

```
// The request message containing the user's full name.
message GreetRequest {
  string first_name = 1;
  string last_name = 2;
}

// The response message containing the greetings
message GreetResponse {
  string greeting = 1;
}
```

4 Projekt aplikacji mikroserwisowej

4.1 Założenia aplikacji

Budowany projekt aplikacji z założenia ma symulować podstawowe funkcjonalności internetowej księgarni. Cały system został rozbity na niewielkie usługi z których każda wykonuje ściśle określone działanie w swojej domenie. Poniżej znajduje się lista dostępnych działań przewidzianych dla użytkownika.

- Utworzenie konta użytkownika korzystającego z sklepu,
- Dodanie książki do magazynu,
- Utworzenie zamówienia,
- Pobranie listy wszystkich zamówień dla danego użytkownika,
- Pobranie dostępnych książek znajdujących się w magazynie,
- Pobranie konkretnej pozycji z katalogu podając numer identyfikacyjny książki,
- Pobranie listy użytkowników posiadających konto wraz za datami utworzenia konta,

4.2 Diagram przypadków użycia

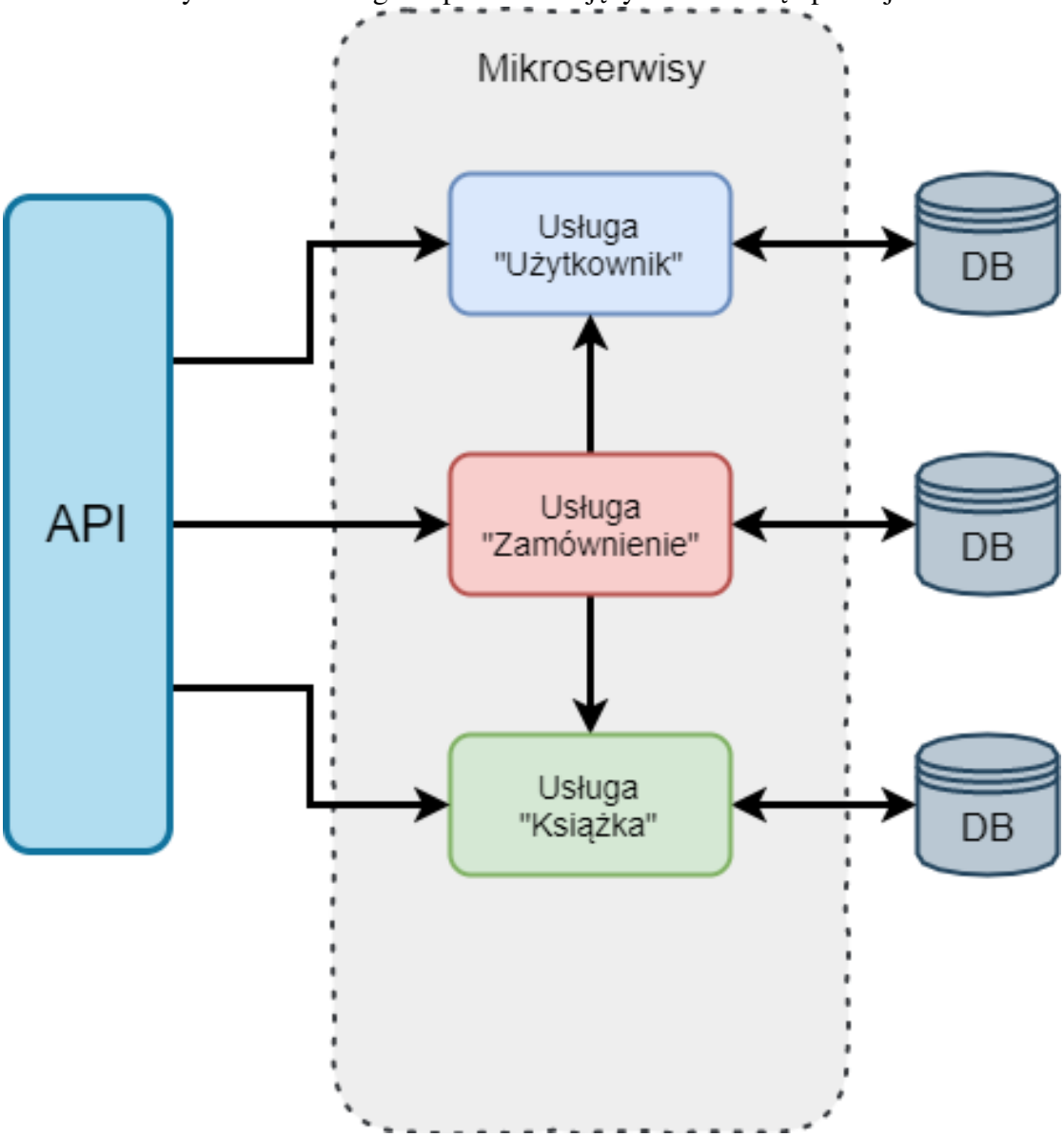
Diagram przedstawiony na rys. 4.1 przedstawia wszystkie możliwe czynności możliwe do wykonania przez system. Dwa przypadki stanowią bazę dla W tym przypadku usługa zamówienia wykorzystując usługi świadczone przez pozostałe mikroserwisy może wymieniać się potrzebnymi danymi nie mając dostępu do bazy danych, gdzie są przechowywane.

Rysunek 4.1: Diagram przypadków użycia

4.3 Schemat blokowy systemu

System opiera się na współpracujących ze sobą usługach sieciowych. Każda z tych usług działa niezależnie od reszty dzięki temu wdrożenie nowej wersji nie powoduje przerwania w działaniu innej. Dobra praktyka przy projektowaniu usług mikroserwisowych zaleca, aby każdy serwis korzystał z niezależnej bazy danych. Projekt uwzględnia i to podejście. Użytkownik do komunikowania z usługami będzie używał API, które będzie wystawiało wszystkie potrzebne metody na zewnątrz do użytku publicznego. Schemat poszczególnych modułów oraz powiązań został przedstawiony na rys. 4.2

Rysunek 4.2: Diagram przedstawiający architekturę aplikacji



5 Implementacja aplikacji

5.1 Wybór technologii oraz narzędzi

Podczas projektowania oraz budowy mikroserwisów jednym z największych wyzwań dla architektów jest odpowiedni dobór stosu technologicznego jaki zostanie wykorzystany przy budowie konkretnego rozwiązania. Natura systemów składających się z wielu współpracujących ze sobą usług powoduje, że jest to kluczowa decyzja, która będzie miała wpływ na powodzenie projektu. Mnogość technologii nie ułatwia w tym przypadku wyboru i trzeba dopasować je do konkretnego zadania realizowanego przez usługę. Równie ważnym aspektem, który jest często pomijany przez osoby odpowiedzialne za dobór technologii jest dopasowanie użytych rozwiązań do kompetencji posiadanych przez pracowników zespołów programistycznych. Nie można użyć języka Ruby do budowania usługi jeśli większość zespołu posiada umiejętności programowania np. w języku Java. Nie mniej istotny jest sposób przechowywania danych przez każdą z usług. W systemach, które udostępniają rodzaj sieci społecznościowej bardzo pomocna może okazać się grafowa baza danych(np. ArangoDB). W innym przypadku, gdy system operuje na znacznych ilościach dokumentów można brać pod uwagę wykorzystania dokumentowej bazy danych(np. MongoDB).

Do realizacji projektu systemu wykorzystano języki Golang oraz C#. Język Go jest młodym językiem stworzonym w firmie Google. Cechując się prostotą oraz możliwością natywnej kompilacji praktycznie na każdą dostępną platformę świetnie nadaje się do budowy mikroserwisów. Moduł API w całości został stworzony na platformie net core. Wybór różnych języków jest podyktowany przedstawieniem w jaki sposób usługi współdziałają ze sobą w heterogenicznym środowisku. Do przechowywania danych został wybrany silnik PostgreSQL niemniej nic nie stoi na przeszkodzie aby wykorzystać inny system zarządzania niekoniecznie RDBMS. Całość została uruchomiona w środowisku Windows. Prace programistyczne wykonano przy pomocy produktów firmy JetBrains: Goland oraz Rider. Program HeidiSQL posłużył do zarządzania bazami danych.

5.2 Architektura serwisów

Usługi zostały stworzone jako osobne projekty. Każdy korzysta z osobnej bazy danych. Warstwa domenowa oraz obsługi bazy danych zostały zaimplementowane niezależnie od warstwy transportu umożliwiając wykorzystanie innych protokołów np.

ampq. W związku z tym, każda usługa ma zaimplementowany serwer http oraz gRPC i tylko od klienta zależy który zostanie użyty do komunikacji. Serwery nasłuchują na portach podanych w pliku konfiguracyjnym listing 5.1. W przypadku usług napisanych w języku Golang serwer http został stworzony przy pomocy szkieletu Gin. Pomimo, że biblioteka standardowa tego języka oferuje bardzo dużo modułów umożliwiających tworzenie aplikacji sieciowych to szkielet aplikacji Gin uprasza wdrażanie punktów końcowych dodatkowo oferując znakomity zbiór oprogramowania pośredniego**(middleware)** umożliwiającego rozszerzenie tworzonego API. Moduł API stanowi projekt konsolowy utworzony w technologii net core, który implementuje wszystkie usługi dostępnych klientów HTTP oraz RPC. W tym przypadku tak samo jak dla mikroserwisów można wykorzystać dowolną technologię obsługującą protokoły http oraz gRPC.

Listing 5.1: Przykład konfiguracji usługi

```
database:
  connectionstring: "host=localhost port=5432 user=paul password=*****
  dbname=user_db sslmode=disable"

serverGrpc:
  port: 7001
serverHttp:
  port: 9090
```

5.2.1 Sposoby komunikacji

Wszystkie serwisy komunikują się pomiędzy sobą na dwa sposoby. W przypadku wykorzystania podejścia REST serwery wystawiają swoje usługi w sieci do których dostęp jest za pomocą metod http w połączeniu z odpowiednimi punktami końcowymi opisanymi za pomocą adresów URI. Natomiast usługi klienckie konsumują je zwracając, bądź nie odpowiedź. Komunikacja pomiędzy usługami w szkielecie komunikacji gRPC oparta jest na implementacji struktur danych komunikatów oraz metod, które służą do ich przekazywania. Dzięki temu możliwe jest wygenerowanie klas oraz metod dla każdego języka obsługiwanego przez szkielet gRPC. Za pomocą wygenerowanego kodu możliwe jest zaimplementowanie

5.2.2 Opis struktur oraz metod wykorzystanych w usługach

1. Usługa użytkownik:

    - **CreateUser**-metoda za pomocą której dodamy do bazy danych nowego użytkownika. W żądaniu przyjmuje nazwę użytkownika zwracając unikalny identyfikator ID oraz czas utworzenia,
    - **GetUserByID**-metoda zwracająca użytkownika o podanym w żądaniu identyfikatorze ID,
    - **GetUsersList**-zwraca listę wszystkich użytkowników. W żądaniu należy przesłać pusty komunikat.

2. Usługa katalog:

    - **CreateBook**-metoda służąca do dodania nowej książki. W żądaniu przekazuje się numer isbn, tytuł, autora oraz cenę. Metoda zwraca strukturę książki wraz z identyfikatorem,
    - **GetBookByID**-metoda zwracająca książkę o podanym w żądaniu identyfikatorze ID,
    - **GetBooksList**-zwraca listę wszystkich książek z katalogu. W żądaniu należy przesłać pusty komunikat.

3. Usługa zamówienie:

    - **CreateOrder**-metoda serwisu tworzy nowe zamówienia dla użytkownika. W żądaniu należy podać identyfikator użytkownika dla którego będzie utworzone zamówienie oraz zbiór zmówionych książek(identyfikator ID) wraz z ilością dla każdej pozycji.
    - **GetOrdersForUser**- metoda zwracająca całą historię zamówień dla użytkownika. Metoda w żądaniu przyjmuje identyfikator użytkownika zwracając rekord zamówienia wraz z wszystkimi pozycjami książek.

W przypadku wykorzystania do komunikacji usług REST wymagane było stworzenie odpowiednich punktów końcowych z specjalnie przygotowanymi adresami URI. **V1** oznacza wersję wstawionego API, w przyszłości gdyby zaszła konieczność zmian można podnieść jego wersję.

Tablica 5.1: Opis identyfikatorów URI dla metod

Operacja	**Metoda**	**Identyfikator URI**
CreateUser	POST	`http://localhost:PORT/v1/user/:name`
GetUserByID	GET	`http://localhost:PORT/v1/user/:id`
GetUsersList	GET	`http://localhost:PORT/v1/users`
CreateBook	POST	`http://localhost:PORT/v1/book`
GetBookByID	GET	`http://localhost:PORT/v1/book/:id`
GetBooksList	GET	`http://localhost:PORT/v1/books`
CreateOrder	POST	`http://localhost:PORT/v1/order`
GetOrdersForUser	GET	`http://localhost:PORT/v1/order/:id`

Na listingach 5.4 - 5.6 zostały przedstawione wszystkie dane potrzebne do wygenerowania kodu serwera i klienta. W każdym z plików proto jako opcjonalnie zdefiniowano nazwy pakietu dla języka Go oraz przestrzeni nazw dla C#. Zabieg ten ma na celu wyeliminowanie pomyłek podczas importu określonych klas oraz struktur danych. Bardzo ważną rzeczą jest to aby klient oraz serwer korzystały z identycznych definicji. Nawet niewielka zmiana w kolejności pól komunikatów spowoduje, że komunikacja

pomiędzy serwerem, a klientem nie dojdzie do skutku. Usługa *„Zamówienie”* pełni tutaj rolę serwera jak i klienta, gdyż korzysta z pozostałych serwisów w celu pobrania danych. Specyfika takiego rozwiązania wymusza na programistach implementację części klienckiej pozostałych usług. Podsumowując usługa ta posiada dostęp do danych z wszystkich struktur danych zawartych w plikach proto. Aby wygenerować kod dla klienta oraz serwera dla języka go w systemie operacyjnym muszą zostać zainstalowany kompilator protobuf oraz plugin dla języka Go umożliwiający wygenerowanie metod dla usług RPC. Następnie należy użyć polecenia w katalogu w którym znajduje się dany plik proto:

Listing 5.2: Polecenie do generowania kodu z plików proto dla języka Go

```
protoc --go_out=plugins=grpc:. user.proto
```

W przypadku języka C# muszą zostać dodane pakiety NuGet Grpc Grpc.Tools oraz Google.Protobuf. Kolejnym krokiem jest edycja pliku projektu csproj.xml i dodanie na poniższej pozycji:

Listing 5.3: Modyfikacjie pliku projektu net core

```
<ItemGroup>
  <Protobuf Include="**/*.proto" OutputDir="%(RelativePath)"
  CompileOutputs="false" />
</ItemGroup>
```

Parametr *Include* mówi kompilatorowi gdzie znajdują się pliki proto natomiast *OutputDir* wskazuje miejsce, gdzie pojawią się wygenerowane klasy.

Listing 5.4: Definicje usług i komunikatów dla usługi katalogu książek

```
syntax = "proto3";

package pb;

option go_package="catalogpb";
option csharp_namespace="catalogpb";

service CatalogService {
  rpc CreateBook (CreateBookRequest) returns (CreateBookResponse) {}

  rpc GetBookByID (GetBookRequest) returns (GetBookResponse) {}

  rpc GetBooksList (GetBooksRequest) returns (GetBooksResponse) {}
}

message Book {
    string id = 1;
    string isbn = 2;
    string title = 3;
    string author = 4;
    double price = 5;
}

message CreateBookRequest {
    string isbn = 1;
    string title = 2;
    string author = 3;
```

```
    double price = 4;
}

message CreateBookResponse {
    Book book = 1;
}

message GetBookRequest {
    string id = 1;
}

message GetBookResponse {
    Book book = 1;
}

message GetBooksRequest {
}
message GetBooksResponse {
    repeated Book books = 1;
}
```

Listing 5.5: Definicje usług i komunikatów dla usługi zamówień

```
syntax = "proto3";

package pb;

option go_package = "orderpb";
option csharp_namespace = "orderpb";

service OrderService {
 rpc CreateOrder (CreateOrderRequest)
 returns (CreateOrderResponse) {}

 rpc GetOrdersForUser (GetOrdersForUserRequest)
 returns (GetOrdersForUserResponse) {}
}
message Order {
    string id = 1;
    bytes createdAt = 2;
    string userId = 3;
    double totalPrice = 4;
    message OrderBook {
        string id = 1;
        string isbn = 2;
        string title = 3;
        string author = 4;
        double price = 5;
        uint32 quantity = 6;
    }
    repeated OrderBook books = 5;
}
message CreateOrderRequest {
    message OrderBook {
        string book_id = 2;
        uint32 quantity = 3;
    }
    string user_id = 2;
    repeated OrderBook books = 4;
}
```

```
message CreateOrderResponse {
    Order order = 1;
}

message GetOrderRequest {
    string id = 1;
}

message GetOrderResponse {
    Order order = 1;
}

message GetOrdersForUserRequest {
    string user_id = 1;
}

message GetOrdersForUserResponse {
    repeated Order orders = 1;
}
```

Listing 5.6: Definicje usług i komunikatów dla usługi użytkownicy

```
syntax = "proto3";

package pb;

option go_package = "userpb";

service UserService {
  rpc CreateUser (CreateUserRequest) returns (CreateUserResponse){}
  rpc GetUserByID (UserRequest) returns (UserResponse){}
  rpc GetUsersList (UsersListRequest) returns (UsersListResponse){}
}
message User {
    string id = 1;
    string user_name = 2;
    bytes created_at = 3;

}
message CreateUserRequest {
    string user_name = 1;
}
message CreateUserResponse {
    User user = 1;
}
message GetUserRequest {
    string id = 1;
}
message GetUserResponse {
    User user = 1;
}
message GetUsersListRequest {
}
message GetUsersListResponse {
    repeated User users = 1;
}
```

5.3 Bezpieczeństwo mikroserwisów

5.3.1 Protokół SSL/TLS

W każdym systemie opartym na architekturze mikroserwisowej niebagatelne znaczenie ma bezpieczeństwo. Z uwagi na fakt, że poszczególne usługi do wymiany informacji pomiędzy sobą wykorzystują połączenia sieciowe należy zadbać, aby dane te nie mogły być odczytane przez niepowołane osoby. Jedną z najpopularniejszych technik szyfrowania danych w sieci jest wykorzystanie protokołów bezpiecznej komunikacji SSL/TLS.

Budowę bezpiecznego tunelu komunikacji sieciowej rozpoczyna się od wygenerowania silnego klucza prywatnego dla serwera, który zostaje podpisany certyfikatem wydanym przez instytucję autoryzacyjną(CA - Certificate Authority). Certyfikat taki jest wydawany czasowo. W chwili obecnej klucze są generowane w oparciu o algorytm RSA o wielkości 2048-bitów. Oczywiście można zapewnić jeszcze większe bezpieczeństwo używając kluczu 4096-bitowych jednak wiąże się to ze zwiększonym zapotrzebowaniem zasobów obliczeniowych oraz pamięci. W rozdziale tym wykorzystana zostanie biblioteka OpenSSL w celu wygenerowania oraz podpisania klucza prywatnego serwera usługi sieciowej.

5.3.2 Generowanie certyfikatu SSL

W poniższych punktach przedstawiono kolejne kroki generowania kluczy dla klienta oraz serwera wraz z certyfikatami:

1. Generowanie klucza dla CA:

```
openssl genrsa -passout pass:1111 -des3 -out ca.key 4096
```

2. Generowanie Certyfikatu dla CA:

```
openssl req -passin pass:1111 -new -x509 -days 365
    -key ca.key -out ca.crt -subj "/CN=MyLocalCA"
```

3. Generowanie klucza dla serwera:

```
openssl genrsa -passout pass:1111 -des3 -out server.key 4096
```

4. Generowanie certyfikatu dla klucza:

```
openssl req -passin pass:1111 -new -key server.key -out
    server.csr -subj "/CN=%COMPUTERNAME%"
```

5. Podpisanie klucza dla serwera

```
openssl x509 -req -passin pass:1111 -days 365 -in server.csr
 -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt
```

6. Generowanie klucza dla klienta:

```
openssl genrsa -passout pass:1111 -des3 -out client.key 4096
```

7. Generowanie podpisu klucza dla klienta:

```
openssl req -passin pass:1111 -new -key client.key -out
    client.csr -subj  "/CN=%CLIENT-COMPUTERNAME%"
```

8. Podpisanie klucza dla klienta:

```
openssl x509 -passin pass:1111 -req -days 365 -in client.csr
 -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt
```

5.3.3 Uruchomienie serwera HTTP z obsługą SSL TLS

Szkielet aplikacji Gin pozwala na prostą operację wczytania kluczy i uruchomienia nasłuchiwania na protokole HTTPS. Wystarczy wywoałać funckję *RunTLS* i w parametrach wywołania podać klucz oraz certyfikat:

Listing 5.7: Dodanie obsługi połączenia szyfrowanego do serwera HTTP

```
func NewHttpServer(port int, service Service) {
  certFile := "certs/server.crt"
  keyFile := "certs/server.key"
  r := SetupRouter(service)
  log.Fatal(r.RunTLS(fmt.Sprintf(":%d", port),certFile,keyFile))
}
```

Po uruchomieniu serwera w trybie debug na konsoli zostanie wypisana odpowiednia informacja:

```
2019/08/22 18:37:47 Server HTTP Running on port 9090
```

Klient próbując wywołać metodę używając do tego protokołu http otrzyma poniższą informację:

```
Client sent an HTTP request to an HTTPS server.
```

Od tej pory, aby klient mógł wywoływać serwer musi korzystać z protokołu HTTPS.

5.3.4 Uruchomienie serwera oraz klienta gRPC z obsługą SSL TLS

Uruchomienie serwera gRPC jak i klienta z obłsugą szyfrowanego połączenia nie jest skomplikowane. Dla serwera będą potrzebne wcześniej wygenerowane pliki server.crt oraz server.key. Poniższy listing przedstawia poprawne wczytanie kluczy oraz uruchomienie usługi na bezpiecznym połączeniu.

Listing 5.8: Dodanie obsługi połączenia szyfrowanego do serwera gGRPC

```
    func NewGrpcServer(s Service, port int) error {
        list, err := net.Listen("tcp", fmt.Sprintf(":%d", port))
        if err != nil {
                return err
        }
        certFile := "certs/server.crt"
        keyFile := "certs/server.key"
        creds, sslErr :=
        credentials.NewServerTLSFromFile(certFile, keyFile)
        if sslErr != nil {
                log.Fatalf("Failed loading certificates %v", sslErr)
        }
        opts := grpc.Creds(creds)
        gs := grpc.NewServer(opts)
        userpb.RegisterUserServiceServer(gs, &grpcServer{s})
        reflection.Register(gs)
        return gs.Serve(list)
}
```

Po stronie klienta należy wczytać klucz oraz certyfikat, którym jest podpisany. W ten sposób

5.4 Refleksja oraz testowanie gRPC bez udziału klienta

Technologia gRPC posiada jeszcze jedną funkcjonalność, która pozwala na inspekcję działającej usługi wprost z terminala. Funkcjonalność ta została aktywowana w każdej usłudze w aplikacji. Posiadając zdalny lub lokalny dostęp do usługi można za pomocą konsoli można uruchomić interaktywne środowisko(**REPL**) pozwalające przejrzeć oraz przetestować zdefiniowane komunikaty oraz serwisy.

Listing 5.9: Przykad wykorzystania reflekcji

```
pb.UserService@127.0.0.1:7001> show service
+ -----------+------------+-------------------+--------------------+
|   SERVICE  |    RPC     |   REQUEST TYPE    |   RESPONSE TYPE    |
+ -----------+------------+-------------------+--------------------+
| UserService|CreateUser  |CreateUserRequest  |CreateUserResponse  |
| UserService|GetUserByID |GetUserRequest     |GetUserResponse     |
| UserService|GetUsersList|GetUsersListRequest|GetUsersListResponse|
+ -----------+------------+-------------------+--------------------+

pb.UserService@127.0.0.1:7001> call GetUserByID
id (TYPE_STRING) => bm3nv7cb462ihp3vkgu0
{
    "user": {
    "id": "bm3nv7cb462ihp3vkgu0          ",
    "userName": "Paul",
    "createdAt": "AQAAAA7VGXadI/4oOAB4"
    }
}
```

Jest to bardzo pomocne narzędzie, gdy trzeba przetestować wdrożone usługi bez implementacji po stronie klienta. Po stronie technologii REST podobne funkcjonalności

posiada Swagger.

5.5 Wydajność

Z systemów opartych na architekturze mikroserwisowej bardzo często korzysta olbrzymia ilość użytkowników co powoduje znaczny wzrost danych przesyłanych między usługami. Dlatego istotne jest, aby paczki danych były jak najmniejsze. Problem stanowią także formaty, które muszą podlegać ciągłej serializacji oraz deserializacji. Pociąga to za sobą wyższe użycie zasobów sprzętowych do przeprowadzenia tej operacji. W podrozdziale tym znajdują się wyniki testów wydajności niektórych metod, które obrazują stosunek wydajności do zastosowanej metody komunikacji. W tabeli 5.2 umieszczono wyniki czasów dla operacji odczytu oraz zapisu danych w bazie dla usługi użytkownik. Przeprowadzono próby dla 100, 1000 oraz 10000 iteracji. Nietrudno

Tablica 5.2: Test wydajności dla wybranych metod

Operacja	**Iteracje 100**	**Iteracje 1000**	**Iteracje 10000**
GetUserByIdHTTP	00:00:00:80	00:00:03:98	00:00:39:08
CreateUserHTTP	00:00:00:66	00:00:04:56	00:00:42:91
GetUserByIdGrpc	00:00:00:17	00:00:00:71	00:00:04:93
CreateUserGrpc	00:00:00:21	00:00:01:07	00:00:08:66

zauważyć, że rozwiązanie oparte na technologii RPC zdecydowanie wygrywa z podejściem REST. Przewaga ta rośnie wraz ze wzrostem iteracji. Ograniczenie stanowi w tym przypadku baza danych jednak tam gdzie się używa tej technologii serwery jak i bazy działają w klastrach i skalują się znacznie lepiej w stosunku do wymagań.

6 Podsumowanie

6.1 Ocena efektów pracy

Budowanie od podstaw aplikacji rozproszonej opartej na mikroserwisach nie jest proste. Zaplanowanie podziału funkcjonalności na poszczególne usługi będzie sprawiało architektom wiele problemów. Szalenie ważne jest określenie warunków przekazywania danych między usługami, gdyż ma to wpływ na dalszy rozwój systemu. Szkielet gRPC potrafi lepiej kontrolować spójność struktur danych jakie są wymieniane pomiędzy usługami. Wolność jaką daje REST może doprowadzać do niespójności w implementacji rozwiązań.

6.2 Planowany rozwój aplikacji

Aplikacja zbudowana na potrzeby niniejszej pracy realizuje wszystkie postawione przed nią założenia. Jednak zważywszy na charakter architektury mikroserwisowej niewielkim kosztem można dokładać do niej kolejne moduły. Podczas planowania budowy sugerowano się całkowitą autonomicznością każdej z usług więc nic nie stoi aby działały osobno w systemach nie związanych z domeną w której miały funkcjonować pierwotnie. Do obecnego projektu można dołożyć takie usługi jak:

- Usługę wysyłania wiadomości email wysyłająca potwierdzenia złożenia zamówienia,
- System autoryzacji użytkowników,
- Moduł rekomendacji oparty na historii zamówień

Jak widać możliwości dokładania kolejnych funkcjonalności są praktycznie nieograniczone. Dodatkowo przez otwarte API każdego z mikroserwisów można rozszerzać funkcje poprzez integrację z systemami zewnętrznych podmiotów takich jak Goole Books.

6.3 Wnioski

Wykorzystując technologie REST i RPC do budowy mikroserwisów w tym samym czasie powoduje, że lepiej można dojrzeć ich wady i zalety. Proponowany przez firmę Google szkielet budowy aplikacji pomimo młodego wieku(10 lat) jest już na tyle rozwinięty, że można go z powodzeniem wykorzystywać w środowisku produkcyjnym. Mimo tego protokół HTTP 1.1 oraz technologia REST nie muszą się obawiać rychłego upadku, gdyż znacznie lepiej są dostosowane do realiów przesyłania treści konsumowanej przez przeglądarki internetowe. Historia już pokazała, że kolejne rozwiązania oparte na RPC zostawały wyparte przez kolejne, a protokół HTTP trwał dalej. Google krok po kroku czy to przez gRPC czy też przez faworyzowanie stron z wdrożonym protokołem SSL przybliża nas do zastąpienia protokołu HTTP 1.1 nowszą wersją. Na chwilę obecną technologia gRPC znalazła swoją niższe i być może w przyszłości będzie wykorzystywana w przeglądarkach internetowych.

6.4 Realizacja efektów kształcenia

1. *Przedstawiać w jasny sposób, zagadnienia teoretyczne niezbędne do zdefiniowania i rozwiązania wybranego problemu informatycznego.* W rozdziale *Teoretyczne podstawy integracji mikroserwisów* opisano szczegółowo technologie oraz koncepcje wykorzystywane podczas budowy architektury opartej na usługach rozproszonych,

2. *Definiować problem informatyczny i jego składowe, dostrzegając ich wzajemne powiązania.* Podrozdział *Cel i założenia pracy* definiuje problemy z którymi trzeba się zmierzyć podczas budowy mikroserwisów,

3. *Wykorzystywać do formułowania i rozwiązywania zadania inżynierskiego wiedzę i umiejętności nabyte w trakcie studiów.* Do rozwiązania problemów napotkanych w kolejnych krokach projektowania, budowy oraz wdrażania zostały wykorzystane umiejętności oraz wiedza zdobyte w ramach następujących kursów:

   - Projektowanie sieci komputerowych,
   - Systemy operacyjne,
   - Zaawansowane systemy baz danych

4. *Projektować i przeprowadzać eksperymenty informatyczne obejmujące zagadnienia niezbędne do rozwiązania nieskomplikowanego problemu informatycznego* Projekt oraz założenia dotyczące budowy systemu zostały Przedstawione w rozdziale *Projekt aplikacji mikroserwisowej*

5. *Formułować prawidłowe wnioski i sądy dotyczące rozwiązywanego problemu informatycznego.* Wnioski dotyczące rozwiązywanych problemów oraz analiza wdrożonych rozwiązań zostały opisane w rozdziale *Podsumowanie i wnioski*

Listingi

Spis tablic

Bibliografia

[1] Mike Belshe, Roberto Peon i Martin Thomson. *Hypertext transfer protocol version 2 (http/2)*. Spraw. tech. 2015.

[2] Tim Berners-Lee, Roy Fielding i Larry Masinter. „Rfc 3986, uniform resource identifier (uri): Generic syntax, 2005". W: *URL: http://www. faqs. org/rfcs/rfc3986. html* (2014).

[3] Fintan Bolton. *Pure Corba*. Indianapolis, IN: Sams, 2002. ISBN: 978-0672318122.

[4] John Brett. *Getting started with hapi.js : build well-structured, testable applications and APIs using hapi.js*. Birmingham, UK: Packt Publishing, 2016. ISBN: 9781785888182.

[5] David Gourley. *HTTP : the definitive guide*. Beijing Sebastopol, CA: O'Reilly, 2002, s. 7. ISBN: 978-1-4493-7958-2.

[6] Miguel Grinberg. *Flask Web Development, 2nd Edition*. City: O'Reilly Media, Inc, 2018. ISBN: 9781491991725.

[7] Daniel Jacobson. *Interfejs API : strategia programisty*. Gliwice: Helion, 2015. ISBN: 978-83-283-0555-7.

[8] Simon Laurent. *Programming web services with XML-RPC*. Beijing Cambridge Mass: O'Reilly, 2001. ISBN: 9781491946473.

[9] STEPHEN LUDIN. *LEARNING HTTP/2*. Place of publication not identified: O'REILLY MEDIA, INC, USA, 2017. ISBN: 978-1-4919-6260-2.

[10] Bruce Jay Nelson. „Remote procedure call". W: (1981).

[11] Krzysztof Rakowski. *Learning Apache Thrift : make applications communicate using Apache Thrift*. Birmingham: Packt Publishing, 2015. ISBN: 978-1-78588-274-6.

[12] Fanie Reynders. *Modern API design with ASP.NET Core 2 : building cross-platform back-end systems*. New York: Apress, 2018. ISBN: 978-1-4842-3519-5.

[13] Mark Slee, Aditya Agarwal i Marc Kwiatkowski. „Thrift: Scalable cross-language services implementation". W: *Facebook White Paper* 5.8 (2007).

[14] James Snell. *Programming Web services with SOAP*. Sebastopol, CA: O'Reilly & Associates, 2002. ISBN: 0-596-00095-2.

[15] Fielding Roy Thomas i in. „Architectural styles and the design of network-based software architectures". W: *Irvine: University of California* (2000).