



Spis treści

1	Wstęp	2
1.1	Wprowadzenie	2
1.2	Cel i założenia pracy	3
2	Przegląd istniejących rozwiązań	4
2.1	Wstęp	4
2.2	Rozwiązania oparte na technologii REST	4
2.2.1	Spark Java	4
2.2.2	Flask	5
2.2.3	HapiJS	5
2.2.4	ASP NET Core Web API	6
2.3	Rozwiązania oparte na technologii RPC	7
2.3.1	CORBA	7
2.3.2	Protokół XML-RPC	10

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 **mikroserwisową**. Sama koncepcja 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 niewielkich autonomicznych komponentów z których każdy odpowiada za konkretne zadanie. Elementy te współpracując ściśle ze sobą pozwalają na dostarczenie wymaganej logiki biznesowej.

Wielu architektów oraz zespołów programistów wizja rozbicia swojej monolitycznej aplikacji na architekturę mikroserwisową 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 kolejnymi mikroserwisami. Tym samym projektowanie interfejsów API[4] wymagało od architektów oraz programistów rozwiązania problemów w następujących kwestiach:

- wyboru formatu wymiany danych (JSON, XML itp.)

- zaprojektowania ścieżek 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 w oparciu o mikroserwisy wykorzystując zróżnicowany stos technologii. Architektura mikroservisowa 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 niewątpliwie 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 scenariuszy, które symulowałyby typowe problemy z jakimi spotykamy się podczas tworzenia architektury mikroservisowej. Założenia każdego scenariusza obejmują:

- identyfikację oraz opis rozpatrywanego problemu,
- implementację mikroservisów w technologiach RPC oraz REST,
- ocenę efektywności zastosowanych rozwiązań popartą wnioskami lub pomiarami

Scenariusze zostały opracowane tak, aby zaprezentować podstawowe problemy w projektowaniu oraz implementacji mikroservisów i nie wyczerpują każdego możliwego aspektu. Wszystkie scenariusze będą zgromadzone w obrębie jednej solucji jako osobne projekty.

2 Przegląd istniejących rozwiązań

2.1 Wstęp

Zarówno technologia REST jak i RPC nie są czymś nowym w świecie wymiany informacji pomiędzy 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 najpopularniejszych dostępnych w chwili tworzenia pracy.

2.2 Rozwiązania oparte na technologii REST

2.2.1 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 wchodzi 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 2.1: 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");
    }
}
```

2.2.2 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 [3]. Ponadto mamy do dyspozycji cały wachlarz modułów odpowiedzialnych za zarządzanie sesją, logowanie, uwierzytelnianie oraz obsługę baz danych.

Listing 2.2: 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!'
    })
```

2.2.3 HapiJS

Postanie platformy nodejs, która zbudowana jest na podstawie silnika przeglądarki *Chrome* 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[2]

Listing 2.3: 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();

```

2.2.4 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 system Linux. Modułowa budowa pozwala na pobranie lub wykorzystanie tylko tych zależności jakie są wymagane do uruchomienia aplikacji. Są to między innymi biblioteki odpowiedzialne za obsługę baz danych, logowania zdarzeń, uwierzytelniania itp. Każda biblioteka jest dostępna w repozytorium NuGet skąd bardzo łatwo można ją dołączyć do projektu rozszerzając[6] jego funkcjonalności.

Listing 2.4: 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;
}
```

2.3 Rozwiązania oparte na technologii RPC

2.3.1 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[1] 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 referencję 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 deklaracyjny 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 2.5: 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 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 referencję 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 pełni przenośność.
- **Pośrednik Zleceń Obiektowych**-ORB jest zasadniczą częścią technologii CORBA. Jest to zbiór całego 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 pominęła platformę .NET, która jest jedną z głównych stosowanych w oprogramowaniu korporacyjnym.

2.3.2 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 technologii HTTP[5]. XML-RPC jest technologią całkowicie niezależną od zastosowanej architektury oraz języka programowania umożliwiając w pełni działanie w systemach heterogenicznych. Zasada działania protokołu polega na przesyłaniu między klientem, a serwerem w postaci żądań oraz odpowiedzi HTTP tak samo jak robią to nasze przeglądarki internetowe z tą różnicą, że dane są reprezentowane w formacie XML. 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 struktury i tablice. Za pomocą notacji xml można również zapisać bardziej skomplikowane typy takie jak struktury oraz tablice. Definicje struktur oraz tablic przedstawiają się następująco:

Listing 2.6: Reprezentacja tablicy w XML-RPC

```
<value>
  <array>
    <data>
```

Tabela 2.1: Podstawowe typy danych i ich reprezentacja w strukturze XML-RPC

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

```

    <value> XML-RPC value </value>
    ...
    <value> XML-RPC value </value>
  </data>
</array>
</value>

```

Listing 2.7: Reprezentacja struktury w XML-RPC

```

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

```

Listing 2.8: Przykład żądania POST

```

POST /rpchandler HTTP/1.0
User-Agent: AcmeXMLRPC/1.0
Host: xmlrpc.example.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>
```

Listing 2.9: 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>
```

Listingi

2.1	Przykład metody zwracającej tekst w Spark Java	4
2.2	Prosty model aplikacji z użyciem Flask	5
2.3	Przykład uruchomienia serwera http w HapiJS	5
2.4	Przykład definicji adresów URI dla metod http w kontrolerze	7
2.5	Przykład zapisu definicji interfejsu w języku IDL	8
2.6	Reprezentacja tablicy w XML-RPC	10
2.7	Reprezentacja struktury w XML-RPC	11
2.8	Przykład żądania POST	11
2.9	Przykład odpowiedzi serwera	12

Spis tabel

2.1	Podstawowe typy danych i ich reprezentacja w strukturze XML-RPC . . .	11
-----	---	----