Politechnika Łódzka
nstytut Informatyki

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 **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 technologi
- 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** 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

Mimo tych przeszkód korporacje pokroju Amazon, Netflix czy Google budują swoje olbrzymie systemy w oparciu o mikroserwisy wykorzystując zróżnicowany stos technologi. 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 technologi 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 mikroserwisowej. Założenia każdego scenariusza obejmują:

- identyfikację oraz opis rozpatrywanego problemu,
- implementację mikroserwisów 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 mikroserwisów 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 nowy 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 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 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. 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 [2]. Ponadto mamy do dyspozycji cały wachlarz modułów odpowiedzialnych za zarządzenie 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 [1]

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 przygotowanym do jego uruchomienia przygotowany do współpracy z systemami z rodziny Windows jak i dostępnymi na system Linux. Modułowa budowa pozwala na pobranie tylko tych zależności jakie są wymaga-

ne 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.

Listingi