PROJEKT Z PRZEDMIOTU “SYSTEMY ROZPROSZONE”

System do zbierania i manipulowania danych z systemów zewnętrznych w czasie rzeczywistym

Bartosz Kordek

Marcin Włodarczyk

Grzegorz Zacharski

Informatyka, semestr VII, studia niestacjonarne

O projekcie **3**

Koncepcja systemu **3**

Cel projektu **3**

Lista wymagań **3**

Główni aktorzy w systemie **4**

Propozycja architektury **4**

Propozycja technologii **5**

Analiza wymagań **6**

Analiza zadania **6**

Wymagania funkcjonalne i niefunkcjonalne - specyfikacja **6**

Wymagania funkcjonalne 6

Wymagania niefunkcjonalne 7

Przypadki użycia **8**

Architektura systemu **9**

Diagramy klas, komponentów, sekwencji **10**

Planowane testy **12**

Opis przeprowadzonych testów **13**

Instrukcja użytkownika **13**

Uruchamianie aplikacji 13

Zatrzymywanie działania aplikacji 13

Korzystanie z aplikacji 14

Opis zawierający wyszczególnienie wykorzystanych technologii **17**

Repozytorium **17**

Podsumowanie **17**

1. O projekcie

Projekt jest realizowany w ramach przedmiotu Systemy rozproszone na kierunku Informatyka na studiach niestacjonarnych w roku akademickim 2020/2021 na Wydziale Informatyki, Elektroniki i Telekomunikacji Akademii Górniczo-Hutniczej im. St. Staszica w Krakowie.

2. Koncepcja systemu

2.1. Cel projektu

Celem projektu jest stworzenie systemu, którego zadaniem będzie zbieranie danych wysyłanych przez systemy zewnętrzne w czasie rzeczywistym, a następnie poddanie ich modyfikacji. Projekt zostanie zaimplementowany w architekturze mikroserwisów.

System będzie odpowiadać za zbieranie danych z różnych serwisów i przesyłanie zbiorczych wyników do klienta. Umożliwi to wspomaganie i skrócenie czasu wyszukiwania pożądanych informacji przez klienta.

Do celów projektu wybraliśmy jedynie serwisy udostępniające darmowe API. W projekcie nie przewiduje się wykonania warstwy front-endowej, zostanie natomiast wykonane REST API w celu umożliwienia podpięcia zewnętrznego front-endu.

2.2. Lista wymagań

- System rozproszony w oparciu np. o architekturę mikroserwisową
- Udostępnianie REST API przez aplikację
- Skalowalność
- Równoważenie obciążenia
- Umożliwienie współbieżnego korzystania z systemu
- Możliwość zarejestrowania się użytkownika
- Możliwość zalogowania się użytkownika
- Możliwość pobierania danych na temat zakażeń COVID-19

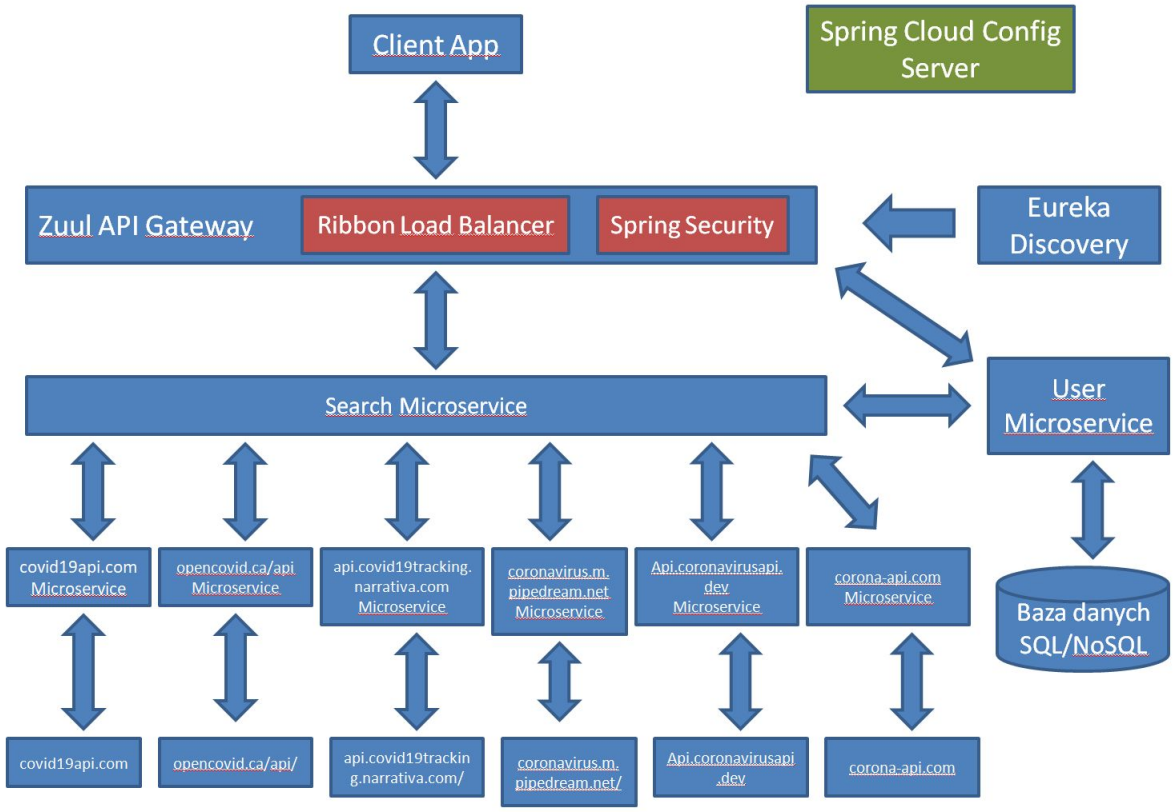
2.3. Główni aktorzy w systemie

W projektowanym systemie można wyróżnić dwa rodzaje aktorów: użytkownika wykorzystującego program klienta API oraz serwisy zewnętrzne udostępniające API. Użytkownik końcowy może jedynie zadawać pytania do systemu, lecz nie może wysyłać do niego żadnych danych. Po otrzymaniu zapytania od klienta, system kieruje swoje zapytanie do serwisów zewnętrznych. Następnie po otrzymaniu odpowiedzi, poddaje odpowiedniej modyfikacji otrzymane dane i następnie wysyła odpowiedź do klienta końcowego.

2.4. Propozycja architektury

Projekt zostanie zrealizowany w architekturze mikroserwisów. Klient komunikuje się poprzez REST API z bramą główną, którą stanowi serwis Zuul API Gateway. Zadaniem tego serwisu jest odbieranie każdego zapytania od użytkownika zewnętrznego, a następnie przekierowanie go do odpowiedniego mikroserwisu. Informację o wszystkich dostępnych działających serwisach w systemie zapewnia Eureka Discovery.

Komunikaty między mikroserwisami oraz API systemów zewnętrznych będą wysyłane przy pomocy protokołu HTTP. Mikroserwisy odpowiadające za bezpośredni kontakt z serwisami zewnętrznymi przesyłają zapytanie HTTP do serwerów zewnętrznych. Odpowiedź zostaje skierowana do serwisu głównego, który zbiera odpowiedzi od pozostałych serwisów i zbiorczo przesyła je do klienta. W przypadku nieotrzymania odpowiedzi zostaje wysłane zapytanie z serwisu proxującego do serwisu głównego w celu umieszczenia informacji o niepowodzeniu w logach.



Rys. 1 Propozycja architektury systemu

2.5. Propozycja technologii

- backend: Java, Spring
- baza danych: MySQL
- konteneryzacja: Docker
- testowanie manualne: Postman
- kontrola wersji: Git
- ciągła integracja (CI): GitHub Actions
- Inne technologie:
    - gateway service: Zuul
    - kontrola mikroserwisów: Eureka
    - równoważenie obciążenia: Ribbon
    - orkiestracja kontenerów: Kubernetes
    - procesowanie logów, audyt: ELK
    - stream and batch data processing: Apache Spark
    - circuit breaker: Hystrix
    - opcjonalnie chaos testing: Chaos Monkey

3. Analiza wymagań

3.1. Analiza zadania

Celem projektu jest stworzenie systemu, którego celem będzie zbieranie danych wysyłanych przez systemy zewnętrzne, a następnie manipulowaniu zebranymi danymi. Jako systemy zewnętrzne wybraliśmy serwisy udostępniające dane na temat zachorowań na COVID:

- covid19tracking.narrativa.com
- covid19api.com
- localcoviddata.com

System będzie odpowiadać za zbieranie danych z różnych serwisów i przesyłanie zbiorczych wyników do klienta. Umożliwi to wspomaganie i skrócenie czasu wyszukiwania pożądanych danych przez klienta.

Do celów projektu wybraliśmy jedynie serwisy udostępniające darmowe API.

W projekcie nie przewiduje się wykonania warstwy front-endowej, zostanie natomiast wykonane REST API w celu umożliwienia podpięcia zewnętrznego front-endu w razie zaistnienia takiej potrzeby.

3.2. Wymagania funkcjonalne i niefunkcjonalne - specyfikacja

3.2.1. Wymagania funkcjonalne

1. Zarejestruj się do systemu
2. Zaloguj się do systemu
3. Pobierz dane na temat ilości nowych zakażeń COVID-19 w danym dniu i w danym kraju
4. Pobierz dane na temat ilości zgonów na COVID-19 w danym dniu i w danym kraju
5. Pobierz dane na temat całkowitej ilości zakażeń w danym kraju

3.2.2. Wymagania niefunkcjonalne

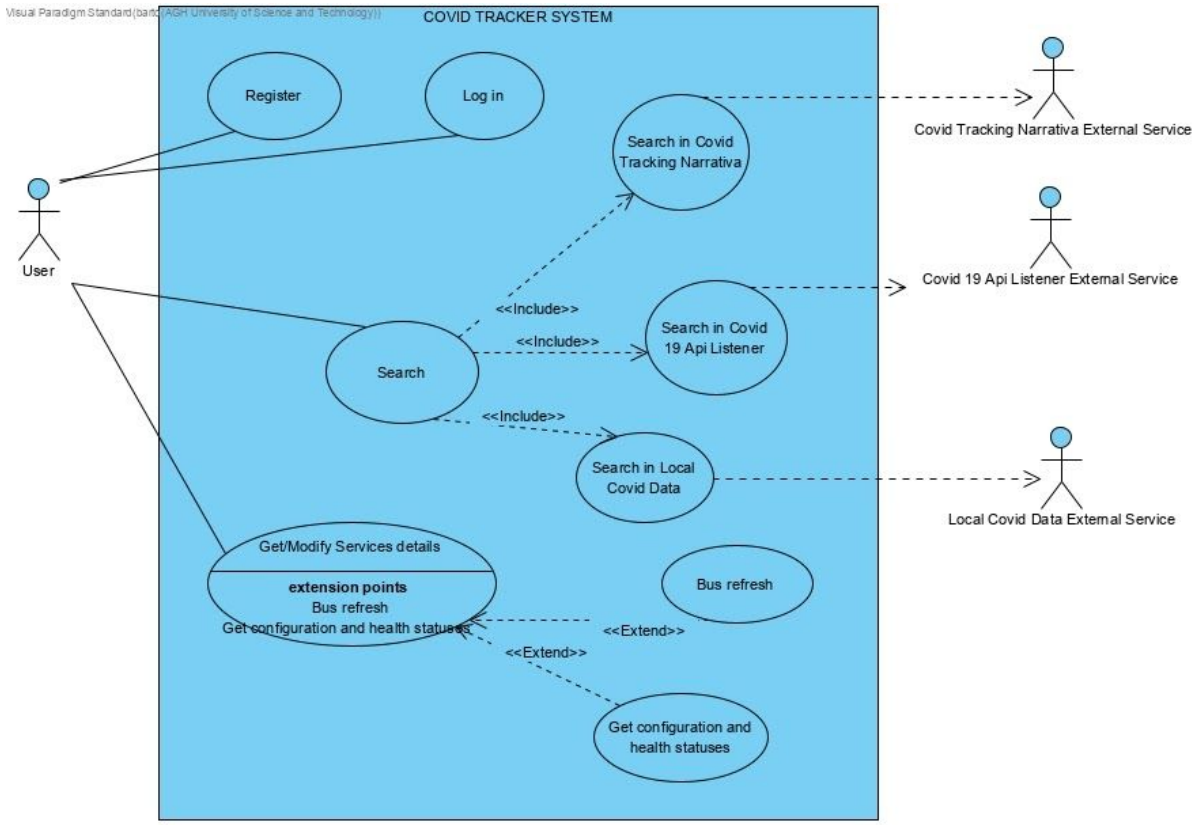
Wymagania produktowe

- Wydajność:
  Liczba transakcji obsłużonych 5/s.
  Rozmiar: Wymagana pamięć RAM minimalnie 8 GB, zalecana 16 GB;
  Łatwość - Czas dla przeszkolenia, użytkowania: Liczba str. dokumentacji, Objętość "Manual" - maksymalnie 10 str.
- Niezawodność:
  Prawdopodobieństwo błędnego wykonania podczas realizacji transakcji: 1/100
  Częstotliwość błędnych wykonań: śr. 1 raz na tydzień
  Średni czas między błędnymi wykonaniami: 1 tydzień
  Dostępność (procent czasu): 99%
- Odporność:
  Czas restartu po awarii systemu: 5 minut

Wymagania implementacyjne

- Język programowania: Java 11
- Nazwy zmiennych w języku angielskim
- Spełnione zasady SOLID
- Wraz z kodem źródłowym POWINNA być dostarczona instrukcja przygotowania środowiska deweloperskiego (od zera)
- Kod POWINIEN być dostarczony wraz z testami automatycznymi dokumentującymi poprawność względem wymagań funkcjonalnych projektu
- Kompilacja całości oprogramowania POWINNA być możliwa przy pomocy jednego polecania
- Kod rozwiązania POWINIEN być przekazany wraz ze wszystkimi bibliotekami i konfiguracjami koniecznymi do jego skompilowania i uruchomienia
- Kod POWINIEN być zrealizowany przy wykorzystaniu następujących konwencji: długość linii powinna wynosić 120 znaków, Wcięcie w kodzie źródłowym
- wykonywane jest przy pomocy spacji, Wcięcie w kodzie źródłowym ma 4 spacje
- Kod rozwiązania POWINIEN posiadać komentarze do wybranych fragmentów i metod kodu
- Rozwiązanie POWINNO być przekazane w postaci umożliwiającej efektywne wgranie kodu do repozytorium GIT

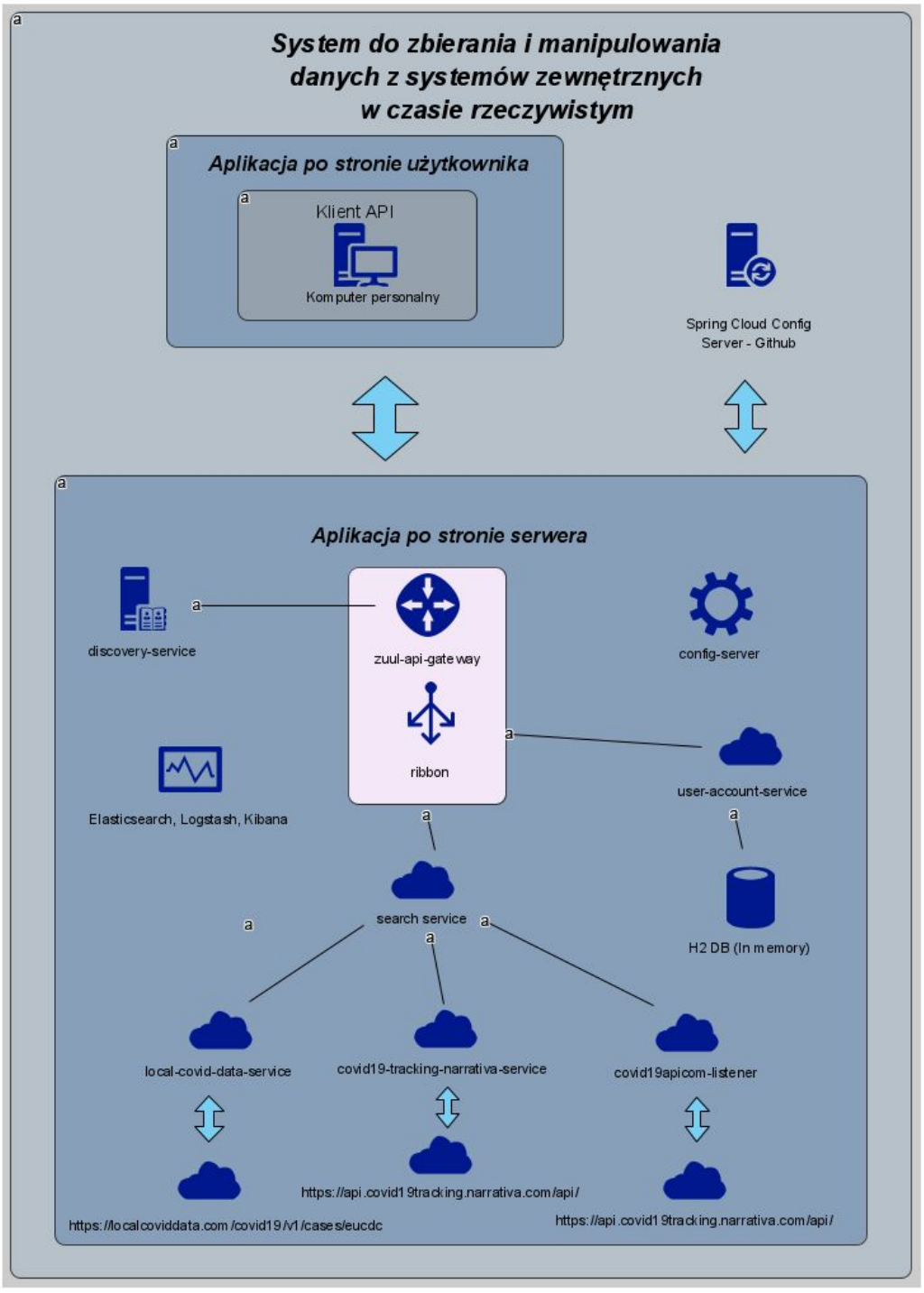
3.3. Przypadki użycia



Rys. 2 Diagram przypadków użycia

Głównym celem aplikacji jest możliwość wyszukiwania statystyk odnośnie COVID19. System obejmuje również podstawowe funkcjonalności takie jak możliwość rejestracji, logowania oraz sprawdzenie poprawności połączenia z mikroserwisami jak również możliwość zmiany konfiguracji korzystając z zewnętrznego pliku i weryfikacji jej działania.

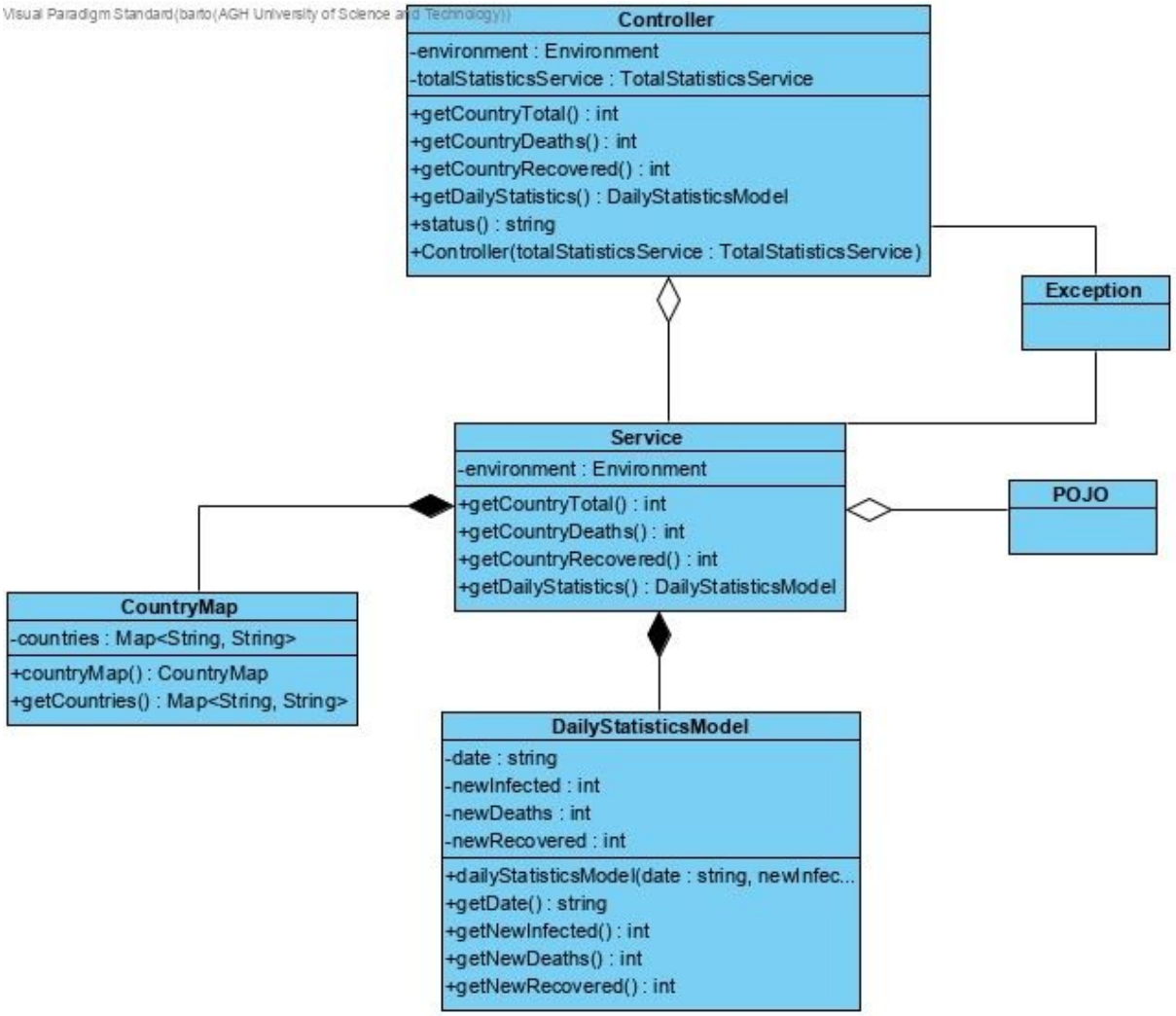
3.4. Architektura systemu



Rys. 3 Architektura systemu

Na powyższym rysunku została przedstawiona architektura systemu do zbierania i manipulowania danymi w czasi rzeczywistym. W porównaniu do założeń koncepcyjnych zmieniła się jedynie liczba serwisów zewnętrznych z którymi komunikuje się zrealizowany system.

3.5. Diagramy klas, komponentów, sekwencji

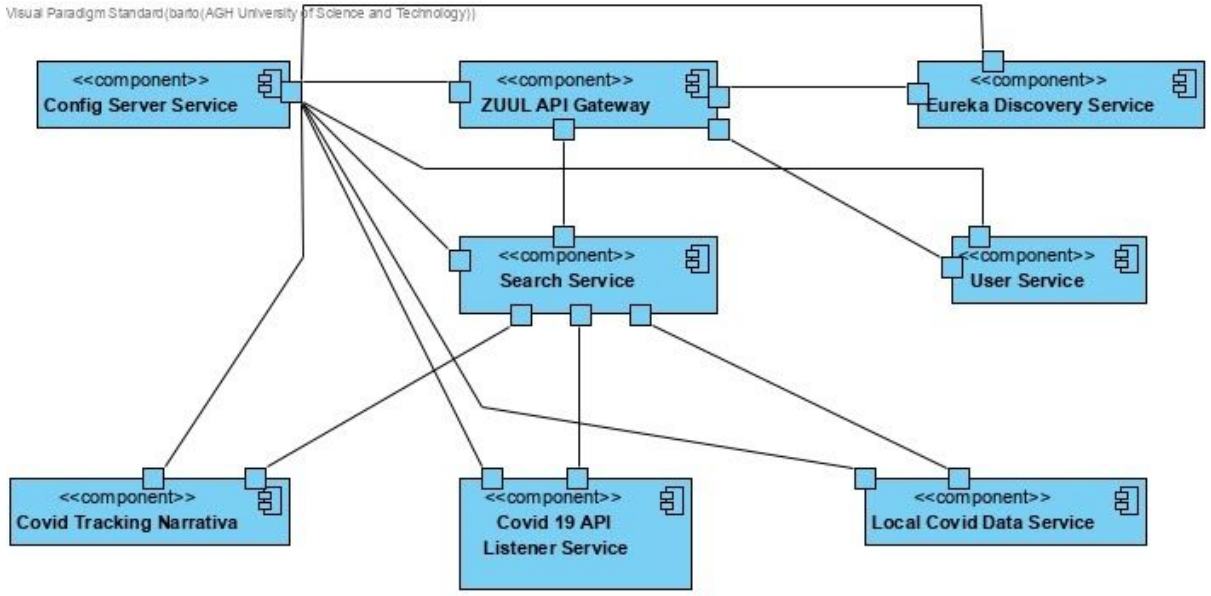


Rys. 4 Diagram klas dla generycznego mikroserwisu serwisu odpowiadającego za kontakt z systemem zewnętrznym

Na rysunku nr 4 został zaprezentowany diagram klas dla serwisu covid19-tracking-narrativa-service. Analogiczną strukturę klas posiada serwis covid19apicom-listener, local-covid-data-service oraz search-microservice. Zastosowaliśmy dwuwarstwową architekturę: warstwa kontrolerów (prezentacji) oraz serwisów (logiki biznesowej). Zapytania skierowane na udostępniane API w pierwszej kolejności zostaną obsłużone przez poszczególne kontrolery. Następnie zapytania są przekazywane do warstwy serwisu, w której wykonywana jest logika biznesowa - zapytanie jest skierowane do API zewnętrznego serwisu. Serwis dokonujący zapytania oczekuje na odpowiedź, a następnie przetwarza zwrócone dane. Na tym etapie zostaje dane zostają poddane obróbce

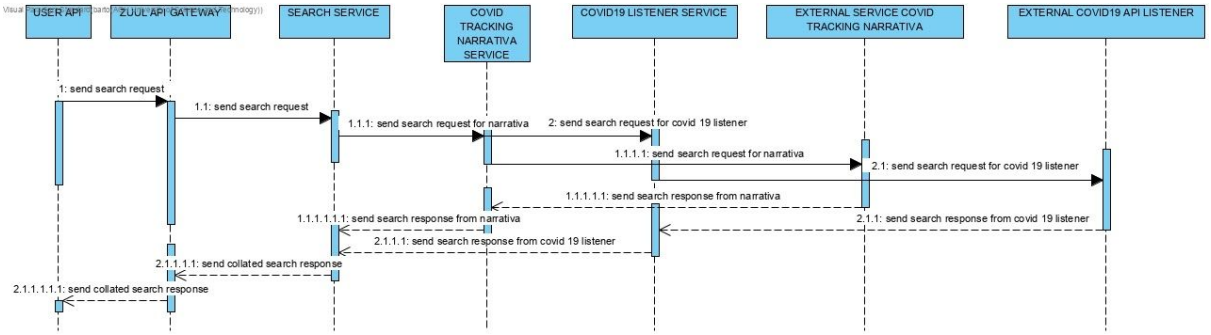
np filtrowaniu. Ostatecznie aplikacja zwraca dane do użytkownika poprzez kontroler lub informuje o wystąpieniu błędu.

Poniżej na rysunku nr 5 znajduje się diagram komponentów. Pierwszy w kontakcie z użytkownikiem systemu jest ZUUL API Gateway Service, który kontaktuje się z Config Server Service, Eureka Discovery Service oraz Search Service. Wspominany Search Service, odpowiada za wysłanie zapytań, a następnie agregację oraz obliczanie średniego wyniku z serwisów odpowiadających za kontakt z zewnętrznymi systemami, tj. Covid Tracking Narrativa Service, Covid 19 API Listener Service, Local Covid Data Service.



Rys. 5 Diagram komponentów dla całego systemu

Na rysunku nr 6 został przedstawiony diagram sekwencji dla operacji wyszukiwania. Klient kontaktuje się poprzez Aplikację użytkownika bezpośrednio do serwisu Zuul. Następnie zapytanie zostanie przesłane do Search serwisu odpowiadającego za przesłanie zapytań do serwisów kontaktujących się bezpośrednio z systemami zewnętrznymi. Zapytanie do każdego z serwisów przesyłane jest w osobnym wątku. Serwis wyszukujący czeka określony czas na odpowiedź każdego z serwisów. Po otrzymaniu odpowiedzi z każdego z nich, bądź przekroczeniu wymaganego czasu oczekiwania na odpowiedź przesłany zostaje wynik do serwisu Zuul, a następnie do użytkownika.



Rys. 6 Diagram sekwencji dla wyszukiwania

3.6. Planowane testy

- Rejestracja nowego użytkownika - podanie prawidłowych danych
- Rejestracja nowego użytkownika - podanie błędnych danych - brak loginu
- Rejestracja nowego użytkownika - podanie błędnych danych - brak hasła
- Rejestracja nowego użytkownika - podanie błędnych danych - brak hasła o wymaganej długości
- Logowanie - podanie prawidłowych danych
- Logowanie - podanie błędnych danych - brak loginu
- Logowanie - podanie błędnych danych - brak hasła
- Logowanie - podanie błędnych danych - błędne hasło
- Logowanie - podanie błędnych danych - brak loginu i hasła
- Wyszukanie dziennych statystyk dla wybranego dnia z dalekiej przeszłości - przed pandemią
- Wyszukanie dziennych statystyk dla wybranego dnia z przeszłości - w trakcie trwania pandemii
- Wyszukanie dziennych statystyk dla wybranego dnia z przyszłości
- Wyszukanie liczby zgonów w danym okresie czasu
- Wyszukanie liczby wyzdrowień w danym okresie czasu
- Wyszukanie liczby całkowitej liczby przypadków zachorowań w danym okresie czasu
- Wyszukanie liczby zgonów od początku pandemii do dnia dzisiejszego
- Wyszukanie liczby wyzdrowień od początku pandemii do dnia dzisiejszego
- Wyszukanie liczby całkowitej liczby przypadków zachorowań od początku pandemii do dnia dzisiejszego

Testy zostaną przeprowadzone dla wybranych krajów: Niemcy (DEU), Francja (FRA), Chiny (CHN), Polska (POL), Hiszpania (ESP), Wielka Brytania (GBR), Stany Zjednoczone (USA).

4. Opis przeprowadzonych testów

Testowanie nowo powstałej aplikacji zostało przeprowadzone za pomocą testów jednostkowych, testów integracyjnych oraz testów bezpieczeństwa na etapie implementacji. Ponadto zostały wykonane testy manualne z wykorzystaniem programu Postman. Wykonano wszystkie zaplanowane testy wymienione w poprzednim podpunkcie dokonując tym samym pozytywnej weryfikacji działania systemu.

5. Instrukcja użytkownika

Do uruchomienia aplikacji wymagany jest zainstalowany Docker.

5.1. Uruchamianie aplikacji

Zbudowanie obrazów Dockerowych dla wszystkich mikroserwisów:
./gradlew dockerBuildImage

Uruchomienie obrazów Dockerowych mikroserwisów w trybie Docker Swarm:
docker swarm init --advertise-addr 172.17.0.1 --listen-addr 0.0.0.0
docker stack deploy --compose-file docker-compose.yaml covid-tracker

Po poprawnym uruchomieniu, aplikacja będzie udostępniać następujące porty do komunikacji:

1. Eureka: http://localhost:8010 - prosty panel administracyjny informujący o działających serwisach
2. API Gateway http://localhost:8011 - brama wejściowa pod którą należy dokonywać zapytania
3. RabbitMQ: http://localhost:15672 (user: guest, pass: guest)
4. Kibana: http://localhost:5601 - wizualizacja logów i monitorowania działania aplikacji
5. Zipkin: http://localhost:9411 - monitorowanie działania aplikacji
6. H2-Console http://localhost:8011/account/h2-console/ (H2-Embedded JDBC URL: jdbc:h2:mem:testdb, user: covid, password: covid) - wbudowana baza danych

6.2. Zatrzymywanie działania aplikacji

Zatrzymanie wszystkich serwisów:

docker stack rm covid-tracker

6.3. Korzystanie z aplikacji

Wszystkie wywołania API poza rejestracją i logowaniem wymagają podania tokena autoryzacyjnego w headerze “Authorization”, które otrzymuje się po zalogowaniu w headerze “token”.

Kraj podawany jako parametr URL należy podawać zgodnie ze standardem iSO 3166-1 alpha-3 (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3) dużymi literami, np. dla Polski będzie to POL. Daty podawane są w formacie yyyy-MM-dd, np. 2020-01-05.

Poniżej znajdują się metody dostępne w API wraz z opisami, przykładowymi ciałami zapytania (dla metod HTTP typu POST) oraz przykładowymi odpowiedziami (dla metod HTTP typu GET).

Metoda	URL	Opis
POST	localhost:8011/account/users	Rejestracja użytkownika w systemie, dane podaje się w formacie JSON w ciele zapytania

Przykładowe ciało zapytania:

```
{
    “firstName”: “Imię”,
    “lastName”: “Nazwisko”,
    “email”: “Imię@Nazwisko.com”,
    “password”: “53cr3tP455w0rd”
}
```

Metoda	URL	Opis
POST	localhost:8011/account/login	Zalogowanie użytkownika, w odpowiedzi oczekujemy statusu 200 OK oraz tokenu autoryzacyjnego w headerze “token”

Przykładowe ciało zapytania:

{

“email”: “Imię@Nazwisko.com”,

“password”: “53cr3tP455w0rd”

}

Metoda	URL	Opis
GET	localhost:8011/search/status/check	Sprawdzenie czy aplikacja działa i odpowiada oraz na jakim porcie działa serwis obsługujący aktualne zapytanie

Przykładowa odpowiedź:

“Working 40625”

Metoda	URL	Opis
GET	localhost:8011/search/{country}?date={date}	Pobranie zbiorczych statystyk dla danego kraju oraz daty

Przykładowa odpowiedź:

{

"date": "2021-01-05",

"newInfected": 7596,

"newDeaths": 341,

"newRecovered": 9338

}

Metoda	URL	Opis
GET	localhost:8011/search/{country}/total?from={dateFrom}&to={dateTo}	Pobranie całkowitej ilości zakażeń w danym kraju oraz w podanym zakresie czasu

Przykładowa odpowiedź:

“24769”

Metoda	URL	Opis
GET	localhost:8011/search/{country}/deaths?from={dateFrom}&to={dateTo}	Pobranie całkowitej ilości zgonów na COVID-19 w danym kraju oraz w podanym zakresie czasu

Przykładowa odpowiedź:

“546”

Metoda	URL	Opis
GET	localhost:8011/search/{country}/recovered?from={dateFrom}&to={dateTo}	Pobranie całkowitej ilości zgonów na COVID-19 w danym kraju oraz w podanym zakresie czasu

Przykładowa odpowiedź:

“32611”

7. Opis zawierający wyszczególnienie wykorzystanych technologii

Ostatecznie użyte technologie różnią się nieznacznie od tych przedstawionych w podpunkcie 2.5. Zdecydowaliśmy, że aplikacja frontendowa nie jest konieczna i nie wniesie dużej wartości do systemu. Ponadto nie wykonywaliśmy testów odporności systemu przy pomocy Chaos Monkey, ponieważ nasz system okazał się zbyt mały dla tego narzędzia. Do ostatecznego zrealizowania naszego projektu wybraliśmy następujące technologie:

- **Front-end:** brak, zapytania wysyłane za pomocą programu klienta API np. Postman
- **Back-end:** Mikroserwisy napisane w języku Java 11, zrealizowana w oparciu o framework Spring Boot. Szczegóły pod tym linkiem.
- **Bazy danych**: H2 in-Memory Database
- **Konteneryzacja:** Docker
- **Orkiestracja:** Docker Swarm
- **Komunikacja:** RabbitMQ
- **Przetwarzanie logów, wyszukiwanie oraz wizualizacja:** ElasticStack: Elasticsearch, Logstash i Kibana
- **Śledzenie i monitorowanie wydajności:** Zipkin

8. Repozytorium

Ostateczna wersja kodu źródłowego systemu jest dostępna w repozytorium *https://github.com/bartoszkordek/CovidTracker*.

9. Podsumowanie

Analizując wykonanie powyższego projektu możemy stwierdzić, że architektura mikroserwisowa jest znacznie cięższa w implementacji w porównaniu do architektury monolitycznej. Jednakże, umożliwia ona bardzo łatwe skalowanie systemu co przekłada się na efektywne dostosowanie się do potrzeb użytkowników. Zapewnia też łatwą możliwość rozbudowy systemu o kolejne serwisy.

Tego typu architektury nie zastosowalibyśmy do małych aplikacji oraz do aplikacji o jasnych wymaganiach, które nie zmienią się znacząco w przyszłości. Musimy się ponadto liczyć, że szybkość działania systemu wykonanego w architekturze mikroserwisowej będzie wolniejsza od szybkości działania systemu monolitycznego ze względu na komunikację międzyserwisową.