Load Balancing - Netflix Zuul

Introduction

Microservices have revolutionized application development by allowing us to build smaller, specialized components that seamlessly interact. While they offer numerous advantages, they also introduce challenges such as routing, load balancing, and maintaining a consistent interface. Netflix Zuul is a robust solution that helps tackle these issues, facilitating the creation of scalable, resilient, and efficient microservices architecture.

As a widely used API gateway and load balancer, Netflix Zuul streamlines routing and load management in microservices-based systems. Serving as a central entry point for client requests, it offers essential features such as authentication, request routing, rate limiting, and more. By distributing incoming requests evenly across multiple service instances, Zuul enhances both performance and reliability.

The prerequisites for this tutorial include Java JDK (version 8 or higher), a build tool such as Maven or Gradle (with Gradle being used in this tutorial), and IDE, like IntelliJ IDEA or Eclipse (IntelliJ IDEA in this tutorial).

Implementation

Step 1: Create the Eureka Server project

- Open IntelliJ IDEA and select “New Project”
- Choose Gradle as the project type and select Java (make sure the Java SDK ≥ 8 is selected)
- Enter the `groupId` and `artifactId` (e.g., `com.soa` and `eurekaservice`)
- Click “Finish”
- After the project is created, edit `build.gradle` to add dependencies for Spring Boot and Eureka server:

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

- Create a main Java class for your Spring Boot application (e.g., `EurekaServiceApplication.java`):

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServiceApplication.class, args);
    }
}
```

- Configure `application.properties` in `src/main/resources/`:

```
spring.application.name=eureka-service
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=true
```

- Build the project
- Create a Docker image from the project, using the following `Dockerfile`:

```
FROM openjdk:17-jdk-alpine
RUN addgroup -S springdocker && adduser -S springdocker -G springdocker
USER springdocker:springdocker
ARG JAR_FILE=build/libs/eureka-service-0.0.1-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar
```

```
ENTRYPOINT ["java","-jar","/app.jar"]
EXPOSE 8761
```

Step 2: Create the Zuul Gateway project

- Repeat the procedure from Step 1 to create a new Gradle project for Zuul, changing the `artifactId` to, for example, `zuulgatewayservice`
- Modify the `build.gradle` to include Zuul and Eureka client dependencies:

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-zuul'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- Create the main application class (e.g., `ZuulGatewayServiceApplication.java`):

```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
public class ZuulGatewayServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZuulGatewayServiceApplication.class, args);
    }

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**").allowedOrigins("*");
            }
        };
    }
}
```

- The `corsConfigurer` method allows cross-origin requests, enabling frontend applications to interact with microservices without being blocked by the browser's same-origin policy, ensuring smooth communication between the client and the backend services
- Configure `application.properties`:

```
spring.application.name=gateway-service
server.port=8765

zuul.ignored-headers=Access-Control-Allow-Credentials, Access-Control-Allow-Origin
zuul.sensitiveHeaders=Cookie,Set-Cookie

zuul.prefix=/api
#When path starts with /api/user/**, redirect it to user-management service.
zuul.routes.user.path=/user/**
zuul.routes.user.serviceId=user-management
#When path starts with /api/product/**, redirect it to products-management service.
zuul.routes.product.path=/product/**
zuul.routes.product.serviceId=products-management

#eureka
eureka.client.service-url.default-zone=http://eureka-container:8761/eureka/
eureka.instance.lease-renewal-interval-in-seconds=30
```

```
eureka.instance.lease-expiration-duration-in-seconds=90

#load balancing
ribbon.eureka.enabled=true
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
ribbon.ServerListRefreshInterval=5000

zuul.ribbon.eager-load.enabled=true
ribbon.ReadTimeout=60000
ribbon.ConnectTimeout=10000
```

- Build the project
- Create a Docker image using the following `Dockerfile`:

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S springdocker && adduser -S springdocker -G springdocker
USER springdocker:springdocker
ARG JAR_FILE=build/libs/zuul-gateway-service-0.0.1-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
EXPOSE 8765
```

Step 3: Create a microservice

- When creating a new microservice, add the necessary Spring Boot and Eureka client dependencies in `build.gradle`:

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-web'
   implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
}
```

- Configure `application.properties`:

```
spring.application.name=products-management
server.port=8081

#eureka
eureka.client.service-url.default-zone=http://eureka-container:8761/eureka/
eureka.instance.lease-renewal-interval-in-seconds=30
eureka.instance.lease-expiration-duration-in-seconds=90

#load balancing
ribbon.eureka.enabled=true
```

- Optionally, create an image for it too using a `Dockerfile`
- For this tutorial, I created a simple application that stores information about products of an online shop and exposes an endpoint for retrieving all products (`GET /service/all`)

Step 4: Run and test the setup

- Create the `docker-compose.yml` file that will create and run your containers:

```
services:
   eureka-container:
      image: eureka-image
      ports:
         - "8761:8761"
      networks:
```

```
            - app-network
    product-container:
        image: product-image
        ports:
            - "8081:8081"
        depends_on:
            - eureka-container
        networks:
            - app-network
        environment:
            server.port: 8081
            eureka.client.serviceUrl.defaultZone: http://eureka-container:8761/eureka/
    zuul-container:
        image: zuul-image
        ports:
            - "8765:8765"
        depends_on:
            - eureka-container
        networks:
            - app-network
        environment:
            server.port: 8765
            eureka.client.serviceUrl.defaultZone: http://eureka-container:8761/eureka/
networks:
    app-network:
        driver: bridge
```

- For example, create 3 services, one for each project, use the images created in the previous steps, map the internal container ports to the host ports, add a network so that the services can communicate using the container names
- Build this file using `docker-compose build`
- Run using `docker-compose up`
- Access the microservice through the Zuul gateway — open a browser of your choice or use a tool like Postman to send a request to an implemented endpoint in the microservice; you should see the response from the microservice
- For example, I can call `http://localhost:8765/api/product/service/all` from the browser and the response should be a list of available products in the shop

This setup creates a basic microservices architecture with Eureka being used for service discovery and Netflix Zuul for routing. More microservices can be added, and Zuul can be configured for more complex routing and load balancing, as needed.