Prof. Bernardo Copstein

Prof. Júlio Machado

Roteiro 7: Comunicação por Filas

OBSERVAÇÃO: o roteiro que segue faz referência ao comando "docker compose" para execução das tarefas solicitadas. Caso você esteja utilizando o ambiente Linux nos laboratórios do prédio 32, utilize o comando equivalente "podman-compose".

Introdução

A figura 1 mostra a situação da nossa arquitetura até o momento.

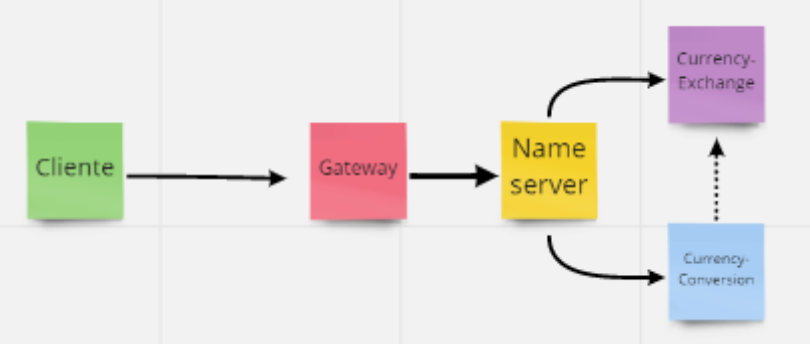

Figura 1 – Sistema desenvolvido até o momento

No roteiro anterior vimos como disponibilizar todos os micros serviços de forma conjunta de maneira a dispor de uma aplicação através do Docker Compose. O propósito deste roteiro é introduzir a comunicação assíncrona entre micros serviços.

Quando fazemos comunicação entre micro serviços podemos optar entre comunicação síncrona e assíncrona. Quando a comunicação é assíncrona temos duas alternativas:

1) A comunicação é assíncrona apenas do ponto de vista de quem demanda, pois quem é demandado deve atender a demanda imediatamente. A opção pelo modo assíncrono neste caso deve-se ao fato do tempo que a demanda levará para ser atendida, evitando que o demandante permaneça bloqueado enquanto aguarda.

2) A comunicação é assíncrona tanto do ponto de vista de quem demanda como do ponto de vista de quem é demandado. Neste caso o demandante dispara a demanda, mas pode não ter interesse em saber quando ela será atendida e, muitas vezes, não necessita de retorno a respeito da demanda. O demandado por sua vez atende a demanda quando tiver disponibilidade.

O suporte para a implementação da comunicação assíncrona pode fazer uso de um "message broker" capaz de implementar filas de mensagens. Desta forma podemos encaminhar as mensagens para uma fila e os micros serviços podem se registrar para atender aquelas que são do seu interesse.

Para podemos trabalhar com as filas de mensagens vamos precisar de um serviço de "message broker" executando (ver figura 2). Neste roteiro iremos usar o RabbitMQ (https://www.rabbitmq.com/) aliado com os recursos da tecnologia Spring-Boot AMQP (https://spring.io/projects/spring-amqp).

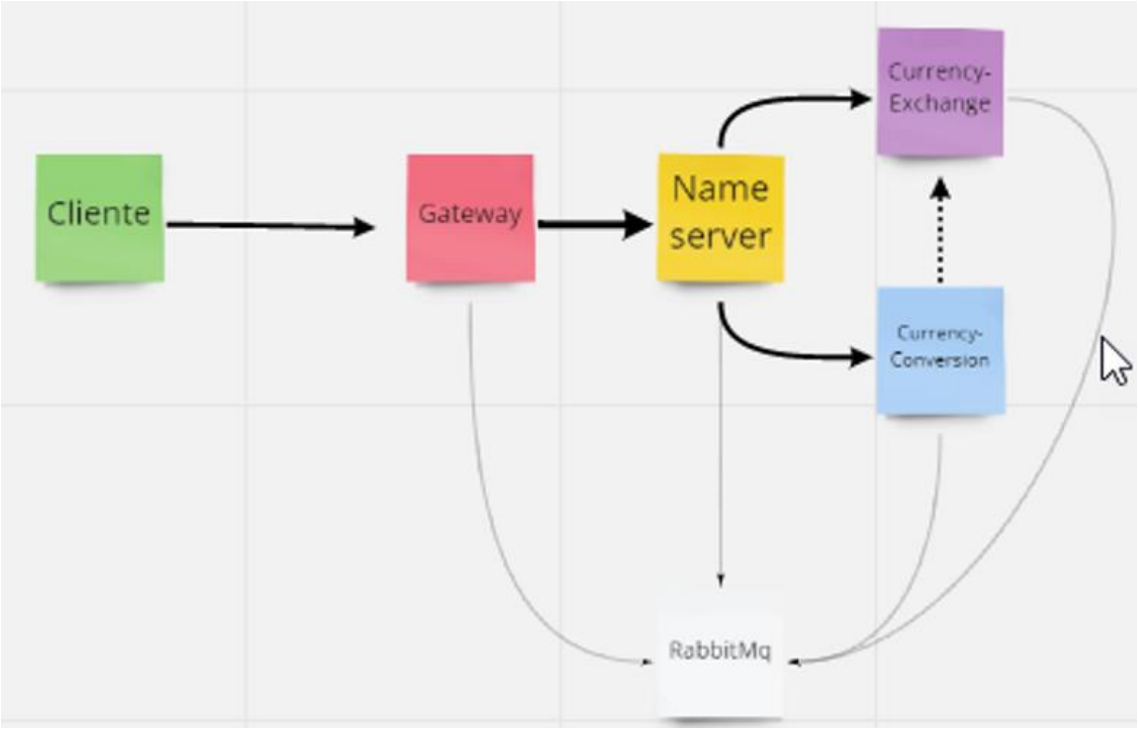

Figura 2 – Sistema com fila de mensagens

RabbitMQ é um servidor de mensageria de código aberto, que faz uso do protocolo AMQP (*Advanced Message Queuing Protocol*). O RabbitMQ é compatível com muitas linguagens de programação e permite lidar com o tráfego de mensagens de forma simples e confiável.

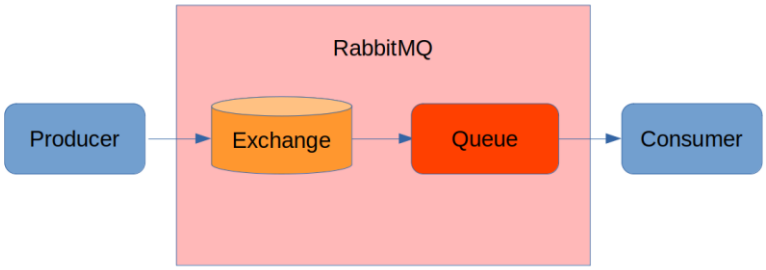

Figura 3 – Componentes do RabbitMQ

- Message: contêm os dados compartilhados entre um produtor e um consumidor.
- Producer/Publisher: é o responsável por produzir/enviar mensagens.
- Exchange: agente responsável pelo roteamento de mensagens para diferentes filas.
- Queue: estrutura de armazenamento de mensagens.
- Consumer: é o responsável por consumir/receber mensagens.

Devido às múltiplas necessidades de roteamento de mensagens, o RabbitMQ possui diferentes tipos de *exchange*, tais como:

- Direct: entrega a mensagem para as filas que estão ligadas a ela baseado no valor de uma *routing keys*; ideal para comunicação *unicast*. Ver figura 4.
- Fanout: esse tipo é mais utilizado como *broadcast*, todos os interessados vão receber as mensagens sem qualquer filtro (ignora *routing keys*). Ver figura 5.
- Topic: permite o envio para várias filas com base no valor de *routing keys* e em padrões configurados na vinculação entre a fila e o *exchange*; ideal para comunicação *multicast*.

Routing key é uma chave enviada junto a mensagem que o *exchange* usa para decidir para onde vai rotear a mensagem.

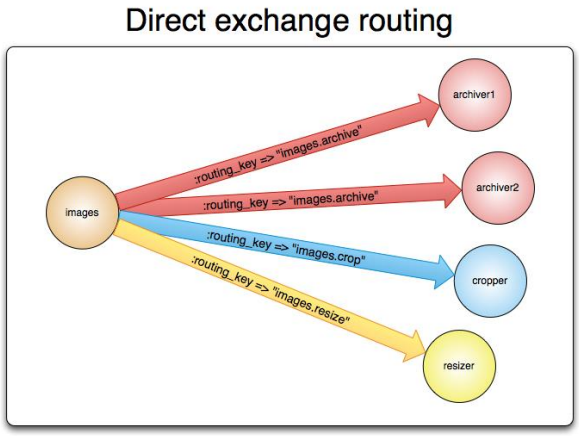

Figura 4 – Direct exchange

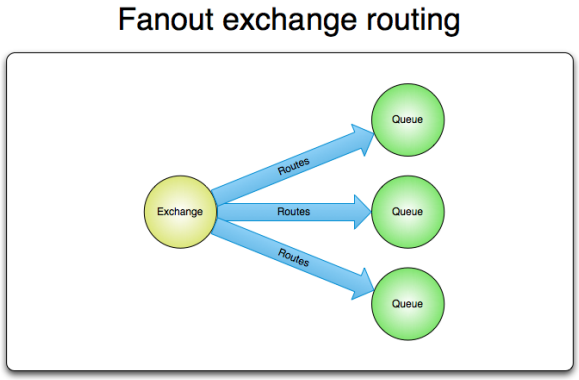

Figura 5 – Fanout exchange

Existem várias opções para usar o RabbitMQ:

- Uso de um servidor instalado localmente;
- Uso de uma imagem docker;
- Uso de um serviço de plataforma da nuvem (tal como https://www.cloudamqp.com/).

Neste roteiro vamos utilizar uma imagem docker.

Passo 1: configurando a imagem docker no "compose.yaml"

O docker compose usa o arquivo "compose.yaml" para descrever a aplicação que queremos executar. Então teremos de descrever a dependência para o RabbitMQ neste arquivo. A figura 6

apresenta a adição que deve ser feita ao arquivo “compose.yaml” resultante do roteiro anterior. Nessa configuração foi alterado o usuário e senha padrão de “guest” para “engsoft2”.

```
services:
  rabbitmq:
    image: rabbitmq:management
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      - RABBITMQ_DEFAULT_USER=engsoft2
      - RABBITMQ_DEFAULT_PASS=engsoft2
    networks:
      - currency-network
```

Figura 6 – Imagem do RabbitMQ no arquivo “compose.yaml”

Os demais serviços que irão fazer uso da comunicação através de filas também devem ter suas configurações alteradas no arquivo “compose.yaml” para indicar a dependência ao RabbitMQ e indicar a sua localização. Para este roteiro, o serviço produtor de mensagens será o “currency-conversion” (ver figura 7). O serviço consumidor será visto mais a frente.

```
services:
  currency-conversion:
    image: conversion
    ports:
      - "8100:8100"
    networks:
      - currency-network
    depends_on:
      - naming-server
      - rabbitmq
    environment:
      - eureka.client.serviceUrl.defaultZone=http://naming-
server:8761/eureka
      - spring.rabbitmq.host=rabbitmq
      - spring.rabbitmq.port=5672
      - spring.rabbitmq.username=engsoft2
      - spring.rabbitmq.password=engsoft2
    deploy:
      resources:
        limits:
          memory: 700m
```

Figura 7 – Configuração de dependência ao RabbitMQ no arquivo “compose.yaml”

Passo 2: configurando o micro serviço produtor

Para este roteiro, o serviço produtor de mensagens será o “currency-conversion-service” e devemos modificar as configurações do Spring no arquivo “pom.xml”, adicionando a dependência conforme a figura 8.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
       <modelVersion>4.0.0</modelVersion>
       <parent>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-parent</artifactId>
              <version>3.0.5</version>
              <relativePath/> <!-- lookup parent from repository -->
       </parent>
       <groupId>com.engsoft2</groupId>
       <artifactId>currency-conversion-service</artifactId>
       <version>0.0.1-SNAPSHOT</version>
       <name>currency-conversion-service</name>
       <description>Currency conversion microservice</description>
       <properties>
    <java.version>17</java.version>
    <spring-cloud.version>2022.0.2</spring-cloud.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>netflix-candidates</id>
      <name>Netflix Candidates</name>
      <url>https://artifactory-oss.prod.netflix.net/artifactory/maven-oss-
candidates</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

</project>
```

Figura 8 – Configuração de dependência ao RabbitMQ no arquivo “pom.xml”

No projeto do micro serviço “currency-conversion” será necessário configurar o acesso ao RabbitMQ da maneira desejada, ou seja, de acordo com o padrão de composição de objetos *exchange*, *queue* e serialização de dados conforme a necessidade. Para tal, adicione um arquivo com nome “RabbitMQConfig.java” conforme a figura 9.

```
import org.springframework.amqp.core.FanoutExchange;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {
    public static final String fanoutExchangeName =
"conversions.v1.conversion-request";
    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange(fanoutExchangeName);
    }
    @Bean
    public MessageConverter jsonMessageConverter() {
        return new Jackson2JsonMessageConverter();
    }
    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory factory,
MessageConverter messageConverter) {
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(factory);
        rabbitTemplate.setMessageConverter(messageConverter);
        return rabbitTemplate;
    }
}
```

Figura 9 – Classe de configuração do RabbitMQ no arquivo “RabbitMQConfig.java”

Nessa classe, foram configurados um *exchange* do tipo *fanout* que realiza um *broadcast* de qualquer mensagem recebida para qualquer fila associada a ele. Observe que não existe nenhuma fila configurada pois o produtor não depende diretamente das filas, apenas do objeto *exchange*. Perceba também que foi configurado um objeto serializador para JSON. A opção escolhida foi o serializador padrão da API “Jackson” utilizada pelo Spring. Por fim, foi customizado

o objeto “RabbitTemplate”, responsável pelo envio de mensagens, com o serializador que suporta mensagens no formato JSON.

O próximo passo é a criação de um objeto DTO com o conteúdo da mensagem. Crie um arquivo “HistoryDTO.java” de acordo com a figura 10. Esse é um objeto POJO contendo apenas propriedades com os dados a serem serializados.

```
public class HistoryDTO {
    private String from;
    private String to;

    public HistoryDTO(String from, String to) {
        this.from = from;
        this.to = to;
    }
    public HistoryDTO() {
    }
    public String getFrom() {
        return from;
    }
    public void setFrom(String from) {
        this.from = from;
    }
    public String getTo() {
        return to;
    }
    public void setTo(String to) {
        this.to = to;
    }
    @Override
    public String toString() {
        return "HistoryDTO [from=" + from + ", to=" + to + "]";
    }
}
```

Figura 10 – Classe DTO para serialização de mensagem no arquivo “HistoryDTO.java”

O último passo é alterar o controlador “CurrencyConversionController.java” para enviar uma mensagem para o RabbitMQ quando receber uma solicitação de conversão de valores monetários. Veja a alteração na figura 11. Observe que é o objeto “RabbitTemplate”, obtido via injeção de dependência, o responsável por serializar e enviar o objeto para o *message broker*.

```
import java.math.BigDecimal;
import java.util.HashMap;
import org.springframework.amqp.core.FanoutExchange;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class CurrencyConversionController {
    @Autowired
    private CurrencyExchangeProxy proxy;

    @Autowired
    private RabbitTemplate template;

    @Autowired
    private FanoutExchange fanout;

    @GetMapping("/currency-
conversion/from/{from}/to/{to}/quantity/{quantity}")
```

```
    public CurrencyConversion calculateCurrencyConversion(@PathVariable
String from, @PathVariable String to, @PathVariable BigDecimal quantity) {
        HashMap<String, String> uriVariables = new HashMap<>();
             uriVariables.put("from",from);
             uriVariables.put("to",to);
        ResponseEntity<CurrencyConversion> responseEntity = new
RestTemplate().getForEntity("http://localhost:8000/currency-
exchange/from/{from}/to/{to}", CurrencyConversion.class, uriVariables);
        CurrencyConversion currencyConversion = responseEntity.getBody();
        return new CurrencyConversion(
            currencyConversion.getId(),
                    from, to, quantity,
                    currencyConversion.getConversionMultiple(),

      quantity.multiply(currencyConversion.getConversionMultiple()),
                    currencyConversion.getEnvironment()+ " " + "rest
template"
        );
    }

    @GetMapping("/currency-conversion-
feign/from/{from}/to/{to}/quantity/{quantity}")
    public CurrencyConversion calculateCurrencyConversionFeign(@PathVariable
String from, @PathVariable String to, @PathVariable BigDecimal quantity) {
        CurrencyConversion currencyConversion =
proxy.retrieveExchangeValue(from, to);
        HistoryDTO dto = new HistoryDTO(currencyConversion.getFrom(),
currencyConversion.getTo());
        template.convertAndSend(fanout.getName(),"",dto);
        return new CurrencyConversion(currencyConversion.getId(),
                    from, to, quantity,
                    currencyConversion.getConversionMultiple(),

      quantity.multiply(currencyConversion.getConversionMultiple()),
                    currencyConversion.getEnvironment() + " " + "feign"
        );
    }
}
```

Figura 10 – Classe controller do microsserviço

Passo 3: configurando o micro serviço consumidor

Para este roteiro, o serviço consumidor de mensagens será o “history-service”. O código-fonte do micro serviço está disponível para download. Atualmente ele apena realiza o logging das mensagens recebidas no console.

De posso do código, observe o arquivo “RabbitMQConfig.java” na figura 11. É neste arquivo que foi realizada a associação entre o *exchange* configurado no produtor com uma *queue* que irá receber as mensagens que o micro serviço irá processar de forma assíncrona.

```
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.FanoutExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
public class RabbitMQConfig {
    public static final String fanoutExchangeName =
"conversions.v1.conversion-request";
    public static final String queueName = "conversions.v1.conversion-
request.save-history";
    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange(fanoutExchangeName);
    }
    @Bean
    public Queue queue() {
        return new Queue(queueName);
    }
    @Bean
    public Binding binding(Queue q, FanoutExchange f) {
        return BindingBuilder.bind(q).to(f);
    }
    @Bean
    public MessageConverter jsonMessageConverter() {
        return new Jackson2JsonMessageConverter();
    }
    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory factory,
MessageConverter messageConverter) {
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(factory);
        rabbitTemplate.setMessageConverter(messageConverter);
        return rabbitTemplate;
    }
}
```

Figura 11 – Classe de configuração do RabbitMQ no arquivo “RabbitMQConfig.java”

Para consumir as mensagens de uma *queue*, o projeto define uma classe que contém um receptor de mensagens representado por um método anotado com “@RabbitListener”. Veja na figura 12 como esse objeto foi definido.

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class Receiver {
    public static final String queueName = "conversions.v1.conversion-
request.save-history";
    private static Logger logger = LogManager.getLogger(Receiver.class);
    @RabbitListener(queues = queueName)
    public void receive(HistoryDTO dto) {
        logger.info("Mensagem recebida: " + dto);
    }
}
```

Figura 11 – Classe receptora de mensagens no arquivo “Receiver.java”

Não esqueça que todo os serviços que irão fazer uso da comunicação através de filas também devem ter suas configurações alteradas no arquivo “compose.yaml” para indicar a dependência ao RabbitMQ e indicar a sua localização. Para este roteiro, o serviço consumidor de mensagens deve ser configurado conforme a figura 12.

```
services:
  history:
    image: history
    ports:
      - "8120:8120"
    networks:
      - currency-network
```

```
    depends_on:
      - naming-server
      - rabbitmq
    environment:
      - eureka.client.serviceUrl.defaultZone=http://naming-
server:8761/eureka
      - spring.rabbitmq.host=rabbitmq
      - spring.rabbitmq.port=5672
      - spring.rabbitmq.username=engsoft2
      - spring.rabbitmq.password=engsoft2
    deploy:
      resources:
        limits:
          memory: 700m
```

Figura 12 – Configuração de dependência ao RabbitMQ no arquivo “compose.yaml”

Passo 4: colocando o sistema para execução

Lembre-se dos passos para compilar, gerar a imagem e executar os contêineres:

Para criar um arquivo “.jar” compile o programa usando o comando:

```
mvnw clean package
```

Crie a imagem propriamente dita usando o comando que segue:

```
docker build -t “nome da imagem”:latest .
```

Para executar o arquivo “compose.yaml” execute o seguinte comando a partir de uma janela de “shell”:

```
docker compose up
```

A fim de testar se o RabbitMQ está executando, acesse a ferramenta de gerenciamento via navegador pela URL: http://localhost:15672.

Teste se tudo está funcionando de acordo enviado uma requisição “HTTP” através do “Postman” ou outro software equivalente. Se tudo ocorreu bem, observe a saída de console da aplicação e procure pelas mensagens do logger associado ao microsserviço “history-service”.

Teste com a URL que segue:

http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10

Para encerrar a execução das imagens use “ctrl-c”.

Para finalizar, digite o comando:

```
docker compose down
```

Exercícios

1) O projeto do micro serviço “history-service” já foi configurado com as dependências para a utilização do banco de dados em memória H2. Faça uso dele para armazenar o histórico de conversões de valores monetários. Acrescente também um “endpoint” que permita consultar o histórico armazenado no banco.