Arquitetura de Software
Prof. Bernardo Copstein
Micros Serviços

Roteiro 8: Comunicação por filas

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 não tem 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.

Para implementar a alternativa 2 precisamos de um "message broker" capaz de implementar filas de mensagens. Desta forma podemos encaminhar as mensagens para uma fila e os serviços podem se registrar para atender aquelas que são do seu interesse. O uso de filas de mensagens é o mais adequado para implementarmos "coreografia" de serviços ao invés de "orquestração" como temos feito até o momento.

Para podemos trabalhar com as filas de mensagens vamos precisar de um serviço de "message broker" executando. Neste roteiro iremos usar o 'RabbitMQ' (www.rabbitmq.com) aliado com os recursos da tecnologia Spring-Boot.

Solicite o código deste exemplo para o professor. O texto que segue apenas destaca os principais aspectos relacionados as filas.

No docker compose é necessário indicar a dependência entre os módulos que usam filas e o serviço de filas. Caso contrário corre-se o risco da conexão não existir por problemas de sincronismo na inicialização.

```
depends_on:
  - naming-server
  - rabbitmq
environment:
  EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://naming-server:8761/eureka
  SPRING.ZIPKIN.BASEURL: http://zipkin-server:9411/
  RABBIT_URI: amqp://guest:guest@rabbitmq:5672
  SPRING_RABBITMQ_HOST: rabbitmq
  SPRING_ZIPKIN_SENDER_TYPE: rabbit
```

Nos módulos que irão usar a fila é necessário configurar a "configuration propertie" que segue:

spring.rabbitmq.host = rabbitmq

Quando o micro serviço de compra e venda deseja mandar uma mensagem para a fila, existem comandos específicos para tanto (veja a figura 2).

```
public void novaCotacao(String de,String para,double valor){
    String msg = de+","+para+","+valor;
    rabbitTemplate.convertAndSend("spring-boot-exchange", "cotacoes.nova", msg);
}
```

Figura 2 – enviando uma mensagem para a fila

No micro serviço que fornece a cotação de moedas, a classe "Receiver" é responsável por receber as mensagens pela fila (outras mensagens são recebidas pelos endpoints) conforme a figura 3. Note que a mensagem vem na forma de string, e precisa ser tratada adequadamente.

```
@Component
public class Receiver {
 @Autowired
 private ServicoCotacao servicoCotacao;

 public void receiveMessage(String message) {
  String dados[] = message.split(",");
  System.out.println("Received <" + message + ">");
  servicoCotacao.novaCotacao(dados[0], dados[1],Double.parseDouble(dados[2]));
 }
}
```

Figura 3 – Recebendo uma mensagem pela fila

Finalmente a classe principal abriga toda a configuração do serviço de fila (ver figura 4).

```
SpringBootApplication
@ComponentScan(basePackages = { "com.bcopstein" })
@EntityScan(basePackages = { "com.bcopstein" })
public class CotacaoMoeda {
        static final String topicExchangeName = "spring-boot-exchange";
        static final String queueName = "spring-boot";

        @Bean
        Queue queue() {      return new Queue(queueName, false); }

        @Bean
        TopicExchange exchange() {  return new TopicExchange(topicExchangeName); }

        @Bean
        // Define que tipo de mensagens este app vai escutar
        Binding binding(Queue queue, TopicExchange exchange) {
         return BindingBuilder.bind(queue).to(exchange).with("cotacoes.#");
        }

        @Bean
        SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,
                MessageListenerAdapter listenerAdapter) {
         SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
         container.setConnectionFactory(connectionFactory);
         container.setQueueNames(queueName);
         container.setMessageListener(listenerAdapter);
         return container;
        }

        @Bean
```

```
        // Define que é a classe que vai receber todas as mensagens
        MessageListenerAdapter listenerAdapter(Receiver receiver) {
         return new MessageListenerAdapter(receiver, "receiveMessage");
        }

        public static void main(String[] args) {
                SpringApplication.run(CotacaoMoeda.class, args);
        }
}
```

O exemplo apresentado utiliza mensagens que trocam strings. Quando existir a necessidade de enviar estruturas mais complexas por parâmetro teremos de converter essas estruturas em sua representação JSON correspondente para então poder enviar. A biblioteca "GSon" é uma solução da Google, fácil de usar, que tem esse objetivo.

Para entender como usar a biblioteca GSon consulte: gson/UserGuide.md at master · google/gson (github.com).

Outro ponto é usar nomes de filas diferentes para diferentes serviços. Caso contrário todas as mensagens serão capturadas por todos os clientes que estiverem ouvindo aquelas fila.

Observação 1: para facilitar recriar as imagens toda a vez que um módulo for compilado acrescentamos o parâmetro "build" no docker-compose.yml como segue:

```
  currency-exchange:
    image: exchange
    build: ./MSP6_currency-exchange-service
```

Nesse caso podemos usar o seguinte comando para forçar que as imagens sejam refeitas:

Docker compose build --no-cache

Depois podemos usar:

Docker compose up

Observação 2: se forem criados novos endpoints não esquecer de definir a rota no gateway !!