MENSAJERÍA ENTRE APLICACIONES DISTRIBUIDAS

Información general	
Duración estimada en minutos:	120
Docente:	Carlos Andrés Florez Villarraga
Guía no.	05

Información de la Guía

La mensajería entre aplicaciones distribuidas se refiere al intercambio de mensajes entre diferentes componentes de software que se ejecutan en sistemas distribuidos, donde estos componentes pueden estar físicamente separados y conectados a través de una red. En este contexto, la mensajería es una forma de comunicación asincrónica en la que las aplicaciones envían y reciben mensajes de manera independiente, sin necesidad de que el emisor y el receptor estén activos al mismo tiempo.

Algunos conceptos clave en la mensajería entre aplicaciones distribuidas incluyen:

- Desacoplamiento: Permite que los componentes de software se comuniquen sin necesidad de conocer los detalles de implementación del otro. Esto facilita la modificación, actualización y mantenimiento de los sistemas distribuidos, ya que los cambios en un componente no necesariamente afectan a los demás.

- Escalabilidad: La mensajería entre aplicaciones distribuidas puede facilitar la escalabilidad horizontal, permitiendo agregar nuevos nodos a la red para manejar cargas de trabajo crecientes.

- Fiabilidad: Los sistemas de mensajería pueden ofrecer garantías de entrega de mensajes, incluso en situaciones de fallos de red o de los propios sistemas. Esto ayuda a garantizar que los mensajes importantes no se pierdan y que las aplicaciones puedan recuperarse de errores de manera robusta.

- Flexibilidad: La mensajería entre aplicaciones distribuidas puede admitir una variedad de patrones de comunicación, como la publicación/suscripción, las colas de mensajes y los eventos, lo que permite adaptarse a diferentes necesidades y casos de uso.

Existen varias herramientas y software que facilitan la implementación de la mensajería entre aplicaciones distribuidas. Algunas de las más populares incluyen: RabbitMQ, Apache Kafka, Redis, Microsoft Azure Service Bus, Amazon Simple Queue Service (SQS) y Amazon Simple Notification Service (SNS).

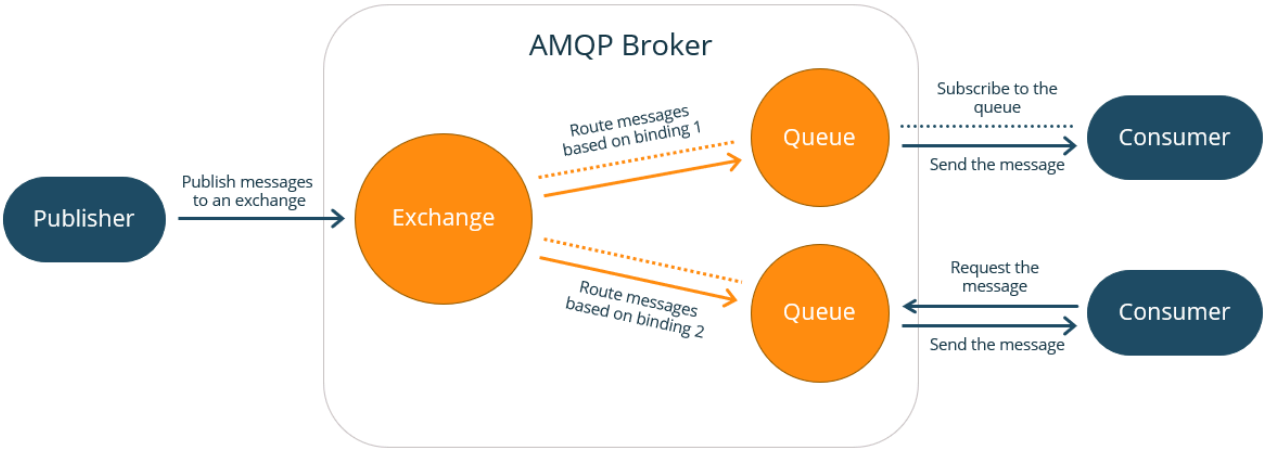
RabbitMQ

En esta ocasión nos centraremos en RabbitMQ. RabbitMQ es un software de mensajería de código abierto que implementa el protocolo Advanced Message Queuing Protocol (AMQP). Es utilizado para la mensajería entre aplicaciones distribuidas y sistemas microservicios. En esencia, RabbitMQ es un intermediario de mensajes que permite a diferentes aplicaciones o componentes de software comunicarse entre sí de manera asíncrona y desacoplada.

Algunos de los conceptos clave en RabbitMQ incluyen:

- Colas (queues): Los mensajes son enviados a colas y luego consumidos por aplicaciones.
- Productores y consumidores: Las aplicaciones que envían mensajes a las colas son llamadas "productores", mientras que las que reciben y procesan los mensajes son llamadas "consumidores".
- Intercambiadores (exchanges): Los intercambiadores son responsables de recibir mensajes de los productores y enviarlos a las colas adecuadas.
- Vinculación (bindings): La relación entre los intercambiadores y las colas se establece a través de las vinculaciones, que especifican cómo los mensajes deben ser enrutados.

El mecanismo de comunicación se puede ejemplificar en la siguiente imagen:



Continuando con el proyecto de la biblioteca, usaremos RabbitMQ para enviar mensajes entre el microservicio del préstamo y los microservicios de clientes y libros, estos mensajes deben validar la existencia de usuarios o registros dados sus ids para que el préstamo se pueda realizar o no.

EJERCICIO

Tome evidencia de los siguientes puntos.

1. Agregue la dependencia de AMQP al microservicio de préstamos, así:

```
implementation 'org.springframework.boot:spring-boot-starter-amqp'
```

2. Cree, una clase de constantes así en el paquete `config` del microservicio de préstamos.

```
package co.edu.eam.biblioteca.config;

public class Constantes {
   public static final String QUEUE = "cola_clientes";
   public static final String EXCHANGE = "rabbit_exchange";
   public static final String ROUTING_KEY = "clientes_routingKey";
```

```
}
```

En esta clase se definen las constantes que serán usadas para centralizar la configuración de RabbitMQ.

3. Cree una nueva clase para la configuración del servicio de mensajería así:

```java
package co.edu.eam.biblioteca.config;

import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
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 MessagingConfig {

   @Bean
   public Queue queue() {
       return new Queue(Constantes.QUEUE);
   }

   @Bean
   public TopicExchange exchange() {
       return new TopicExchange(Constantes.EXCHANGE);
   }

   @Bean
   public Binding binding(Queue queue, TopicExchange exchange) {
       return BindingBuilder.bind(queue).to(exchange).with(Constantes.ROUTING_KEY);
   }

   @Bean
   public MessageConverter converter() {
       return new Jackson2JsonMessageConverter();
   }

   @Bean
   public AmqpTemplate template(ConnectionFactory connectionFactory) {
       final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
       rabbitTemplate.setMessageConverter(converter());
       return rabbitTemplate;
   }
}
```

Este código es una configuración de RabbitMQ en Spring Boot utilizando la anotación `@Configuration` para definir un conjunto de beans relacionados con la mensajería entre aplicaciones distribuidas.

Primero, se define un bean para crear una cola (`queue()`) y un intercambiador de tipo tema (`exchange()`) que se utilizarán para la comunicación entre productores y consumidores. Luego, se define un bean para establecer la vinculación (`binding()`) entre la cola y el intercambiador, especificando una clave de enrutamiento (routing key).

Además, se define un bean para configurar el convertidor de mensajes (`converter()`) que se utilizará para serializar y deserializar los mensajes en formato JSON utilizando Jackson. Finalmente, se define un bean para proporcionar un `AmqpTemplate` (`template()`) que facilita la publicación de mensajes en la cola y el intercambio de mensajes con RabbitMQ, utilizando la conexión proporcionada por `ConnectionFactory`.

4. Modifique el método `validarCodigoCliente()` de la clase `PrestamoServicio` para que quede así:

```
private void validarCodigoCliente(String codigoCliente){

   Object respuesta = rabbitTemplate.convertSendAndReceive(Constantes.EXCHANGE,
Constantes.ROUTING_KEY, codigoCliente);

   if(Objects.isNull(respuesta)){
       throw new RuntimeException("Hubo un error recuperando la información del cliente");
   }

   boolean existe = (Boolean) respuesta;

   if(!existe){
       throw new RuntimeException("El cliente con código: "+codigoCliente+" no existe");
   }

}
```

Este método se encarga de validar si un cliente con un código específico existe en el sistema. El método utiliza un objeto `rabbitTemplate` para enviar un mensaje a la cola (`cola_clientes`) por medio del intercambiador (`rabbit_exchange`) con una clave de enrutamiento (`clientes_routingKey`) y un cuerpo de mensaje que contiene el código del cliente a validar. Luego, espera una respuesta del sistema, que indica si el cliente existe o no. Si la respuesta es nula, se lanza una excepción indicando un error al recuperar la información del cliente. Si la respuesta indica que el cliente no existe, se lanza otra excepción especificando que el cliente con el código dado no existe.

Debe agregar la variable `rabbitTemplate` en la clase, así:

```
private final RabbitTemplate rabbitTemplate;
```

5. Agregue la dependencia de AMQP al microservicio de clientes, así:

```
implementation 'org.springframework.boot:spring-boot-starter-amqp'
```

6. Cree el paquete config y al igual que con el microservicios de préstamos agregue la clase `Constantes` y `MessagingConfig`.

```
package co.edu.eam.biblioteca.config;

public class Constantes {
   public static final String QUEUE = "cola_clientes";
   public static final String EXCHANGE = "rabbit_exchange";
```

```
    public static final String ROUTING_KEY = "clientes_routingKey";
}
```

```
package co.edu.eam.biblioteca.config;

import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
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 MessagingConfig {

    @Bean
    public Queue queue() {
        return new Queue(Constantes.QUEUE);
    }

    @Bean
    public TopicExchange exchange() {
        return new TopicExchange(Constantes.EXCHANGE);
    }

    @Bean
    public Binding binding(Queue queue, TopicExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with(Constantes.ROUTING_KEY);
    }

    @Bean
    public MessageConverter converter() {
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public AmqpTemplate template(ConnectionFactory connectionFactory) {
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(converter());
        return rabbitTemplate;
    }
}
```

7. Cree una clase que permita escuchar los mensajes que se ponen en la cola `"cola_clientes"` de RabbitMQ.

```
package co.edu.eam.biblioteca.config;

import co.edu.eam.biblioteca.service.ClienteServicio;
import lombok.RequiredArgsConstructor;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
@RequiredArgsConstructor
public class Mensajes {
```

```
    private final ClienteServicio clienteServicio;

    @RabbitListener(queues = Constantes.QUEUE )
    public Object receiveMessageAndReply(String codigoCliente) {
        boolean buscado = clienteServicio.existsById(codigoCliente);
        System.out.println("Buscado: (Cliente-service) " + buscado);
        return buscado;
    }

}
```

Esta clase llamada `Mensajes` es un componente de Spring Boot anotado con `@Component`, lo que indica que es un bean administrado por Spring. Además, utiliza la anotación `@RequiredArgsConstructor` para generar automáticamente un constructor que inyecta las dependencias requeridas, en este caso, una instancia de `ClienteServicio`.

Dentro de la clase, se define un método llamado `receiveMessageAndReply`, anotado con `@RabbitListener`, que indica que este método es un listener de RabbitMQ y estará escuchando mensajes en la cola especificada (`"cola_clientes"`). Cuando un mensaje llega a esa cola, este método se ejecuta, recibiendo el mensaje (en este caso, el código del cliente a buscar). Luego, el método utiliza el servicio `clienteServicio` para verificar si existe un cliente con el código proporcionado. Finalmente, retorna un booleano que indica si el cliente fue encontrado o no.

8. En el archivo `docker-compose` agregue el servicio de `rabbitmq`:

```
rabbitmq:
 image: rabbitmq:3-management
 container_name: rabbitmq
 restart: always
 ports:
   - "15672:15672"
   - "5672:5672"
 environment:
   RABBITMQ_DEFAULT_USER: guest
   RABBITMQ_DEFAULT_PASS: guest
```

Y en la parte de `environment` del servicio de préstamos y clientes agregue lo siguiente:

```
spring.rabbitmq.host: rabbitmq
```

Así mismo en la parte del `depends_on` del servicio de préstamos y clientes agregue lo siguiente:

```
- rabbitmq
```

9. Vuelva a compilar el proyecto de préstamos y clientes, haga el build del `Dockerfile` y ejecute el archivo `docker-compose`.

10. Pruebe enviando una solicitud al servicio de préstamos para crear uno nuevo, envíe el `id` de un cliente y verifique que el servicio de mensajería funciona correctamente, tanto para un `id` existente como para uno inexistente.

    Ingrese a http://localhost:15672/ y use `guest` tanto en user como en password (esto lo configuramos en el `docker-compose`). Esta página ofrece un monitor de mensajería de RabbitMQ, allí podemos ver las colas de mensajes, podemos ver si los mensajes se entregan correctamente, si hay mensajes pendientes, etc.

11. Haga lo mismo que hicimos con el microservicio de clientes pero con el microservicios de libros para validar que los isbn si existen al momento de crear un nuevo préstamo, por lo tanto debe crear una cola nueva (`QUEUE_2`) y un routing key nuevo (`ROUTING_KEY_2`) para que el microservicio de préstamos se comunique con el de libros.

12. Con esto ya no es necesario usar `RestTemplate` ni la clase `RestTemplateConfig` en el microservicios de préstamos y así el responsable de comunicar microservicios es RabbitMQ, desacoplando su implementación y mejorando su mantenibilidad.