Serviços de Mensageria

Os serviços de mensageria são componentes essenciais em sistemas distribuídos, permitindo que diferentes partes do sistema se comuniquem de forma eficiente e confiável. Em um sistema assíncrono, serviços ou microsserviços trabalham de forma independente, mas precisam trocar informações entre si. Um serviço de mensageria gerencia essa troca de mensagens, garantindo uma comunicação estruturada e eficaz.

A comunicação assíncrona é crucial para que os serviços operem sem esperar respostas imediatas, aumentando a eficiência e resiliência do sistema. Nesta aula, aprenderemos o que é um serviço de mensageria e como implementá-lo na prática, destacando sua importância e aplicação em diferentes cenários.

Introdução aos Serviços de Mensageria

Os serviços de mensageria são componentes essenciais em sistemas distribuídos, onde múltiplas partes do sistema precisam se comunicar de forma eficiente e confiável. Em um sistema assíncrono, diferentes serviços ou microsserviços trabalham de forma independente, mas precisam trocar informações entre si. Um serviço de mensageria facilita essa comunicação, gerenciando a troca de mensagens de maneira estruturada.

A comunicação assíncrona é vital para garantir que os serviços possam continuar operando sem esperar respostas imediatas. Por exemplo, em um sistema de e-commerce, ao fazer um pedido, várias operações ocorrem: confirmação de pagamento, atualização de estoque, notificação ao cliente, entre outras. Usar serviços de mensageria permite que cada uma dessas

operações seja tratada de forma independente, aumentando a eficiência e a resiliência do sistema.

Exemplo Prático:

```
java</p><p style="margin-right: 0.2pt;text-align: justify">public class PedidoService {</p><p style="margin-right: 0.2pt;text-align: justify">    private final MessageBroker messageBroker;</p><p style="margin-right: 0.2pt;text-align: justify">    public PedidoService(MessageBroker messageBroker) {</p><p style="margin-right: 0.2pt;text-align: justify">        this.messageBroker = messageBroker;</p><p style="margin-right: 0.2pt;text-align: justify">    }</p><p style="margin-right: 0.2pt;text-align: justify">    public void processarPedido(Pedido pedido) {</p><p style="margin-right: 0.2pt;text-align: justify">        // Enviar mensagem para a fila de confirmação de pagamento</p><p style="margin-right: 0.2pt;text-align: justify">        messageBroker.enviarMensagem("filaConfirmacaoPagamento", pedido);</p><p style="margin-right: 0.2pt;text-align: justify">        </p><p style="margin-right: 0.2pt;text-align: justify">        // Enviar mensagem para a fila de atualização de estoque</p><p style="margin-right: 0.2pt;text-align: justify">        messageBroker.enviarMensagem("filaAtualizacaoEstoque", pedido);</p><p style="margin-right: 0.2pt;text-align:
```

```
justify">        </p><p
style="margin-right:                    0.2pt;text-align:
justify">        //
Enviar mensagem para a fila de notificação ao cliente</p><p
style="margin-right:                    0.2pt;text-align:
justify">        messageBroker.enviarMensagem("filaNotificacaoCliente",    pedido);</p><p
style="margin-right:                    0.2pt;text-align:
justify">    }</p><p     style="margin-right:
0.2pt;text-align:      justify">}</p><p      style="margin-right:
0.2pt;text-align: justify">
```

Neste exemplo, o serviço de pedidos envia mensagens para diferentes filas, permitindo que cada operação relacionada ao pedido seja processada de forma independente.

Arquiteturas de Serviços de Mensageria

A arquitetura dos serviços de mensageria pode variar conforme as necessidades do sistema, mas alguns componentes são comuns na maioria das implementações. O message broker, ou corretor de mensagens, é o núcleo dessa arquitetura. Ele recebe mensagens dos produtores e as roteia para os consumidores, garantindo que as mensagens sejam entregues corretamente.

Componentes Principais:

1. Message Broker: Gerencia o roteamento e a entrega de mensagens. Exemplos incluem RabbitMQ, Apache Kafka e ActiveMQ.

2. Produtores: Responsáveis por criar e enviar mensagens ao message broker.

3. Consumidores: Recebem e processam as mensagens do message broker.

Arquiteturas Comuns:

Filas: As mensagens são armazenadas em uma fila e processadas sequencialmente pelos consumidores.

Tópicos: Permitem categorizar mensagens, onde diferentes consumidores podem assinar e receber apenas as mensagens de interesse.

Exemplo Prático:


```
java</p><p style="margin-right: 0.2pt;text-align: justify">@Configuration</p><p style="margin-right: 0.2pt;text-align: justify">public class RabbitMQConfig {</p><p style="margin-right: 0.2pt;text-align: justify">    @Value("${rabbitmq.queue}")</p><p style="margin-right: 0.2pt;text-align: justify">    private String queue;</p><p style="margin-right: 0.2pt;text-align: justify">    @Value("${rabbitmq.exchange}")</p><p style="margin-right: 0.2pt;text-align: justify">    private String exchange;</p><p style="margin-right: 0.2pt;text-align: justify">    @Value("${rabbitmq.routingkey}")</p><p style="margin-right: 0.2pt;text-align: justify">    private String routingKey;</p><p style="margin-right: 0.2pt;text-align: justify">    @Bean</p><p style="margin-right: 0.2pt;text-align: justify">    Queue queue() {</p><p style="margin-right: 0.2pt;text-align: justify">        return new Queue(queue, false);</p><p style="margin-right: 0.2pt;text-align: justify">    }</p><p style="margin-right: 0.2pt;text-align:
```

justify">   @Bean   DirectExchange exchange() {   return new DirectExchange(exchange);   }   @Bean   Binding binding(Queue queue, DirectExchange exchange) {   return BindingBuilder.bind(queue).to(exchange).with(routingKey);   } }

Essa configuração cria uma fila, um exchange e define o roteamento das mensagens.

Frameworks Populares de Serviços de Mensageria

Os frameworks de mensageria facilitam a implementação e gestão dos serviços de mensageria em diferentes linguagens e ambientes. Aqui estão alguns dos mais populares:

1. Spring Cloud Stream: Facilita a criação de microsserviços em Java para o processamento de mensagens. Integra-se bem com o ecossistema Spring.

2. Apache Kafka Streams: Focado em processamento de dados em tempo real e orientado a eventos, utilizado principalmente com Java.

3. RabbitMQ: Amplamente utilizado por ser multiplataforma e fácil de integrar com diferentes linguagens, incluindo Java e JavaScript.

4. Apache Pulsar: Suporta Java, Python e Go, é escalável e utilizado para mensagens e eventos distribuídos.

5. Kafka JS: Biblioteca para JavaScript que permite a configuração e consumo de mensagens usando clusters de Kafka.

Exemplo Prático (RabbitMQ):

```
java</p><p style="margin-right: 0.2pt;text-align: justify">@RestController</p><p style="margin-right: 0.2pt;text-align: justify">@RequestMapping("/api/pedidos")</p><p style="margin-right: 0.2pt;text-align: justify">public class PedidoController {</p><p style="margin-right: 0.2pt;text-align: justify">    private final PedidoService pedidoService;</p><p style="margin-right: 0.2pt;text-align: justify">    public PedidoController(PedidoService pedidoService) {</p><p style="margin-right: 0.2pt;text-align: justify">        this.pedidoService = pedidoService;</p><p style="margin-right: 0.2pt;text-align: justify">    }</p><p style="margin-right: 0.2pt;text-align: justify">    @PostMapping</p><p style="margin-right: 0.2pt;text-align: justify">    public ResponseEntity&lt;Pedido&gt; criarPedido(@RequestBody Pedido pedido) {</p><p style="margin-right: 0.2pt;text-align: justify">        pedidoService.processarPedido(pedido);</p><p style="margin-right: 0.2pt;text-align:
```

justify">   return ResponseEntity.status(HttpStatus.CREATED).body(pedido);   } }

Neste exemplo, o controlador de pedidos recebe uma solicitação para criar um pedido e utiliza o serviço de pedidos para enviar mensagens para diferentes filas.

Implementação Prática com RabbitMQ

A implementação prática com RabbitMQ envolve a instalação e configuração do servidor RabbitMQ e a integração com a aplicação. O RabbitMQ utiliza o Erlang, por isso é necessário instalar o Erlang antes do RabbitMQ. Após a instalação, configure as variáveis de ambiente e reinicie a máquina para aplicar as configurações.

Passos de Instalação:

1. Instale o Erlang.

2. Instale o RabbitMQ.

3. Configure as variáveis do ambiente.

4. Reinicie a máquina.

5. Habilite o plugin de gerenciamento do RabbitMQ.

Exemplo Prático (Configuração no Spring Boot):

properties # application.properties <p style="margin-right:

0.2pt;text-align: justify">spring.rabbitmq.host=localhost spring.rabbitmq.port=5672 spring.rabbitmq.username=guest spring.rabbitmq.password=guest

Classe de Configuração:

java @Configuration public class RabbitMQConfig {   @Bean   public Queue queue() {   return new Queue("filaPedidos", false);   }   @Bean   public DirectExchange exchange() {   return new DirectExchange("exchangePedidos");   }   @Bean   public Binding binding(Queue queue, DirectExchange exchange) { <p style="margin-right: 0.2pt;text-align:

justify">   return BindingBuilder.bind(queue).to(exchange).with("routingKeyPedidos");   }   @Bean   public MessageConverter jsonMessageConverter() {   return new Jackson2JsonMessageConverter();   }   @Bean   public AmqpTemplate rabbitTemplate(ConnectionFactory connectionFactory) {   final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);   rabbitTemplate.setMessageConverter(jsonMessageConverter());   return rabbitTemplate;   } }

Consumer:

java @Component public class PedidoConsumer {   @RabbitListener(queues = "filaPedidos")   public void receberPedido(Pedido pedido) {   System.out.println("Pedido recebido: " + pedido);   // Processar o pedido...   } }

Service:

java @Service public class PedidoService {   private final AmqpTemplate amqpTemplate;   private final String exchange;   private final String routingKey;   public PedidoService(AmqpTemplate amqpTemplate,   &

nbsp;   @Value("${spring.rabbitmq.exchange}") String exchange,</p><p style="margin-right: 0.2pt;text-align: justify">   @Value("${spring.rabbitmq.routingKey}") String routingKey) {   this.amqpTemplate = amqpTemplate;   this.exchange = exchange;   this.routingKey = routingKey;   }   public void processarPedido(Pedido pedido) {   amqpTemplate.convertAndSend(exchange, routingKey, pedido);   } }

Essa implementação ilustra a configuração e uso do RabbitMQ para enviar e processar mensagens de pedidos, demonstrando uma integração prática de um serviço de mensageria em uma aplicação Spring Boot.

GitHub da Disciplina

Você pode acessar o repositório da disciplina no GitHub a partir do seguinte link:

https://github.com/FaculdadeDescomplica/Framework. Esse espaço é o seu portal para mergulhar fundo no universo da aprendizagem interativa. Nele, você encontrará todos os códigos, além dos links para os arquivos e dados.

Conteúdo Bônus

Um ótimo conteúdo bônus gratuito para alunos de graduação sobre Framework e Serviços de Mensageria é o curso "Messaging with RabbitMQ" disponível no site Spring Framework. Este curso oferece uma introdução detalhada sobre como integrar serviços de mensageria com aplicações Spring Boot utilizando RabbitMQ.

Título: Messaging with RabbitMQ

Plataforma: Spring.io

Esse material é valioso porque fornece exemplos práticos e instruções passo a passo sobre a configuração e uso de RabbitMQ com Spring Boot, ajudando os alunos a entenderem como implementar serviços de mensageria de forma eficiente.

Referência Bibliográfica

CARDOSO, L. da C. **Design de aplicativos**. Intersaberes: 2022

ELMASRI, R.; NAVATHE, S. B. **Sistemas de banco de dados**. 7.ed. Pearson: 2018.

JOÃO, B. do N. **Usabilidade e interface homem-máquina**. Pearson: 2017

LEAL, G. C. L. **Linguagem, programação e banco de dados**: guia prático de aprendizagem. Intersaberes: 2015.

MEDEIROS, L. F. de. **Banco de dados**: princípios e prática. Intersaberes: 2013.

PUGA, S.; FRANÇA, E.; GOYA, M. **Banco de dados**: implementação em SQL, PL/SQL e Oracle 11g. Pearson: 2013.

SETZER, V. W.; SILVA, F. S. C. **Bancos de dados**. Blucher: 2005.

VICCI, C. (Org.). **Banco de dados**. Pearson: 2014.

Ir para exercício