**Sistema de Vendas: Exemplo de implementação de uma Arquitetura Orientada a Eventos utilizando Spring Boot e RabbitMQ**

**José Ricardo Serathiuk da Silveira[[1]](#footnote-1)**

**Resumo**

Documentação de um exemplo simples de uma implementação de uma arquitetura orientada a eventos utilizando Spring Boot 3.0 e RabbitMQ.

**Palavras-chave:** spring boot, rabbitmq, arquitetura orientada a eventos, java.

**Abstract**

Documentation of a simple example of an implementation of an event-driven architecture using Spring Boot 3.0 and RabbitMQ.

**Keywords:** spring boot, rabbitmq, arquitetura orientada a eventos, java.

**Introdução**

O objetivo desse documento é explicar a implementação de uma arquitetura orientada à eventos utilizando Spring Boot 3.0 e RabbitMQ. O Spring Boot 3.0 foi escolhido por ser um framework com grande adesão no mercado e que o autor tem familiaridade e que ajuda no desenvolvimento da soluçäo. O RabbitMQ foi escolhido para fins de aprendizagem. O autor possui mais familiaridade e experiência profissional com outras ferramentas de mensageria, como o Amazon SNS/SQS. A proposta é não ter nenhuma lógica de negócio mais profunda implementada, mas sim apenas a estrutura de comunicação funcionando. Também não foi pensado sobre transações distribuídas e outras questões mais complexas necessárias para o desenvolvimento de sistemas distribuídos funcionando em produção.

**Estrutura conceitual do projeto**

Serão criados 4 filas (queues) e 2 tópicos (exchanges). E cada tópico irá enviar mensagem para 3 filas. Um tópico será relativo a vendas de serviço e outro vendas de materiais. A diferença entre os 2 será apenas a emissão de nota, que para um caso é uma nota de Material e para outro uma nota de Serviço.

As filas que serão criadas:

* q.vendas.pagamento: Responsável por gerar as informações de pagamento
* q.vendas.inventario: Responsável por gerar por processar o inventário da venda
* q.vendas.nfe: Responsável por gerar a NF-e da venda
* q.venda.nfse: Responsável por gerar a NFS-e da venda

Os tópicos (exchanges) a serem criados:

* ex.venda.material: Tópico responsável por processar vendas de materiais.
* ex.venda.servico: Tópico responsável por processar vendas de serviços.

O vínculo (bindings) entre as filas e tópicos será o seguinte:

Diagrama

Descrição gerada automaticamente

Figura 1 Mapa de vínculos entre as filas e tópicos

**Estrutura inicial do projeto**

A estrutura inicial do projeto é a sugerida pela ferramenta de automação e gerenciamento de pacotes Maven e foi gerada pela ferramenta online própria do framework Spring, a Spring initializr (<https://start.spring.io>). Foi criado um projeto do tipo Maven, com linguagem Java e com o Spring Boot na versão 3.0.5, utilizando o empacotamento Jar e Java 17. As dependências escolhidas foram a Spring Web e a Spring for RabbitMQ.

Tela de geraçäo da estrutura inicial do projeto


Figura 2: Tela de geraçäo da estrutura inicial do projeto.

Foi criado um arquivo docker-compose.yaml na raiz do projeto, que utilizando as ferramentas Docker e Docker Compose, será responsável por criar um servidor RabbitMQ para o exemplo

*/docker-compose.yaml:*

version: '3.8'  
  
networks:  
 vendas:  
  
services:  
 rabbitmq:  
 image: rabbitmq:3-management-alpine  
 container\_name: 'rabbitmq'  
 ports:  
 - 5672:5672  
 - 15672:15672  
 networks:  
 - vendas

Para iniciar o RabbitMQ é só executar um ‘docker compose up -d’ (ou ‘docker-compose up -d’ dependendo da instalação) com o docker devidamente instalado.

**Configuraçäo do projeto**

Primeiramente, foi alterado o application.properties com as configurações de conexão do RabbitMQ:

*/src/main/resources/application.properties:*

spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest

Após isso foi adicionada a anotação @EnableRabbit na classe VendasApplication (classe de inicialização do projeto).

*/src/main/java/com/serathiuk/vendas/VendasApplication.java*

@EnableRabbit  
@SpringBootApplication  
public class VendasApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(VendasApplication.class, args);  
 }  
  
}

Foi criada a classe AppConfig para configurar o Spring para utilizar corretamente o RabbitMQ com as mensagens com serialização em JSON, e também para que sejam criadas as queues, bindings e exchanges na inicialização da aplicação, caso não existam.

*/src/main/java/com/serathiuk/vendas/AppConfig.java*

@Configuration

public class AppConfig {  
  
  
 public static final String *TOPICO\_VENDA\_SERVICO* = "ex.venda.servico";  
 public static final String *TOPICO\_VENDA\_MATERIAL* = "ex.venda.material";  
 public static final String *QUEUE\_PAGAMENTO* = "q.vendas.pagamento";  
 public static final String *QUEUE\_INVENTARIO* = "q.vendas.inventario";  
 public static final String *QUEUE\_NFE*= "q.vendas.nfe";  
 public static final String *QUEUE\_NFSE*= "q.vendas.nfse";  
  
 *// Configura o Spring Boot para utilizar JSON como o formato de serializaçäo das mensagens* @Bean  
 public RabbitTemplate rabbitTemplate(final ConnectionFactory connectionFactory) {  
 final var rabbitTemplate = new RabbitTemplate(connectionFactory);  
 rabbitTemplate.setMessageConverter(producerJackson2MessageConverter());  
 return rabbitTemplate;  
 }  
  
 *// Pré-requisito para o Spring Boot utilizar JSON para fazer as conversões das mensagens do RabbitMQ* @Bean  
 public Jackson2JsonMessageConverter producerJackson2MessageConverter() {  
 return new Jackson2JsonMessageConverter();  
 }  
  
 *// Faz a criaçäo das queues, exchanges e bindings caso não existam ainda no RabbitMQ* @Bean  
 public AmqpAdmin amqpAdmin(RabbitTemplate rabbitTemplate) {  
 var ampq = new RabbitAdmin(rabbitTemplate);  
 ampq.declareQueue(new Queue(*QUEUE\_PAGAMENTO*));  
 ampq.declareQueue(new Queue(*QUEUE\_INVENTARIO*));  
 ampq.declareQueue(new Queue(*QUEUE\_NFE*));  
 ampq.declareQueue(new Queue(*QUEUE\_NFSE*));  
  
 ampq.declareExchange(new FanoutExchange(*TOPICO\_VENDA\_MATERIAL*));  
 ampq.declareExchange(new FanoutExchange(*TOPICO\_VENDA\_SERVICO*));  
  
 createBinding(ampq, *TOPICO\_VENDA\_MATERIAL*, *QUEUE\_PAGAMENTO*);  
 createBinding(ampq, *TOPICO\_VENDA\_MATERIAL*, *QUEUE\_INVENTARIO*);  
 createBinding(ampq, *TOPICO\_VENDA\_MATERIAL*, *QUEUE\_NFE*);  
 createBinding(ampq, *TOPICO\_VENDA\_SERVICO*, *QUEUE\_PAGAMENTO*);  
 createBinding(ampq, *TOPICO\_VENDA\_SERVICO*, *QUEUE\_INVENTARIO*);  
 createBinding(ampq, *TOPICO\_VENDA\_SERVICO*, *QUEUE\_NFSE*);  
 return ampq;  
 }  
  
 *// Método utilitário para criar os bindings.* private void createBinding(RabbitAdmin admin, String exchange, String queueName) {  
 admin.declareBinding(new Binding(queueName, Binding.DestinationType.*QUEUE*, exchange, "", new HashMap<>()));  
 }  
  
}

**Implementações dos assinantes**

Foi implementada 1 assinante para cada fila. As implementações apenas apresentam as informações recebidas no log, nem nenhum tipo de manipulação adicional. As implementações estão no pacote *com.serathiuk.vendas.assinantes*. As classes são InventarioListener, NfeListener, NfseListener e PagamentoListener.

*/src/main/java/com/serathiuk/vendas/assinantes/InventarioListener.java*

@Service  
public class InventarioListener {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(InventarioListener.class);  
  
 @RabbitListener(queues = AppConfig.*QUEUE\_INVENTARIO*)  
 public void execute(Venda venda) {  
 *logger*.info("Criando Inventario. Descriçäo: "+venda.getDescricao()+" Valor: "+venda.getValor()+" Quantidade: "+venda.getQuantidade());  
 }  
}

*/src/main/java/com/serathiuk/vendas/assinantes/NfeListener.java*

@Service  
public class NfeListener {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(NfeListener.class);  
  
 @RabbitListener(queues = AppConfig.*QUEUE\_NFE*)  
 public void execute(Venda venda) {  
 *logger*.info("Criando NF-e. Descriçäo: "+venda.getDescricao()+" Valor: "+venda.getValor()+" Quantidade: "+venda.getQuantidade());  
 }  
  
}

*/src/main/java/com/serathiuk/vendas/assinantes/NfseListener.java*

@Service  
public class NfseListener {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(NfseListener.class);  
  
 @RabbitListener(queues = AppConfig.*QUEUE\_NFSE*)  
 public void execute(Venda venda) {  
 *logger*.info("Criando NFS-e. Descriçäo: "+venda.getDescricao()+" Valor: "+venda.getValor()+" Quantidade: "+venda.getQuantidade());  
 }  
}

*/src/main/java/com/serathiuk/vendas/assinantes/PagamentoListener.java*

@Service  
public class PagamentoListener {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(PagamentoListener.class);  
  
 @RabbitListener(queues = AppConfig.*QUEUE\_PAGAMENTO*)  
 public void execute(Venda venda) {  
 *logger*.info("Criando Pagamento. Descriçäo: "+venda.getDescricao()+" Valor: "+venda.getValor()+" Quantidade: "+venda.getQuantidade());  
 }  
}

**Implementações dos clientes geradores**

Foi criada a classe VendaController, que possuí 2 endpoints REST, o *`/venda/material`*, que envia a solicitação para o tópico *ex.venda.material*, e o endpoint *`/venda/serviço`* que envia a solicitação para o tópico *ex.venda.servico*.

*/src/main/java/com/serathiuk/vendas/controller/VendaController.java*

@RestController  
public class VendaController {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(VendaController.class);  
  
 @Autowired  
 private RabbitTemplate rabbitTemplate;  
  
 @PostMapping(value = "/venda/material")  
 public ResponseEntity<?> criarVendaDeMaterial(@RequestBody Venda venda) {  
 *logger*.info("Criando Venda de Material. Descriçäo: "+venda.getDescricao()+" Valor: "+venda.getValor()+" Quantidade: "+venda.getQuantidade());  
  
 rabbitTemplate.convertAndSend(AppConfig.*TOPICO\_VENDA\_MATERIAL*, "", venda);  
  
 return ResponseEntity.*ok*().build();  
 }  
  
 @PostMapping(value = "/venda/servico")  
 public ResponseEntity<?> criarVendaDeServico(@RequestBody Venda venda) {  
 *logger*.info("Criando Venda de Servico. Descriçäo: "+venda.getDescricao()+" Valor: "+venda.getValor()+" Quantidade: "+venda.getQuantidade());  
  
 rabbitTemplate.convertAndSend(AppConfig.*TOPICO\_VENDA\_SERVICO*, "", venda);  
  
 return ResponseEntity.*ok*().build();  
 }  
  
}

**Requisiçäo de Venda de Serviço**

Foi criada uma requisição via o software Postman.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Figura 3: Venda de Materiais via endpoint utilizando o Postman

c

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Figura 4: Retorno do Venda de Materiais

**Requisiçäo de Venda de Serviço**

Foi criada uma requisição via o software Postman.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Figura 5: Venda de Serviços via endpoint utilizando o Postman

Venda de Serviços via endpoint utilizando o Postman.

Texto

Descrição gerada automaticamente

Figura 6: Retorno do Venda de Serviços

**Considerações Finais**

A arquitetura funcionou conforme o esperado. As mensagens estão sendo passadas corretamente para os clientes consumidores.

O código-fonte da solução pode ser acessado em https://github.com/serathiuk/vendasutfpr.

**Referências**

**Sending and receiving JSON messages with Spring Boot AMQP and RabbitMQ**. Disponível em: <https://thepracticaldeveloper.com/produce-and-consume-json-messages-with-spring-boot-amqp/>. Acesso em: 25 mar. 2023.

‌**RabbitMQ In Practice**. Disponível em: <https://www.udemy.com/course/rabbitmq-in-practice>. Acesso em: 25 mar. 2023.

**Getting Started | Messaging with RabbitMQ**. Disponível em: <https://spring.io/guides/gs/messaging-rabbitmq/>. Acesso em: 25 mar. 2023.

‌

‌

1. Especialista em Arquitetura de Software Distribuído pela PUC Minas e Especialista em Engenharia de Software Ágil pela Universidade Positivo. E-mail: ricardo@serathiuk.com [↑](#footnote-ref-1)