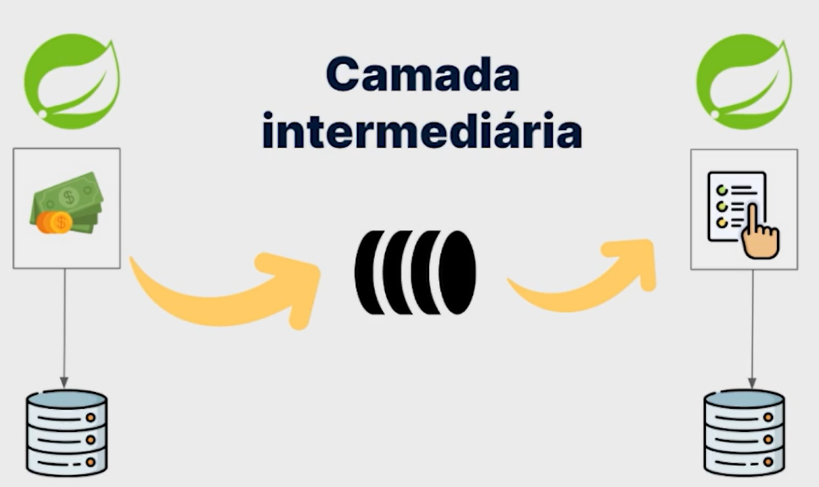
**MÓDULO 1 – REQUISIÇÃO ASSÍNCRONA**

Antes utilizava-mos a comunicação síncrona, quando um pagamento era confirmado, uma mensagem era enviada ao MS de pedidos, e para isso utilizamos o Spring Cloud OpenFeign.

Isso era feito por meio de uma interface PedidoClient e dentro de uma rota de PUT que atualizava o status do pagamento e salvava no repository.

Contras: Caso o MS de pagamento estivesse fora do ar, nós acabamos setando o Status como “confirmado sem integração”, não é a melhor maneira e há uma transferência de responsabilidade. Também tem um alto acoplamento, coisa q vai de contra a base de um MS.

Agora vamos adicionar uma camada intermediária, com o RabbitMQ, entre a comunicação desses 2 MS. Dessa forma os MS consumidores da mensagem nem precisam ser em Java.

Padrão pub/sub e FILA de mensagens.

As Exchanges são as responsáveis por pegar a mensagem publicada pelo Publisher e analisa, processa e avalia para qual/quais filas ela será destinada.

Também é possível programar o tempo que uma mensagem pode ficar disponível para ser consumida.

A distribuição de mensagens para os subscribers da fila é feito de maneira balanceada.

docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13-management

Acesse <http://localhost:15672/>, por padrão o username e password é guest.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**MÓDULO 2 – PADRÃO PUB/SUB**

Vamos adicionar a seguinte dependência ao MS de pagamentos:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

Também vamos adicionar ao application.properties:

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=victor

spring.rabbitmq.password=200694

Vamos criar a pasta amqp e dentro dela a classe que irá configurar o rabbit:

@Configuration

public class PagamentoAMQPConfiguration {

**Vamos criar um RabbitAdmin que poderá realizar ações administrativas.**

@Bean

public RabbitAdmin criarRabbitAdmin(ConnectionFactory connectionFactory) {

return new RabbitAdmin(connectionFactory);

}

**Vamos criar a nossa fila não durável**

@Bean

public Queue criaFila() {

return new Queue("pagamento.concluído", false);

}

**Agora vamos criar um listener que inicializa o Rabbit assim que a aplicação rodar:**

@Bean

public ApplicationListener<ApplicationReadyEvent> inicializaAdmin(RabbitAdmin rabbitAdmin) {

return event -> rabbitAdmin.initialize();

}

}

Vamos utilizar uma classe helper chamada RabbitTemplate e injetá-la no controller de Pagamento

Vamos disparar uma mensagem ao criar um pagamento, então iremos alterar a rota post Mapping:

@Autowired

private **RabbitTemplate** rabbitTemplate;

@PostMapping

public ResponseEntity<PagamentoDto> cadastrar(@RequestBody @Valid PagamentoDto dto, UriComponentsBuilder uriBuilder) {

PagamentoDto pagamento = service.criarPagamento(dto);

URI endereco = uriBuilder.path("/pagamentos/{id}").buildAndExpand(pagamento.getId()).toUri();

**Enviando mensagem para o RabbitMQ**

**Message** message = new Message(("Criei um pagamento com o id " + pagamento.getId()).**getBytes()**);

**A mensagem, por padrão, é em bytes.**

rabbitTemplate.**send**("pagamento.concluido", message);

**.send(nome da fila, mensagem)**

return ResponseEntity.created(endereco).body(pagamento);

}

**Consumindo mensagens:**

Para “escutar/ouvir” mensagens de uma fila, vamos precisar do RabbitListener, então vamos criar a pasta amq no ms de pedidos, e nele vamos criar a classe PagamentosListener que irá consumir as mensagens geradas pelo ms de pagamentos:

**@Component**

public class **PagamentoListener** {

**@RabbitListener**(queues = "pagamento.concluido") 🡪 nome da fila

public void **recebeMensagem**(Message mensagem) {

System.out.println("Recebi a mensagem " + mensagem.toString());

}

}

Resumo:

Para fazer o envio das mensagens para o RabbitMQ, ao realizar uma requisição do tipo POST em nosso endpoint, utilizamos o **RabbitTemplate**.

Já a anotação **@RabbitListener**, que recebe como parâmetro um array de Strings correspondentes aos nomes das filas que serão consumidas. Dessa forma, assim que a aplicação é inicializada, o método com essa anotação começará a ser executado e as mensagens da(s) fila(s) serão consumida(s).

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**MÓDULO 3 – APRIMORANDO AS MENSAGENS**

Vamos adicionar os 2 métodos abaixo para modificar o conversor de mensagens padrão do RabbitTemplate para converter um objeto Message em JSON, então vamos modificar nossa classe PagamentoAMPQConfiguration:

Também vamos precisar criar a classe PedidoAMPQConfiguration, dentro do ms de pedido e adicionar esses 2 métodos nela.

**@Bean**

public **Jackson2JsonMessageConverter** **messageConverter**(){

return new Jackson2JsonMessageConverter();

}

**@Bean**

public **RabbitTemplate** **rabbitTemplate**(**ConnectionFactory** connectionFactory,

**Jackson2JsonMessageConverter** messageConverter){

RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);

rabbitTemplate.**setMessageConverter**(messageConverter);

return rabbitTemplate;

}

Agora vamos criar um ms de avaliação, que será notificado cada vez que um pedido receber o status APROVADO, e assim solicitar uma avaliação do cliente.

Então, agora a nossa fila terá 2 consumidores e para não ter q criar várias filas, vamos usar uma Exchange, do tipo Fanout, essa Exchange irá enviar a mensagem para todas as filas que estão relacionadas com ela. Existe um simulador para essas exchanges:

<https://tryrabbitmq.com/>

Tipos de Exchange:

**Direct**: Geralmente usada quando desejamos enviar mensagens para um consumidor específico. Para isso, utilizamos uma **routing key**, que é uma chave enviada junto com a mensagem para que a exchange identifique qual a fila irá receber a mensagem.

**Fanout**: Quando enviamos uma mensagem para uma exchange desse tipo, ela vai ser **enviada para todas as filas que estiverem ligadas a essa exchange**. Ou seja, se existirem 30 filas ligadas a essa exchange, essas 30 filas receberão a mensagem. **Para conectar uma fila a uma exchange é preciso criar um bind**, que é uma relação (ou vínculo) entre uma fila e uma exchange. Nomeia-se como binding key essa chave de ligação entre a fila e a exchange.

**Headers**: É um tipo muito menos usado, mas basicamente ignora o routing key e passa no próprio cabeçalho da mensagem para qual binding key deve ser encaminhada a mensagem.

**Topic**: A palavra-chave desse tipo de exchange é flexibilidade. Você pode nomear as binding keys de uma forma a criar padrões e/ou regras para que o envio seja de forma personalizada. Essas binding keys são descritas como se fossem expressões regulares. Dessa forma, podemos fazer composições para que mais de um consumidor receba as mensagens.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**MÓDULO 4 – FALHAS E RESILIÊNCIA**

Podemos estabelecer um número máximo de tentativas de ler a mensagem, para evitarmos um loop infinito, então sete no application.properties:

spring.rabbitmq.listener.simple.retry.enabled=true

spring.rabbitmq.listener.simple.retry.max-attempts=4

Para que a mensagem não seja excluída da fila caso o listener não consiga processá-la:

Podemos jogar todas essas mensagens de erro para uma exchange auxiliar e fila auxiliar ex: pagamentos.dlx e pagamentos.detalhes-avaliacao-errors, essa estratégia recebe a sigla **DLQ**.

Vamos criar essa Exchange e fila auxiliar no próprio ms de avaliação.

Agora na fila principal, nos vamos configurar para que caso ocorra algum erro, ele já saberá para qual Exchange e fila deverá ser enviado.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**MÓDULO 5 – ALTA DISPONIBILIDADE**

Após fazer uma tratativa de erros, podemos mover as mensagens da fila de erro para a fila normal, para isso precisamos de 2 plugins, o shovel e o shovel management, para habilitá-los digite o seguinte comando no terminal do container do rabbitmq:

**rabbitmq-plugins enable rabbitmq\_shovel rabbitmq\_shovel\_management**

Agora vamos criar um cluster com várias instâncias do rabbit rodando, para n corrermos o risco de ficar sem esse serviço essencial para o alura food.

1º vamos criar uma rede no Docker para que todos os containers consigam se comunicar por ela