RABBITMQ COM C#

Thiago Moura

2024

Resumo **3**

1 Introdução **4**

2 Introdução ao Docker e Instalação do RabbitMQ **5**

3 Producer **8**

3.1 Implementando um producer 9

4 Consumer **13**

4.1 Implementando um consumer 14

5 Cenários do Mundo Real com RabbitMQ **17**

6 Conclusão **19**

7 Glossário **20**

Referências Bibliográficas **21**

RESUMO

Este guia prático apresenta uma introdução ao uso do RabbitMQ com C#, incluindo a configuração do ambiente com Docker, a implementação de produtores e consumidores de mensagens, além de exemplos de cenários do mundo real. Este material é ideal para desenvolvedores que desejam integrar mensageria em suas aplicações e promover o desacoplamento e a escalabilidade.

1 INTRODUÇÃO

RabbitMQ é um dos brokers de mensagens mais populares utilizados para promover a comunicação assíncrona entre partes de um sistema. Ele é utilizado em diversos cenários onde precisamos de alta escalabilidade, desacoplamento e processamento de mensagens de maneira eficiente. Neste artigo, vamos explorar como implementar um producer e um consumer em C#, utilizando RabbitMQ.

2 INTRODUÇÃO AO DOCKER E INSTALAÇÃO DO RABBITMQ

Introdução ao Docker e Instalação do RabbitMQ

Docker é uma plataforma que facilita a criação, execução e gerenciamento de containers - ambientes isolados que contêm todas as dependências necessárias para uma aplicação funcionar. Docker é amplamente utilizado para criar ambientes de desenvolvimento consistentes e para facilitar o deployment de aplicações.

Para utilizar RabbitMQ de maneira simples e rápida, podemos rodá-lo dentro de um container Docker. Abaixo estão os passos para instalar e rodar o RabbitMQ usando Docker:

1. **Instalar Docker:** Se ainda não tiver o Docker instalado, você pode baixá-lo e instalá-lo através do site oficial: Docker Download.
2. **Rodar RabbitMQ com Docker:** Uma vez que o Docker esteja instalado, execute o comando abaixo para baixar e rodar uma instância do RabbitMQ:

```
docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management
```

Explicação do Comando:

- docker run: Cria e roda um novo container.
- -d: Roda o container em modo “detached” (em segundo plano).
- --name rabbitmq: Nomeia o container como “rabbitmq”.
- -p 5672:5672: Mapeia a porta 5672 do container para a porta 5672 do host. Essa é a porta padrão para comunicação com o RabbitMQ.
- -p 15672:15672: Mapeia a porta 15672 do container para a porta 15672 do host. Essa é a porta para acessar o painel de administração do RabbitMQ.
- rabbitmq:management: Usa a imagem oficial do RabbitMQ com o plugin de gerenciamento habilitado.

Acessar o Painel de Administração do RabbitMQ

- Após rodar o container, você pode acessar o painel de administração do RabbitMQ através do navegador usando o endereço: http://localhost:15672.
- O login padrão é usuário: guest e senha: guest.

RabbitMQ™

Username: *

Password: *

Login

RabbitMQ™ RabbitMQ 4.0.3 Erlang 26.2.5.5

Refreshed 2024-11-15 20:53:33 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@b0cef06e667d**

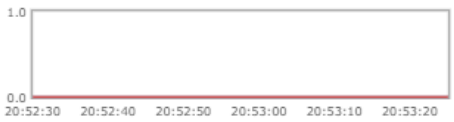
User **guest** Log out

Overview | Connections | Channels | Exchanges | Queues and Streams | Admin

Overview

Totals

Queued messages last minute ?

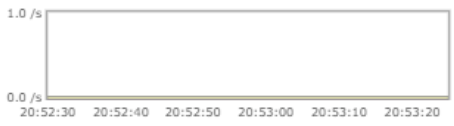


Ready 0

Unacked 0

Total 0

Message rates last minute ?



Publish	0.00/s	Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Get (empty)	0.00/s		

Global counts ?

Connections: 0 | Channels: 0 | Exchanges: 7 | Queues: 1 | Consumers: 0

Nodes

Name	File descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Cores	Info	Reset stats	+/-
rabbit@b0cef06e667d	43 1048576 available	451 1048576 available	146 MiB 4.6 GiB high watermark	224 GiB 48 MiB low watermark	5h 12m	16	basic 2 rss	This node All nodes	

Churn statistics

Ports and contexts

Export definitions

Import definitions

HTTP API | Documentation | Tutorials | New releases | Commercial edition | Commercial support | Discussions | Discord | Plugins | GitHub

3 PRODUCER

O producer é a parte que gera as mensagens e as envia para uma fila no RabbitMQ. Abaixo estão os passos essenciais para criar um produtor.

1. **Criar Conexão:** O primeiro passo é criar uma conexão com o RabbitMQ usando ConnectionFactory e CreateConnectionAsync(). Esta conexão é necessária para estabelecer o link de comunicação com o broker RabbitMQ.

2. **Criar Canal:** Em seguida, criamos um canal com CreateChannelAsync(). O canal permite que o produtor interaja com o RabbitMQ, possibilitando declarar filas e publicar mensagens.

3. **Declarar a Fila**: Antes de publicar mensagens, é necessário declarar a fila para garantir que ela exista. Usamos QueueDeclareAsync() para declarar a fila “hello”. Configurações importantes incluem durable: false, indicando que a fila não será persistida após a reinicialização do RabbitMQ.

4. **Criar e Serializar Mensagem:** O próximo passo é criar um objeto Aluno e popular seus dados. O objeto é serializado em formato JSON, e depois convertido para bytes (UTF-8), uma vez que o RabbitMQ trabalha com dados binários.

5. **Publicar a Mensagem:** Por fim, utilizamos BasicPublishAsync() para enviar a mensagem para a fila “hello”. Utilizamos a exchange padrão e uma routingKey que identifica a fila que irá receber a mensagem.

3.1 IMPLEMENTANDO UM PRODUCER

```
using RabbitMQ.Client;

using System.Text;

using System.Text.Json;


namespace Producers

{

    public class Program

    {

        static async Task Main(string[] args)

    {

        var factory = new ConnectionFactory { HostName = "localhost" };

        using var connection = await factory.CreateConnectionAsync();

        using var channel = await connection.CreateChannelAsync();


        await channel.QueueDeclareAsync(queue: "hello",

                                        durable: false,

                                        exclusive: false,

                                        autoDelete: false,

                                        arguments: null);
```

```
Console.Write("Digite uma mensagem e aperte ENTER: ");


while (true)
{
    var aluno = new Aluno();


    Console.Write("NOME: ");
    aluno.Nome = Console.ReadLine();
    Console.Write("E-MAIL: ");
    aluno.Email = Console.ReadLine();


    if (string.IsNullOrEmpty(aluno.Nome)
        && string.IsNullOrEmpty(aluno.Email))
        break;


var mensagem = JsonSerializer.Serialize(aluno);
var corpo = Encoding.UTF8.GetBytes(mensagem);


await channel.BasicPublishAsync(exchange: string.Empty,
                                routingKey: "hello",
                                body: corpo);


Console.WriteLine($"[x] Enviado '{mensagem}'");
```

```
        }
    }
}


public class Aluno
{
    public string Nome { get; set; }
    public string Email { get; set; }
}
```

RabbitMQ™ RabbitMQ 4.0.3 Erlang 26.2.5.5

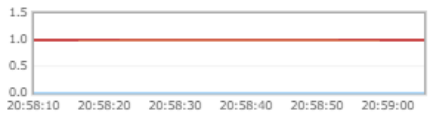
Refreshed 2024-11-15 20:59:07 Refresh every 5 seconds
Virtual host All
Cluster **rabbit@b0cef06e667d**
User **guest** Log out

Overview | Connections | Channels | Exchanges | Queues and Streams | Admin

Queue hello

Overview

Queued messages last minute ?



Ready	1
Unacked	0
Total	1

Message rates last minute ?

Publish	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Get (empty)	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s		

Details

					Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Features	arguments: x-queue-type: classic	State	idle	Messages ?	1	1	0	1	0	0
	queue storage version: 2	Consumers	0	Message body bytes ?	44 B	44 B	0 B	44 B	0 B	0 B
Policy		Consumer capacity ?	0%	Process memory ?	5.2 KiB					
Operator policy										
Effective policy definition										

▸ **Consumers (0)**

▸ **Bindings (1)**

▸ **Publish message**

▾ **Get messages**

Warning: getting messages from a queue is a destructive action. ?

Ack Mode: Nack message requeue true
Encoding: Auto string / base64 ?
Messages: 1

Get Message(s)

Message 1

The server reported **0** messages remaining.

Exchange	(AMQP default)
Routing Key	hello
Redelivered	=
Properties	
Payload 44 bytes Encoding: string	{"Nome":"Thiago","Email":"thiago@gmail.com"}

▸ **Move messages**

▸ **Delete**

4 CONSUMER

O consumer é a parte que se conecta ao RabbitMQ e recebe mensagens de uma fila para processá-las.

1. **Criar Conexão:** Assim como no produtor, iniciamos estabelecendo uma conexão com o RabbitMQ usando ConnectionFactory e CreateConnectionAsync(). A conexão garante o link com o servidor RabbitMQ.

2. **Criar Canal:** O próximo passo é criar um canal utilizando CreateChannelAsync(). Esse canal permite que o consumidor interaja com o RabbitMQ, declarando filas e consumindo mensagens.

3. **Declarar a Fila:** Declaramos a fila “hello” para garantir que ela exista antes de consumir as mensagens. Esta etapa previne erros quando o consumidor tenta se conectar a uma fila não existente.

4. **Criar Consumidor:** Criamos um consumer assíncrono utilizando AsyncEventingBasicConsumer. Ao receber uma mensagem, o evento ReceivedAsync é disparado, onde a mensagem é desserializada para um objeto Aluno, e os dados são exibidos no console.

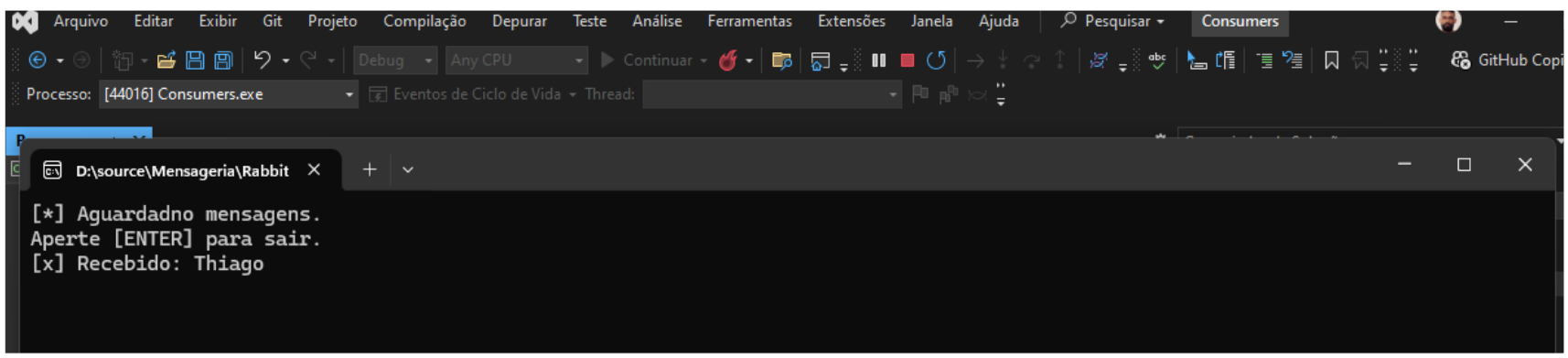
5. **Consumir Mensagem:** Por fim, utilizamos BasicConsumeAsync() para consumir mensagens da fila “hello”. Configuramos autoAck: true para confirmar automaticamente o recebimento das mensagens.

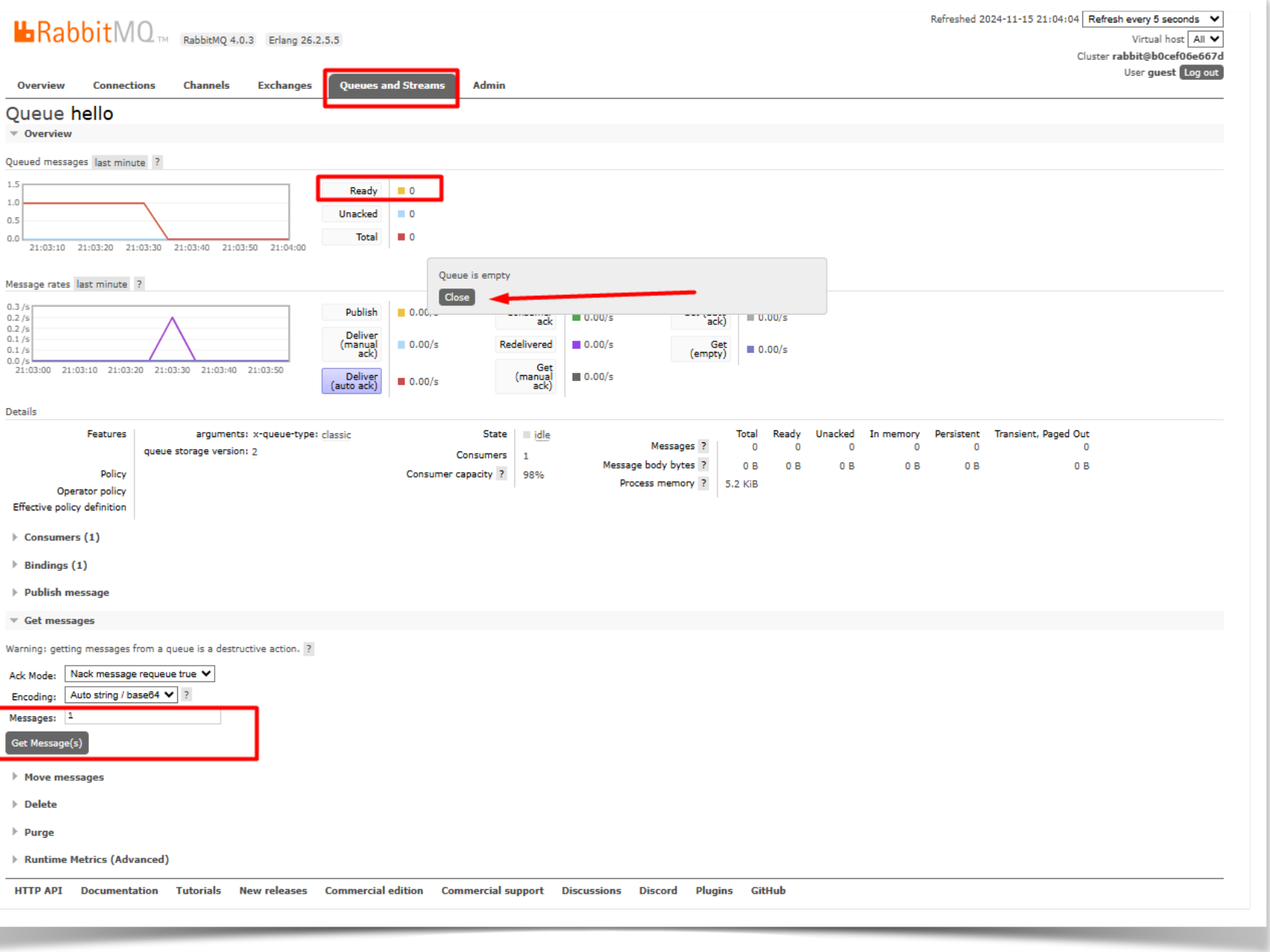
4.1 IMPLEMENTANDO UM CONSUMER

```
using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

using System.Text.Json;



namespace Consumers

{

    public class Program

    {

static async Task Main(string[] args)

{

    var factory = new ConnectionFactory { HostName = "localhost" };

    using var connection = await factory.CreateConnectionAsync();

    using var channel = await connection.CreateChannelAsync();



    await channel.QueueDeclareAsync(queue: "hello",

                                    durable: false,

                                    exclusive: false,

                                    autoDelete: false);
```

```
Console.WriteLine("[*] Aguardando mensagens.");


var consumer = new AsyncEventingBasicConsumer(channel);
consumer.ReceivedAsync += (model, ea) =>
{
    var corpo = ea.Body.ToArray();
    var mensagem = Encoding.UTF8.GetString(corpo);
    var aluno = JsonSerializer.Deserialize<Aluno>(mensagem);
    Console.WriteLine($"[x] Recebido: {aluno.Nome}");


    return Task.CompletedTask;
};


    await channel.BasicConsumeAsync(queue: "hello",
                                    autoAck: true,
                                    consumer: consumer);


    Console.WriteLine("Aperte [ENTER] para sair.");
    Console.ReadLine();
  }
}


public class Aluno
```

```
    {

        public string Nome { get; set; }

        public string Email { get; set; }

    }

}
```





5 CENÁRIOS DO MUNDO REAL COM RABBITMQ

RabbitMQ é amplamente utilizado em diversos cenários do mundo real que envolvem comunicação assíncrona e escalabilidade. Abaixo estão alguns exemplos práticos de como RabbitMQ pode ser aplicado:

Processamento de Pedidos de E-commerce

Em sistemas de e-commerce, RabbitMQ é usado para processar pedidos de forma assíncrona. Quando um cliente realiza um pedido, o produtor envia uma mensagem para uma fila contendo os detalhes do pedido. Consumers diferentes podem então processar o pagamento, atualizar o inventário e enviar notificações ao cliente, permitindo que cada tarefa seja tratada independentemente e em paralelo.

Sistema de Notificações em Tempo Real

Aplicações que precisam enviar notificações em tempo real, como mensagens de confirmação ou alertas, utilizam RabbitMQ para garantir que as notificações sejam entregues e processadas de maneira confiável. O produtor envia a mensagem de notificação para uma fila, enquanto consumers a processam e encaminham ao usuário final, seja via SMS, e-mail ou push notification.

Integração Entre Microserviços

Em arquiteturas de microserviços, RabbitMQ é usado para coordenar a comunicação entre serviços distintos. Por exemplo, um microserviço de cadastro de usuários pode enviar uma mensagem para uma fila informando que um novo usuário foi criado. Outros serviços, como o de envio de e-mails de boas-vindas ou o de criação de perfis de usuário, podem consumir essa mensagem e realizar suas respectivas ações de forma desacoplada.

Processamento de Dados em Lote

RabbitMQ pode ser utilizado para orquestrar o processamento de dados em lote. Um sistema que coleta grandes volumes de dados pode enviar esses dados para uma fila e ter consumidores que os processam em partes menores. Isso ajuda a distribuir a carga e evitar sobrecarga em um único ponto do sistema.

Log de Eventos e Auditoria

Sistemas que precisam registrar eventos para auditoria podem se beneficiar do RabbitMQ para coletar e armazenar logs. Cada serviço do sistema pode enviar logs para uma fila central, e consumidores podem gravar esses logs em um banco de dados de auditoria, garantindo que todos os eventos sejam registrados de maneira centralizada e eficiente.

6 CONCLUSÃO

Implementar um sistema de mensageria usando RabbitMQ é uma solução excelente para promover o desacoplamento e a escalabilidade dos sistemas. A criação de um produtor e de um consumidor com C# demonstra como o RabbitMQ pode ser utilizado de forma simples e eficiente para transmitir mensagens entre diferentes partes de uma aplicação. Esses conceitos podem ser expandidos para arquiteturas mais robustas, utilizando recursos como exchanges mais sofisticadas, filas duráveis e tratamento de mensagens com alta confiabilidade.

Continuar praticando e explorando funcionalidades mais avançadas do RabbitMQ ajudará no desenvolvimento de soluções mais escaláveis e resilientes para aplicações distribuídas.

7 GLOSSÁRIO

- Broker de Mensagens: Um software intermediário que facilita o envio e recebimento de mensagens entre sistemas diferentes, garantindo que as mensagens cheguem ao destino.
- RabbitMQ: Um broker de mensagens amplamente utilizado para implementar a comunicação assíncrona em sistemas distribuídos.
- Docker: Uma plataforma que permite criar, executar e gerenciar containers, que são ambientes isolados contendo todas as dependências necessárias para rodar uma aplicação.
- Container: Um ambiente isolado que contém todas as dependências necessárias para executar uma aplicação de forma consistente.
- Producers: Uma aplicação ou componente que gera e envia mensagens para uma fila.
- Consumers: Uma aplicação ou componente que recebe mensagens de uma fila para processá-las.
- Fila: Estrutura de armazenamento no RabbitMQ onde as mensagens são mantidas até serem consumidas.
- Exchange: Componente do RabbitMQ responsável por receber mensagens de produtores e direcioná-las para as filas apropriadas.
- Routing Key: Uma chave que define como as mensagens devem ser roteadas entre exchanges e filas no RabbitMQ.
- Assíncrono: Uma forma de comunicação em que as partes envolvidas não precisam esperar uma resposta imediata, permitindo que as tarefas sejam executadas em paralelo.
- Mensagem: O pacote de dados enviado de um produtor para uma fila no RabbitMQ.
- Persistência (Durable): Configuração que define se as filas e mensagens devem ser armazenadas de forma persistente, garantindo que não sejam perdidas em caso de falha no servidor.
- ACK (Acknowledgment): Confirmação de recebimento e processamento de uma mensagem pelo consumidor.
- ConnectionFactory: Classe utilizada para criar uma conexão com o RabbitMQ.

REFERÊNCIAS BIBLIOGRÁFICAS

RabbitMQ: Documentação oficial - https://www.rabbitmq.com/documentation.html

Docker: Documentação oficial - https://docs.docker.com/

.NET: Documentação oficial da Microsoft - https://learn.microsoft.com/dotnet/