

Scaling your app with RabbitMQ

Why use RabbitMQ

RabbitMQ is a message broker. A message broker is software that allows applications to communicate with each other, regardless of the platform they were built on.

A message broker acts as a middleman for various services. They can be used to reduce loads and delivery times for your applications by delegating tasks that would normally take up a lot of time or resources to a third party process. You can remove the heavy work from our applications such as generating reports, sending an email, http requests or SMS.

RabbitMQ enables software applications to scale. Applications can connect to each other, as components of a larger application. Messaging is asynchronous, decoupling applications by separating sending and receiving data.

So it can be used to:

- Decoupling software components
- Performing background operations
- Performing asynchronous operation

In this example we will show how to decouple an application, providing performance and scalability.

At the beginning...

We have an image viewing API, which at first looked like this:

```
[HttpGet]
[Route("{id}")]
0 references
public async Task<Image> GetAsync(string id)
{
    return await _imageService.GetImage(id);
}
```

Everything is OK. Its just returns an image. Pretty simple, right?

New features

Over time, we implemented some new features, based on rankings and recommendations:

```
1 reference
public Image GetImage(string id)
{
    UpdateMostViewed(id);

    UpdateRecommendations(id);

    return new Image
    {
        Name = "image-01",
        Content = "<image-content>"
    };
}
```

Does this feel right?

Now the API call is doing things that don't really matter to the customer, at least not in *real time*. All of these actions can be done *asynchronously*, freeing the api call to do almost exclusively what the client is asking for: return an image.

By removing these "reporting" logics, we make the API more performant for the client and remove some points of failure in the API.

RabbitMQ

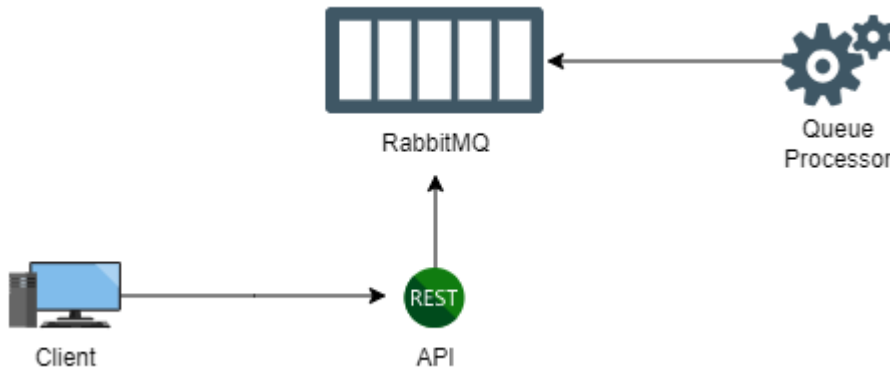
So what will we do?

When an image is requested, we will trigger an *event* called *ImageViewEvent*. this event will be sent to RabbitMQ, and another service will *consume* this event, processing all the rules we removed from the API stream.

```
1 reference
public async Task<Image> GetImage(string id)
{
    await _busControl.Publish(new ImageViewEvent
    {
        ImageID = id,
    });

    return new Image
    {
        Name = "image-01",
        Content = "<image-content>"
    };
}
```

Our event will only contain the image ID, which is what our new methods will need.



Configuring the solution so far...

Let's use the *MassTransit* client to communicate with rabbitmq, as it already comes with some really cool features that make life a lot easier:

Required Nugets packages:

```
<PackageReference Include="MassTransit" Version="8.3.1" />
<PackageReference Include="MassTransit.AspNetCore" Version="7.3.1" />
<PackageReference Include="MassTransit.RabbitMQ" Version="8.3.1" />
```

in *startup.cs* we configure it as follows:

```
services.AddMassTransit(x =>
{
    x.UsingRabbitMq((context, cfg) => {
        cfg.Host(new Uri($"rabbitmq://queue"), host =>
        {
            host.Username("guest");
            host.Password("guest");
        });

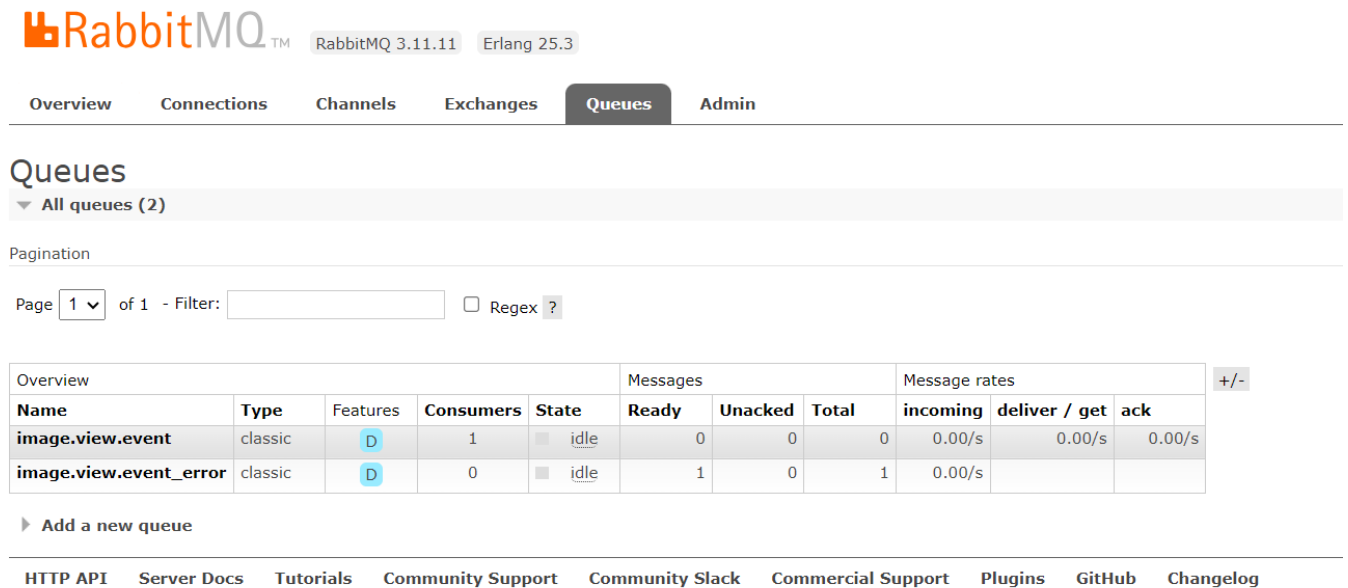
        cfg.ConfigureEndpoints(context);
    });
});
```

With this, the *IBusControl* class is already injected into the net core dependency injection container. This is the class used to send events to RabbitMQ, as shown in the example above.

Our service that will consume the messages will be a *Console Application*, with the following packages:

```
<PackageReference Include="MassTransit" Version="8.3.1" />
<PackageReference Include="MassTransit.RabbitMQ" Version="8.3.1" />
```

and with the following settings:



RabbitMQ 3.11.11 Erlang 25.3

Overview Connections Channels Exchanges **Queues** Admin

Queues

▼ All queues (2)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

| Overview | | | | | Messages | | | Message rates | | | +/- |
|------------------------|---------|----------|-----------|-------|----------|---------|-------|---------------|---------------|--------|-----|
| Name | Type | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack | |
| image.view.event | classic | D | 1 | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | 0.00/s | |
| image.view.event_error | classic | D | 0 | idle | 1 | 0 | 1 | 0.00/s | | | |

► Add a new queue

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

The configuration is similar to the API, with the addition of *ReceiveEndpoint*, which configures the *ImageViewProcessor* class, which has all the logic extracted from the API:

```
1 reference
public class ImageViewProcessor : IConsumer<ImageViewEvent>
{
    0 references
    public Task Consume(ConsumeContext<ImageViewEvent> context)
    {
        UpdateMostViewed(context.Message.ImageID);

        UpdateRecommendations(context.Message.ImageID);

        return Task.CompletedTask;
    }

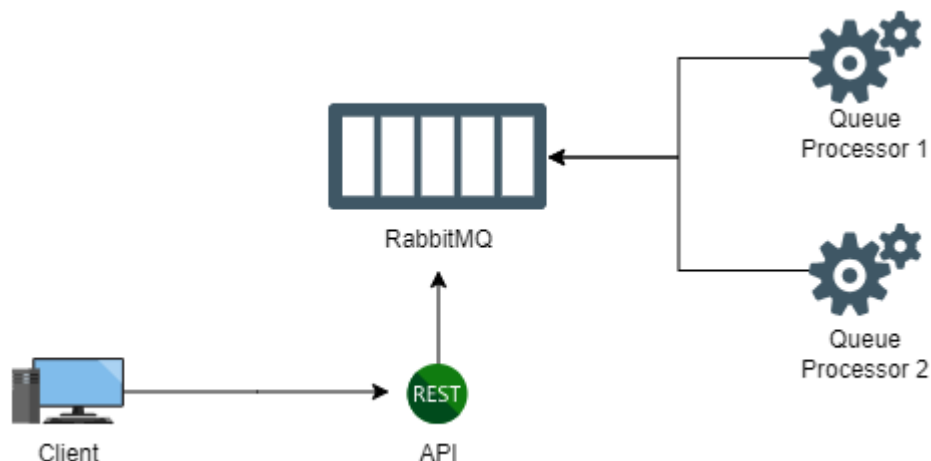
    1 reference
    private void UpdateMostViewed(string id)
    {
        Console.WriteLine("UpdateMostViewed: " + id);
        return;
    }

    1 reference
    private void UpdateRecommendations(string id)
    {
        Console.WriteLine("UpdateMostViewed: " + id);
        return;
    }
}
```

Our class that will process the event must implement the *IConsumer* interface, passing the object we want to process.

Horizontal scalability

We were also able to horizontally scale rule processing. You can just run other instance of the same version of our service. The two services will be listening to the same queue, and rabbitmq itself will distribute the messages to consumers appropriately (*round-robin* for example), achieving parallel processing.



RabbitMQ guarantees that the same message does not go to the same consumer.

Explaining RabbitMQ better

Well, now let's explain a little better how rabbitmq and masstransit work.

Let's quickly run *RabbitMQ* with Docker:

```
docker run -p 5672:5672 -p 15672:15672 rabbitmq:3.8.9-management
```

The port 5672 is used for sending and receiving messages, which is already standard in our client, while 15672 is the UI port.

RabbitMQ 3.11.11 Erlang 25.3
Refreshed 2023-09-18 20:25:01

[Overview](#)
[Connections](#)
[Channels](#)
[Exchanges](#)
[Queues](#)
[Admin](#)

Overview

Totals

Nodes

| Name | File descriptors ? | Socket descriptors ? | Erlang processes | Memory ? | Disk space | Uptime | Info | Reset stats |
|---------------------|-------------------------|-----------------------|--------------------------|-----------------------------------|---------------------------------|--------|------------------|---------------------|
| rabbit@fcb1dd6f35b8 | 56 1048576 available | 2 943629 available | 427 1048576 available | 148 MiB 6.2 GiB high watermark | 230 GiB 48 MiB low watermark | 1m 3s | basic disc 2 rss | This node All nodes |

Churn statistics

Ports and contexts

Export definitions

Import definitions

[HTTP API](#)
[Server Docs](#)
[Tutorials](#)
[Community Support](#)
[Community Slack](#)
[Commercial Support](#)
[Plugins](#)
[GitHub](#)
[Changelog](#)

We have two main entities, the *Exchange* and the *Queue*. When publishing a message, it is published on an *Exchange*, and it is sent to the *Queue*.

Using *MassTransit*, an *Exchange* equal to the *namespace* of the sending object is already created at the time the message is published. When a consumer is connected, the queue configured in the *ReceiveEndpoint* (*image.view* in the example) is created and the *bind* with the exchange is also done.

The "bind" happens through the message object, which has to be the same, including the namespace, in the producer and consumer, so the consumer knows which message has to be processed.

Do we have the *Queue*:

RabbitMQ™ RabbitMQ 3.11.11 Erlang 25.3

Overview Connections Channels Exchanges **Queues** Admin

Queues

▼ All queues (1)

Pagination

Page of 1 - Filter: ☐ Regex ?

| Overview | | | | | Messages | | | Message rates | | |
|------------------|---------|----------|-----------|-------|----------|---------|-------|---------------|---------------|-----|
| Name | Type | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| image.view.event | classic | D | 1 | idle | 0 | 0 | 0 | | | |

► Add a new queue

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

The *Exchange*:

Exchanges

▼ All exchanges (9)

Pagination

Page

1 ▼

 of 1 - Filter:

☐ Regex

?

| Name | Type | Features | Message rate in | Message rate out | +/- |
|-----------------------|---------|----------------|-----------------|------------------|-----|
| (AMQP default) | direct | <div>D</div> | | | |
| Events:ImageViewEvent | fanout | <div>D</div> | 0.00/s | | |
| amq.direct | direct | <div>D</div> | | | |
| amq.fanout | fanout | <div>D</div> | | | |
| amq.headers | headers | <div>D</div> | | | |
| amq.match | headers | <div>D</div> | | | |
| amq.rabbitmq.trace | topic | <div>D I</div> | | | |
| amq.topic | topic | <div>D</div> | | | |
| image.view.event | fanout | <div>D</div> | | | |

► Add a new exchange

And the *Binding*:



RabbitMQ 3.11.11

Erlang 25.3

[Overview](#)[Connections](#)[Channels](#)[Exchanges](#)[Queues](#)[Admin](#)

Exchange: image.view.event

▼ Overview

Message rates **last minute** ?

Currently idle

Details

| | |
|----------|---------------|
| Type | fanout |
| Features | durable: true |
| Policy | |

▼ Bindings

| From | Routing key | Arguments | |
|-----------------------|-------------|-----------|--------|
| Events:ImageViewEvent | | | Unbind |



This exchange



| To | Routing key | Arguments | |
|------------------|-------------|-----------|--------|
| image.view.event | | | Unbind |

It's starting to look good! We started with an *event-driven architecture* and *asynchronous* and parallel processing. We also don't need to worry too much about service downtime, as the messages will continue in the queue, and when the service goes up, it will consume the messages.

And the errors?

The *masstransit* provides us with features that make error handling easier. Let's look at *Retries* and *CircuitBreaker*.

When an *exception* happens in our consumer, the message goes to an `_error` queue, for example:

Queues

▼ All queues (2)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

| Overview | | | | | Messages | | | Message rates | | | +/- |
|------------------------|---------|----------------|-----------|-------------------|----------|---------|-------|---------------|---------------|--------|-----|
| Name | Type | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack | |
| image.view.event | classic | D | 1 | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | 0.00/s | |
| image.view.event_error | classic | D | 0 | idle | 1 | 0 | 1 | 0.00/s | | | |

► Add a new queue

With *retries* we can configure retries. For example: 5 attempts at 1 second intervals. Only after these attempts does the message go to the error queue.

```
config.ReceiveEndpoint("image.view.event", e =>
{
    e.UseMessageRetry(r => r.Interval(5, TimeSpan.FromSeconds(1)));
    e.Consumer<ImageViewProcessor>();
});
```

Another interesting feature that we can configure is the *Circuit Breaker*. This pattern is very important for microservices. The number of errors in the service is monitored, and if the quantity exceeds the expected value in a certain period of time, the *circuit opens*, preventing future requests, which may possibly fail, from overloading the service. After a set time, and if there are no more errors occurring, the *circuit closes*, returning to normal flow.

```
config.UseCircuitBreaker(cb =>
{
    cb.TrackingPeriod = TimeSpan.FromMinutes(1);
    cb.TripThreshold = 15;
    cb.ActiveThreshold = 10;
    cb.ResetInterval = TimeSpan.FromMinutes(5);
});
```

Quite a lot, right?

Want to see everything running? download the branch *master* from <https://github.com/hebermattos/scaling-app-rabbitmq> and run:

```
docker-compose up
```

The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane displays the project structure for 'SCALING-APP-RABBITMQ'. The file 'ImageViewProcessor.cs' is selected. The main editor shows the code for 'ImageViewProcessor.cs', which implements the 'IConsumer<ImageViewEvent>' interface. The code includes using statements for 'System', 'System.Threading.Tasks', 'Events', and 'MassTransit'. It defines a 'namespace service' and a 'public class ImageViewProcessor'. The 'Consume' method is implemented to call 'UpdateMostViewed' and 'UpdateRecommendations'. Both methods are private and write to the console. The bottom pane shows the 'TERMINAL' output with logs from RabbitMQ and the application, including connection messages and the start of the MassTransit bus.

```

1  using System;
2  using System.Threading.Tasks;
3  using Events;
4  using MassTransit;
5
6  namespace service
7  {
8      1 reference
9      public class ImageViewProcessor : IConsumer<ImageViewEvent>
10     {
11         0 references
12         public Task Consume(ConsumeContext<ImageViewEvent> context)
13         {
14             UpdateMostViewed(context.Message.ImageID);
15             UpdateRecommendations(context.Message.ImageID);
16             return Task.CompletedTask;
17         }
18
19         1 reference
20         private void UpdateMostViewed(string id)
21         {
22             Console.WriteLine("UpdateMostViewed: " + id);
23             return;
24         }
25
26         1 reference
27         private void UpdateRecommendations(string id)
28         {
29             Console.WriteLine("UpdateRecommendations: " + id);
30             return;
31         }
32     }

```

TERMINAL

```

scaling-app-rabbitmq-queue-1 | 2023-09-18 23:42:31.209097+00:00 [info] <0.870.0> connection <0.870.0> (172.20.0.3:356
scaling-app-rabbitmq-queue-1 | 2023-09-18 23:42:31.215841+00:00 [info] <0.870.0> connection <0.870.0> (172.20.0.3:356
scaling-app-rabbitmq-queue-1 | 2023-09-18 23:42:32.362921+00:00 [info] <0.897.0> accepting AMQP connection <0.897.0>
scaling-app-rabbitmq-queue-1 | 2023-09-18 23:42:32.388930+00:00 [info] <0.897.0> connection <0.897.0> (172.20.0.4:358
scaling-app-rabbitmq-queue-1 | 2023-09-18 23:42:32.392854+00:00 [info] <0.897.0> connection <0.897.0> (172.20.0.4:358
scaling-app-rabbitmq-api-1 | info: MassTransit[0]
scaling-app-rabbitmq-queue-processor-1 | Bus started: rabbitmq://queue/
scaling-app-rabbitmq-queue-processor-1 | UpdateMostViewed: 1
scaling-app-rabbitmq-queue-processor-1 | UpdateRecommendations: 1

```

Requests can be made at the url <http://localhost:5000/images/123> for example, and the rabbit UI can be viewed at <http://localhost:15672>. The messages being "processed" will appear in the console.

Well, that's it for now. I think it's a good start to start dealing with RabbitMq. There are a lot of concepts here that you can explore later:

- Microservices
- Horizontal Scalability
- Event-Driven Architecture
- Sagas, Orchestration vs Choreography
- RabbitMQ features
- And the list go on....