Using MassTransit with RabbitMQ in ASP.NET Core

Posted by **Code Maze** | Updated Date Jul 11, 2022 | **14**

Want to build **great APIs?** Or become **even better** at it? Check our **Ultimate ASP.NET Core Web API program** and learn how to create a full production-ready ASP.NET Core API using only the latest .NET technologies. Bonus materials (Security book, Docker book, and other bonus files) are included in the Premium package!

In this article, we are going to take a look at how we can use the open-source, distributed application library MassTransit in conjunction with RabbitMQ in an ASP.NET Core application. First, we are going to cover some of the more advanced RabbitMQ features, as well as some of the concepts we will come across in the MassTransit library. Finally, we will learn how to use these libraries in an ASP.NET Core Web API application.

This article makes use of Docker to run our RabbitMQ server locally.

> To download the source code for this article, you can visit our **GitHub repository**.

For this tutorial, we recommend that you have some knowledge of RabbitMQ. We will cover some of the more advanced features of RabbitMQ in this article, but for an introduction, we have a great article on the topic: **How to Use RabbitMQ in ASP.NET Core**.

After finishing this article, we expect you to have a good understanding of what MassTransit is, what benefits it provides, and how we can use it in conjunction with RabbitMQ.

What is RabbitMQ?

To very briefly recap, RabbitMQ is a message broker, which handles the accepting, storing, and sending of messages between our applications. Using a message broker allows us to build decoupling, performant applications, relying on asynchronous communication between our applications.

What Are RabbitMQ Exchanges?

When working with RabbitMQ, producers can send messages to a couple of different endpoints:

- Queues
- Exchanges

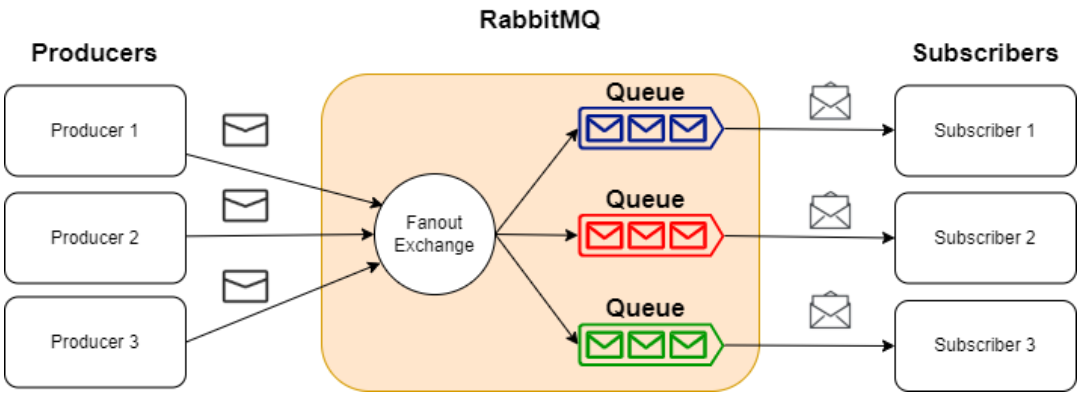
As we have covered queues in the previous article, we are going to focus just on exchanges. When a producer sends directly to a queue, this message will be received by all consumers of that queue. But what if we want to selectively send messages to different queues based on metadata found in the message? This is where exchanges come into play.

An exchange receives messages from producers, and depending on its configuration, will send the message to one or many queues. We must create a **binding**, which will ensure our messages get sent from our exchange to one or many queues.

We can define exchanges from one of the following types:

- Direct
- Topic
- Headers
- Fanout

For this article, we will focus on the **Fanout** type, as that is what MassTransit uses by default. The fanout exchange type is very simple. It will just broadcast all the messages it receives to all the queues that have created a binding with it.



What is MassTransit?

MassTransit is a free, open-source, distributed application framework for .NET applications. **It abstracts away the underlying logic required to work with message brokers, such as RabbitMQ, making it easier to create message-based, loosely coupled applications**.

There are a few fundamental concepts we should cover first:

Service Bus, usually shortened to **Bus**, is the term given to the type of application that handles the movement of messages.

Transports are the different types of message brokers MassTransit works with, including RabbitMQ, InMemory, Azure Service Bus, and more.

> Wanna join Code Maze Team, help us produce more awesome .NET/C# content and **get paid? >> JOIN US! <<**

Message is a contract, defined *code first* by creating a .NET class or interface.

Command is a type of message, specifically used to tell a service to do something. These message types are **sent** to an endpoint (queue) and will be expressed using a verb-noun sequence.

Events are another message type, signifying that something has happened. Events are **published** to one or multiple consumers and will be expressed using noun-verb (past tense) sequence.

Why Use MassTransit?

There are a few benefits to choosing to use a library such as MassTransit, instead of working with the native message broker library. Firstly, **by abstracting the underlying message broker logic, we can work with**

multiple message brokers, without having to completely rewrite our code. This allows us to work with something such as the InMemory transport when working locally, then when deploying our code, use another transport such as Azure Service Bus or Amazon Simple Queue Service.

Additionally, when we work with a message-based architecture, **there are a lot of specific patterns we need to be aware of and implement, such as *retry*, *circuit breaker*, *outbox* to name a few. MassTransit handles all of this for us**, along with many other features such as *exception handling*, *distributed transactions*, and *monitoring*.

Now that we have an understanding of what MassTransit is and why we would use it, let's see how we can use it along with RabbitMQ in ASP.NET Core.

Implementing MassTransit With RabbitMQ in ASP.NET Core

Let's start by installing **RabbitMQ**.

Installing RabbitMQ

Before we start creating our application, we will first need to spin up a RabbitMQ server, by making use of Docker. Given that Docker is installed, we'll open a command-line terminal and use the `docker run` command to spin up our server:

```
docker run -d --hostname my-rabbitmq-server --name rabbitmq -p
5672:5672 -p 15672:15672 rabbitmq:3-management
```

We are using the `rabbitmq:3-management` image from **DockerHub** which will provide us with a UI, available on port **15672**. We must also add a port mapping for **5672**, which is the default port RabbitMQ uses for communication. In order for us to access the management UI, we open a browser window and navigate to `localhost:15672`, using the default login of **guest/guest**. We will come back to this management UI later to see what MassTransit creates for us in RabbitMQ.

Creating a Shared Class Library

If you remember back to the concepts of MassTransit, when we use **Messages**, we must define a .NET class or interface. MassTransit includes the namespace

for message contracts, so we can use a shared class/interface to set up our bindings correctly.

With this in mind, the first thing we are going to do is create a class library that will contain the shared interface that we will use for our **Producer** and **Consumer** applications.

We will first create a class library, which we will call `SharedModels`, within which we will define an interface:

```
public interface OrderCreated
{
    int Id { get; set; }
    string ProductName { get; set; }
    decimal Price { get; set; }
    int Quantity { get; set; }
}
```

We are using the past tense for the interface name, which indicates that this is an **event** message type. This is everything we need for our `SharedModels` class library. Next up, we will implement our Producer.

Creating a Producer Using MassTransit

Let's create our Producer, which we will implement as an ASP.NET Core Web API. The first thing we want to do is add a project reference to our `SharedModels` class library.

Next up, we need to add a couple of Nuget packages for MassTransit:

- `MassTransit`
- `MassTransit.AspNetCore`
- `MassTransit.RabbitMQ`

Now let's configure MassTransit to use RabbitMQ in `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddMassTransit(x =>
{
    x.UsingRabbitMq();
});

builder.Services.AddMassTransitHostedService();
```

NOTE: For the newest versions of the MassTransit package, you can ignore the `builder.Services.AddMassTransitHostedService()` line.

First, we create our `WebApplicationBuilder`.

Next up we configure MassTransit to use RabbitMQ. As this is our Producer, and RabbitMQ is going to be running on `localhost`, we don't need to define any more configuration here. The final step is to call `AddMassTransitHostedService`, which automatically handles the starting/stopping of the bus (you don't have to do this for the newest library version).

Before we create an API controller, we must first create an `OrderDto` class, which will be used as a parameter for our endpoint method:

```
public class OrderDto
{
    public string ProductName { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
}
```

With MassTransit configured to use **RabbitMQ** and our DTO defined, let's now create an API controller that will publish a message, or more specifically an **event**, using our `OrderCreated` interface:

```
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase
{
    private readonly IPublishEndpoint _publishEndpoint;

    public OrdersController(IPublishEndpoint publishEndpoint)
```

```
        {
            _publishEndpoint = publishEndpoint;
        }
    }
```

The first thing we must do is inject an `IPublishEndpoint` into our controller, which is what we'll use to publish our event.

Now we can create an endpoint for creating an order:

```
[HttpPost]
public async Task<IActionResult> CreateOrder(OrderDto orderDto)
{
    await _publishEndpoint.Publish<OrderCreated>(new
    {
        Id = 1,
        orderDto.ProductName,
        orderDto.Quantity,
        orderDto.Price
    });

    return Ok();
```

```
}
```

The first thing we must do is create a new method called `CreateOrder`, and decorate it with the `HttpPost` attribute. This method will take our `OrderDto` as a parameter. Within the method, we call the generic `Publish` method on the `IPublishEndpoint` interface, using our `OrderCreated` interface to define what type of event we are going to be publishing.

We can then create an anonymous type, ensuring we use the same property names that are defined in our `OrderCreated` interface. This is all we require to publish an event to our configured transport, RabbitMQ.

Finally, we will return an `OkResult`.

Creating a Consumer Using MassTransit

Now that we have our Producer in place, we'll take a look at creating a very simple Consumer, using a .NET console application.

Within the same solution, let's create a console application called `Consumer`. Like our Producer application, we must add a project reference to our `SharedModels` class library. We must also add a couple of Nuget packages for MassTransit:

- `MassTransit`
- `MassTransit.RabbitMQ`

This time, we don't need the `MassTransit.AspNetCore` package, as we won't be making use of dependency injection in this project.

With the references correctly added, we first need to create a Consumer implementation, that will contain the logic for what we want to do with any message received:

```
class OrderCreatedConsumer : IConsumer<OrderCreated>
{
    public async Task Consume(ConsumeContext<OrderCreated> context
    {
        var jsonMessage = JsonConvert.SerializeObject(context.Mess
        Console.WriteLine($"OrderCreated message: {jsonMessage}");
    }
}
```

First, we create a class called `OrderCreatedConsumer`, ensuring we implement the `IConsumer` generic interface provided by MassTransit, using our `OrderCreated` interface defined in our ShareModels library. In the `Consume` method, we can simply serialize the message object and log the message to the console for the purposes of this article.

> Wanna join Code Maze Team, help us produce more awesome .NET/C# content and **get paid?** **>> JOIN US! <<**

Now that our Consumer class is defined, let's configure our Consumer to use MassTransit in `Program.cs`:

```
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
    cfg.ReceiveEndpoint("order-created-event", e =>
    {
        e.Consumer<OrderCreatedConsumer>();
    });

});
```

We create an `IBusControl`, using the static `Bus` class provided by MassTransit, which we're going to configure to use RabbitMQ. We must then configure the `ReceiveEndpoint`, which will receive messages from the `order-created-event` queue. Finally, we use our previously created `OrderCreatedConsumer` to consume messages from this queue.

With our Consumer configured to receive messages, the final thing we need to do is start our bus:

```
await busControl.StartAsync(new CancellationToken());

try
{
    Console.WriteLine("Press enter to exit");

    await Task.Run(() => Console.ReadLine());
}
finally
```

```
{
    await busControl.StopAsync();
}
```

First, we call `StartAsync` on our `busControl`, passing in a new `CancellationToken`. Now we set up the console app to run without exiting until a `Console.ReadLine` is registered. The final step is to ensure we stop our `busControl` by calling `StopAsync`.

Testing Our Application

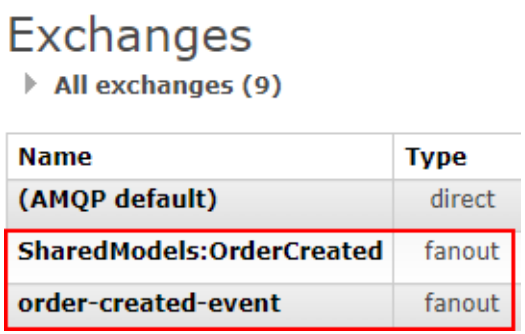
Now it's time to test our code. Ensuring we have both the Producer and Consumer applications set up run on startup, hitting **F5** will open a web browser (Producer) and a console window (Consumer). Next, we can send a POST request to `https://localhost:7188/api/orders`, providing an order in the request body:

```
{
    "productName": "keyboard",
    "quantity": 1,
    "price": 99.99
}
```

Let's look at our console window, and we will see that our Producer correctly sent a message to RabbitMQ which our Consumer has successfully received:

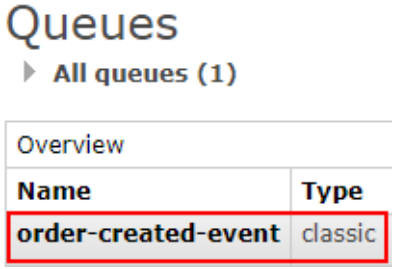
```
OrderCreated message: {"Id": 1, "ProductName": "keyboard",
"Price": 99.99, "Quantity": 1}
```

Furthermore, we can navigate to our RabbitMQ management UI, to see what exchanges MassTransit has created for us:



We should be able to see 2 exchanges created, both with the fanout type. One for the `SharedModels:OrderCreated` event type, which uses the namespace and model name from our shared library. We also have one for the receive endpoint we defined in our Consumer, `order-created-event`, which has a binding to the `SharedModels:OrderCreated` exchange.

And we can also check what queues were created:



We can see our queue, `order-created-event` created, which has a binding to the exchange of the same name.

Conclusion

In this article, we've learned about a more advanced feature of RabbitMQ, exchanges. We also had a look at the MassTransit library, and why we would choose to use it over one of the native message broker libraries. Finally, we brought this altogether by learning how to create a Producer and Consumer using MassTransit and RabbitMQ.

SHARE:

14 COMMENTS Oldest

View Comments