Messaging through a service bus in .NET using MassTransit part 3: publishing messages to multiple consumers

SEPTEMBER 20, 2016 3 COMMENTS (HTTPS://DOTNETCODR.COM/2016/09/20/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-3-PUBLISHING-MESSAGES-TO-MULTIPLE-CONSUMERS/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/14/messaging-through-a-service-bus-in-net-using-masstransit-part-2-starting-with-some-code/) we got our hands dirty and started coding a small demo application around MassTransit. We managed to send a message from a publisher to a consumer using the MassTransit/RabbitMq client library. We saw a very basic configuration of the bus control and how to register a consumer for a message type. The message type can by convention be an event or a command. Both are best encapsulated in an interface with get-set properties and separate naming conventions. Therefore commands and events are not some special C# language features in this case. Instead, they are basic terminology in the world of messaging. Our first example centred around sending a single command using a single queue.

In this post we'll extend our demo to publishing a message that can be consumed by multiple receivers. The goal is to publish a customer registered event from the register customer command consumer. The event will be consumed by 2 receivers that the event publisher will not have any knowledge of.

The message contract

We know from the previous post and the MassTransit documentation page we referred to that an event name consists of a noun, which is the resource or the domain, followed by a verb in past tense that describes what happened to the resource. Our demo project has a MyCompany.Messaging C# library where we already have a command-style interface called IRegisterCustomer. Go ahead and insert the following event interface into the library:

```
using System;

namespace MyCompany.Messaging
{
    public interface ICustomerRegistered
    {
        Guid Id { get; }
        DateTime RegisteredUtc { get; }
        string Name { get; }
        string Address { get; }
    }
}
```

It has some of the same properties as IRegisterCustomer. When a new customer is registered then we won't publish all the details to the consuming parties.

Updates in the command receiver

We also have a project called MassTransit.Receiver in our demo project. It currently listens to commands on the queue called mycompany.domains.queues. We've also registered an IConsumer called RegisterCustomerConsumer when building the bus control. We'll first make a slight change in the Main method of Program.cs:

```
static void Main(string[] args)
{
    Console.Title = "This is the customer registration command receiver.";
    Console.WriteLine("CUSTOMER REGISTRATION COMMAND RECEIVER.");
    RunMassTransitReceiverWithRabbit();
}
```

We'll have 4 command windows up and running when starting the demo at the end of this post so it's good to have an easy means of identifying which window belongs to what.

Our goal is to publish an event as soon as the command has been taken care of. One way of achieving it is publishing the event from the RegisterCustomerConsumer object. The IBusControl interface has a Publish method that could do the job. Here's an example of publishing an IRegisterCustomer:

```
rabbitBusControl.Publish<IRegisterCustomer>(new
{
    Address = "New Street",
        Id = Guid.NewGuid(),
    Preferred = true,
    RegisteredUtc = DateTime.UtcNow,
    Name = "Nice people LTD",
    Type = 1,
    DefaultDiscount = 0
});
```

However we have no access to the bus in the consumer. We're lucky because the ConsumeContext object of the Consume method has the same function. Here's the updated RegisterCustomerConsumer object:

```
using MyCompany.Messaging;
using System;
using System.Threading.Tasks;

namespace MassTransit.Receiver
{
    public class RegisterCustomerConsumer : IConsumer<IRegisterCustomer>
    {
        public Task Consume(ConsumeContext<IRegisterCustomer> context)
        {
            IRegisterCustomer newCustomer = context.Message;
            Console.WriteLine("A new customer has signed up, it's time to register it in the command
            Console.WriteLine(newCustomer.Address);
            Console.WriteLine(newCustomer.Name);
            Console.WriteLine(newCustomer.Id);
            Console.WriteLine(newCustomer.Preferred);

            context.Publish<ICustomerRegistered>(new
            {
                Address = newCustomer.Address,
                Id = newCustomer.Id,
                RegisteredUtc = newCustomer.RegisteredUtc,
                Name = newCustomer.Name
            });

            return Task.FromResult(context.Message);
        }
    }
}
```

Note that we didn't have to specify a queue name here as opposed to sending a command to a single queue. We'll see that the queue names are only provided in the consumers. MassTransit will create the necessary queues in the background.

Changes in the publisher

We'll add a small change to the MassTransit.Publisher console project as well so that we can easily identify its command window:

```
static void Main(string[] args)
{
    Console.WriteLine("CUSTOMER REGISTRATION COMMAND PUBLISHER");
    Console.Title = "Publisher window";
    RunMassTransitPublisherWithRabbit();
}
```

The Management and Sales consumers

Let's say that Management and Sales want to be notified of customer registered events. Let's see how to sign them up as consumers. Add two new C# console applications to the solution:

- MassTransit.Receiver.Management
- MassTransit.Receiver.Sales

Do the following to both:

- Add the MassTransit.RabbitMq NuGet package
- Add a project reference to the MyCompany.Messaging library

Here's Program.cs of the Management consumer:

```
using MassTransit.RabbitMqTransport;
using System;

namespace MassTransit.Receiver.Management
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = "Management consumer";
            Console.WriteLine("MANAGEMENT");
            RunMassTransitReceiverWithRabbit();
        }

        private static void RunMassTransitReceiverWithRabbit()
        {
            IBusControl rabbitBusControl = Bus.Factory.CreateUsingRabbitMq(rabbit =>
            {
                IRabbitMqHost rabbitMqHost = rabbit.Host(new Uri("rabbitmq://localhost:5672/accountin
                {
                    settings.Password("accountant");
                    settings.Username("accountant");
                });

                rabbit.ReceiveEndpoint(rabbitMqHost, "mycompany.domains.queues.events.mgmt", conf =>
                {
                    conf.Consumer<CustomerRegisteredConsumerMgmt>();
                });
            });
            rabbitBusControl.Start();
            Console.ReadKey();
            rabbitBusControl.Stop();
        }
    }
}
```

…where CustomerRegisteredConsumerMgmt looks as follows:

```
using MyCompany.Messaging;
using System;
using System.Threading.Tasks;

namespace MassTransit.Receiver.Management
{
    public class CustomerRegisteredConsumerMgmt : IConsumer<ICustomerRegistered>
    {
        public Task Consume(ConsumeContext<ICustomerRegistered> context)
        {
            ICustomerRegistered newCustomer = context.Message;
            Console.WriteLine("A new customer has been registered, congratulations from Management to
            Console.WriteLine(newCustomer.Address);
            Console.WriteLine(newCustomer.Name);
            Console.WriteLine(newCustomer.Id);
            return Task.FromResult(context.Message);
        }
    }
}
```

This is nothing new compared to what we saw before. Note the queue name in the ReceiveEndpoint extension method.

The Sales consumer is almost identical. Here's Program.cs:

```
using MassTransit.RabbitMqTransport;
using System;

namespace MassTransit.Receiver.Sales
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = "Sales consumer";
            Console.WriteLine("SALES");
            RunMassTransitReceiverWithRabbit();
        }

        private static void RunMassTransitReceiverWithRabbit()
        {
            IBusControl rabbitBusControl = Bus.Factory.CreateUsingRabbitMq(rabbit =>
            {
                IRabbitMqHost rabbitMqHost = rabbit.Host(new Uri("rabbitmq://localhost:5672/accountin
                {
                    settings.Password("accountant");
                    settings.Username("accountant");
                });

                rabbit.ReceiveEndpoint(rabbitMqHost, "mycompany.domains.queues.events.sales", conf =>
                {
                    conf.Consumer<CustomerRegisteredConsumerSls>();
                });
            });

            rabbitBusControl.Start();
            Console.ReadKey();

            rabbitBusControl.Stop();
        }
    }
}
```

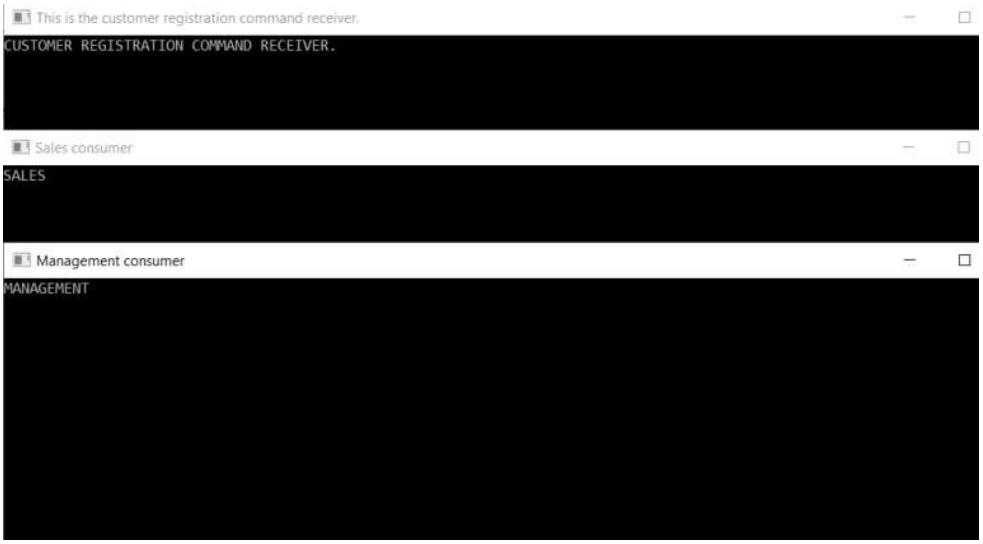
```
using MyCompany.Messaging;
using System;
using System.Threading.Tasks;

namespace MassTransit.Receiver.Sales
{
    public class CustomerRegisteredConsumerSls : IConsumer<ICustomerRegistered>
    {
        public Task Consume(ConsumeContext<ICustomerRegistered> context)
        {
            ICustomerRegistered newCustomer = context.Message;
            Console.WriteLine("Great to see the new customer finally being registered, a big sigh fro
            Console.WriteLine(newCustomer.Address);
            Console.WriteLine(newCustomer.Name);
            Console.WriteLine(newCustomer.Id);
            return Task.FromResult(context.Message);
        }
    }
}
```

Running the demo

Start the receiver projects first:

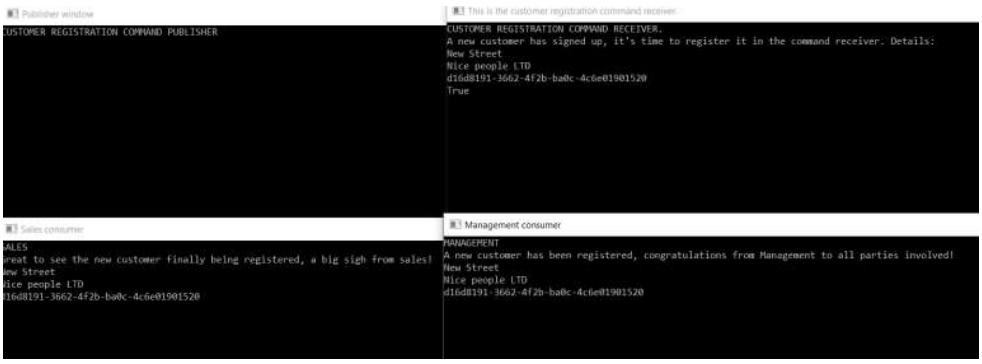
1. MassTransit.Receiver
2. MassTransit.Receiver.Management
3. MassTransit.Receiver.Sales

At this point you'll have 3 command windows on your screen:



(https://dotnetcodr.files.wordpress.com/2016/09/3-receivers-up-and-running-in-masstransit-rabbitmq-demo.png)

Finally start the MassTransit.Publisher project as well. You should see that the various Consume methods are triggered. The RegisterCustomerConsumer is first to receive the initial customer registration command. Then both the Management and Sales consumers receive their customer registered events as well:



(https://dotnetcodr.files.wordpress.com/2016/09/demoing-publishing-a-message-to-multiple-consumers-with-masstransit-rabbitmq.png)

The real beauty with such a system is that MassTransit.Receiver has no knowledge of the actual consumers of the event. It only publishes the event and then it's up to the various other projects to sign up. New consumers of the event can easily sign up or also quit being notified, the event publisher won't care. We didn't have to couple the publishers and consumers in any way. The only coupling is the queue name between MassTransit.Receiver and MassTransit.Publisher in the GetSendEndpoint method, otherwise there's not even a project reference among them. I think this is one of the most important advantages of a distributed system based on messaging and it's good that we nailed it down so early.

We'll continue with dependency injection in the next post.

View the list of posts on Messaging here (https://dotnetcodr.com/messaging/).
FILED UNDER .NET, MESSAGING TAGGED WITH C#, MASSTRANSIT, MESSAGING, RABBITMQ

About Andras Nemes
I'm a .NET/Java developer living and working in Stockholm, Sweden.

3 Responses to *Messaging through a service bus in .NET using MassTransit part 3: publishing messages to multiple consumers*

Bardia says:
April 13, 2017 at 7:30 am
Hi andras, I'm a begginer to using message brokers and have a question.
We have a ticketing service which has multiple sub service. A supervisor service get any request from web API and send them to sub services.
Any request has a header which use to detect command type (suc as Reserve, Refund, Availability or etc.), then deserialize to object and using it.
Now, Who to send various command type by MassTransit from a publisher such as our supervisor and get them in consumer and use it?
Thanks

Reply

Calabonga says:
July 21, 2017 at 4:22 am
I need help. I was created the solution as you shown and everythig works fine, but there is one issue. The RabbitMQ alway creates additional queue which name ended "_skipped". But solution works as expected correctly.

Please, answer the questions: What is the queue "_skipped"? Is is ok or where I was wrong?

Reply

Tim B. says:
August 17, 2017 at 2:47 am
@Calabonga I would look at the following stackoverflow reply https://stackoverflow.com/a/34911002 specifically:
"For the receive endpoint, it … perhaps not consuming the correct message type. The message type must be the same message contract in both the consumer and publisher for the message to be consumed by the consumer.
When the message is moved to _skipped, there is no consumer on that endpoint actually consuming the message types in the message itself…"

Reply

Blog at WordPress.com.