

Messaging through a service bus in .NET using MassTransit part 7: intercepting messages

OCTOBER 4, 2016 5 COMMENTS ([HTTPS://DOTNETCODR.COM/2016/10/04/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-7-INTERCEPTING-MESSAGES/#COMMENTS](https://dotnetcodr.com/2016/10/04/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-7-INTERCEPTING-MESSAGES/#COMMENTS))

Introduction

In the [previous post \(https://dotnetcodr.com/2016/09/29/messaging-through-a-service-bus-in-net-using-masstransit-part-6-message-types-and-inheritance-support/\)](https://dotnetcodr.com/2016/09/29/messaging-through-a-service-bus-in-net-using-masstransit-part-6-message-types-and-inheritance-support/), we investigated how types are handled by MassTransit and how they are represented in RabbitMQ exchanges and queues. With MassTransit we can send and receive concrete objects and it will take care of creating the necessary queues and exchanges for us. The objects are serialised and deserialised without us having to worry about JSON or XML strings.

In this post we'll see how to intercept messages both in the publisher and receiver.

The message observers

Messages can be intercepted and inspected using 5 different interfaces in MassTransit. Note that we cannot modify the messages with them, i.e. they are read-only operations.

They can be divided into two groups:

Publisher-based message observers

These are the observers that intercept published and sent messages of any message type. They are not generic in the sense that we cannot intercept just any type of message.

- `ISendObserver`
- `IPublishObserver`

Consumer-based message observers

With these observers we can intercept messages received and consumed.

- `IReceiveObserver`: to intercept any received message of any concrete type
- `IConsumeObserver`: to intercept any consumed message of any concrete type
- `IConsumeMessageObserver of T`: the generic version of `IConsumeObserver` to intercept specific message types consumed

Intercepting messages received

`IReceiveObserver` is the largest of the options with 5 methods to be implemented:

- `ConsumeFault`
- `PostConsume`
- `PostReceive`
- `PreReceive`
- `ReceiveFault`

The method names are descriptive of their respective role. All of them return a `Task` i.e. they are well suited for `await-async` operations. Insert the following `MessageReceiveObserver` class into the `MassTransit.Receiver` project of the demo application:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace MassTransit.Receiver
8  {
9      public class MessageReceiveObserver : IReceiveObserver
10     {
11         private readonly string _logger = "[MessageReceiveObserver]";
12
13         public async Task ConsumeFault<T>(ConsumeContext<T> context, TimeSpan duration, string consum
14         {
15             Console.WriteLine(string.Concat(_logger, ": A fault is consumed for type ", context.Messa
16             await context.CompleteTask;
17         }
18
19         public async Task PostConsume<T>(ConsumeContext<T> context, TimeSpan duration, string consume
20         {
21             Console.WriteLine(string.Concat(_logger, ": A message is consumed for type ", context.Mes
22             ". Consumer type: ", consumerType, ", duration in millis: ", duration.TotalMillisecon
23             await context.CompleteTask;
24         }
25
26         public async Task PostReceive(ReceiveContext context)
27         {
28             Console.WriteLine(string.Concat(_logger, ": A message is past the receive stage. Has been
29             context.IsDelivered));
30             await context.CompleteTask;
31         }
32
33         public async Task PreReceive(ReceiveContext context)
34         {
35             Console.WriteLine(string.Concat(_logger, ": A message is before the receive stage. Has be
36             context.IsDelivered));
37             await context.CompleteTask;
38         }
39
40         public async Task ReceiveFault(ReceiveContext context, Exception exception)
41         {
42             Console.WriteLine(string.Concat(_logger, ": A fault is received with exception ", excepti
43             await context.CompleteTask;
44         }
45     }
46 }

```

We need to register the observer with the bus control. Add the following line of code...

```
1 | rabbitBusControl.ConnectReceiveObserver(new MessageReceiveObserver());
```

...just after...

```
1 | rabbitBusControl.Start();
```

...in the RunMassTransitReceiverWithRabbit method in Program.cs of MassTransit.Receiver. Start the receiver project and then the publisher MassTransit.Publisher. You can set breakpoints within the various methods of the observer to see how and when they are triggered. If everything goes fine then you should see messages similar to the following in the consumer's command window:

```

[MessageReceiveObserver]: A message is before the receive stage. Has been delivered to at least one consumer: False
New domain registered. Target and importance: Customers / 1
[MessageReceiveObserver]: A message is consumed for type
DynamicInternal08d3ec860054c792fc3fdb87d1ba0000.MyCompany.Messaging.IRegisterDomain. Consumer type:
MassTransit.Receiver.RegisterDomainConsumer, duration in millis: 3,8467
A new customer has signed up, it's time to register it in the command receiver. Details:
New Street

```

Nice people LTD

dc049caf-e645-4d31-8671-81b4f29178b0

True

The concrete customer repository was called for customer Nice people LTD

[MessageReceiveObserver]: A message is consumed for type

DynamicInternal08d3ec860054c792fc3fdb87d1ba0000.MyCompany.Messaging.IRegisterCustomer. Consumer type:

MassTransit.Receiver.RegisterCustomerConsumer, duration in millis: 84,9193

[MessageReceiveObserver]: A message is past the receive stage. Has been delivered to at least one consumer: True

Note that both IRegisterDomain and IRegisterCustomer are observed as they should. The ConsumeContext class is like the one we saw in the various consumer classes like RegisterCustomerConsumer, i.e. it offers the same level of information like the conversation ID and all the rest.

Let's now try the consume observer interface. It has the PostConsume and ConsumeFault methods like above but it also has a PreConsume method, i.e. we get to analyse the context before the message is consumed.

Here's a minimal implementation of the IConsumeObserver interface in the MassTransit.Receiver project:

```
1  using MassTransit.Pipeline;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace MassTransit.Receiver
9  {
10     public class MessageConsumeObserver : IConsumeObserver
11     {
12         private readonly string _logger = "[MessageConsumeObserver]";
13         public async Task ConsumeFault<T>(ConsumeContext<T> context, Exception exception) where T : class
14         {
15             Console.WriteLine(string.Concat(_logger, ": A fault is consumed for type ",
16                 context.Message.GetType(), ", exception message: ", exception.Message));
17             await context.CompleteTask();
18         }
19
20         public async Task PostConsume<T>(ConsumeContext<T> context) where T : class
21         {
22             Console.WriteLine(string.Concat(_logger, ": A message is past the consume stage for type "));
23             await context.CompleteTask();
24         }
25
26         public async Task PreConsume<T>(ConsumeContext<T> context) where T : class
27         {
28             Console.WriteLine(string.Concat(_logger, ": A message is before the consume stage for type "));
29             await context.CompleteTask();
30         }
31     }
32 }
```

Again, we need to register the observer with the bus control:

```
1  //rabbitBusControl.ConnectReceiveObserver(new MessageReceiveObserver());
2  rabbitBusControl.ConnectConsumeObserver(new MessageConsumeObserver());
```

I commented out the MessageReceiveObserver registration so that we can easily see the messages from the consumer observer. The same test yields the following output in the receiver's command window:

[MessageConsumeObserver]: A message is before the consume stage for type

DynamicInternal08d3ec87df9416fbfc3fdb87d1ba0000.MyCompany.Messaging.IRegisterCustomer

A new customer has signed up, it's time to register it in the command receiver. Details:

New Street

Nice people LTD

e2a1a036-d5f5-417d-83a8-d3d0b0a0ab76

True

The concrete customer repository was called for customer Nice people LTD

[MessageConsumeObserver]: A message is past the consume stage for type

DynamicInternal08d3ec87df9416fbfc3fdb87d1ba0000.MyCompany.Messaging.IRegisterCustomer

[MessageConsumeObserver]: A message is before the consume stage for type

DynamicInternal08d3ec87df9416fbfc3fdb87d1ba0000.MyCompany.Messaging.IRegisterDomain

New domain registered. Target and importance: Customers / 1

[MessageConsumeObserver]: A message is past the consume stage for type

DynamicInternal08d3ec87df9416fbfc3fdb87d1ba0000.MyCompany.Messaging.IRegisterDomain

The generic IConsumeMessageObserver of T interface is the same as the non-generic IConsumeObserver but for specific message types, e.g.:

```
1 public class RegisterCustomerMessageObserver : IConsumeMessageObserver<IRegisterCustomer>
2 {
3     //code ignored
4 }
```

It has the same methods as IConsumeObserver so we'll not go through a separate example for that. It too requires to be registered with the bus:

```
1 rabbitBusControl.ConnectConsumeMessageObserver(new RegisterCustomerMessageObserver());
```

Intercepting messages sent

The usage of ISendObserver and IPublishObserver are about the same as the 3 receiver observers above. Note that at the time of writing this post there's no generic version of either of these interfaces. Here's an ISendObserver that I'm adding to the MassTransit.Publisher console application:

```
1 using MassTransit;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace MassTransitTests
9 {
10     public class SendObjectObserver : ISendObserver
11     {
12         private readonly string _logger = "[SendObjectObserver]";
13         public async Task PostSend<T>(SendContext<T> context) where T : class
14         {
15             Console.WriteLine(string.Concat(_logger, ": A message is post the sending stage for type ");
16             await Task.FromResult(0);
17         }
18
19         public async Task PreSend<T>(SendContext<T> context) where T : class
20         {
21             Console.WriteLine(string.Concat(_logger, ": A message is before the sending stage for typ");
22             await Task.FromResult(0);
23         }
24
25         public async Task SendFault<T>(SendContext<T> context, Exception exception) where T : class
26         {
27             Console.WriteLine(string.Concat(_logger, ": A message fault is sent for the type ", conte
28                 , ", exception: ", exception.Message));
29             await Task.FromResult(0);
30         }
31     }
32 }
```

It needs to be registered with bus in the RunMassTransitPublisherWithRabbit method:

```
1 | rabbitBusControl.ConnectSendObserver(new SendObjectObserver());
```

...just after the IBusControl declaration.

The publisher's command window will show the following messages:

[SendObjectObserver]: A message is before the sending stage for type
DynamicInternal08d3ec8af8a9b9bffc3fdb87d1ba0000.MyCompany.Messaging.IRegisterCustomer

[SendObjectObserver]: A message is post the sending stage for type
DynamicInternal08d3ec8af8a9b9bffc3fdb87d1ba0000.MyCompany.Messaging.IRegisterCustomer

The IPublishObserver has the same function but it's for intercepting published messages. You can surely implement it yourself after these examples. It needs to be registered with the IBusControl.ConnectPublishObserver method.

So, in case you want to do something extra with the messages sent, published, consumed or received then it's a piece of cake using these interfaces.

We'll continue in the [next post by looking at intercepting bus-related events \(https://dotnetcodr.com/2016/10/06/messaging-through-a-service-bus-in-net-using-masstransit-part-8-observing-events-in-the-bus/\)](https://dotnetcodr.com/2016/10/06/messaging-through-a-service-bus-in-net-using-masstransit-part-8-observing-events-in-the-bus/).

View the list of posts on Messaging [here \(https://dotnetcodr.com/messaging/\)](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.

5 Responses to *Messaging through a service bus in .NET using MassTransit part 7: intercepting messages*

ramessesx says:

[October 5, 2016 at 11:18 am](#)

Love these posts Andras, thanks again for your time in putting them together.

Reply

[Andras Nemes says:](#)

[October 5, 2016 at 2:45 pm](#)

Thanks for your comment, I'm glad you like the posts. //Andras

Reply

Hasibul says:

[June 19, 2017 at 10:15 pm](#)

Hi Andras, Your posts are awesome. I always follow your posts. They are very helpful and informative. Thank you very much for sharing..

It will be great if you have some time to check following question regarding message interception. I am expecting your help/suggestion.

<https://stackoverflow.com/questions/44639244/how-to-create-masstransit-message-interceptor-observer-with-scoped-lifetime>

Reply

[Andras Nemes says:](#)

[June 20, 2017 at 8:24 am](#)



Hi Hasibul, thanks for your message, I'll what I can do. //Andras

Reply

[Hasibul Haque says:](#)

[June 20, 2017 at 9:43 am](#)

Just want to know Can we intercept the message with scoped life cycle? If yes then how we can do that? Expecting your help.

JustAnswer

Chat w/ an expert Online Now

REPORT THIS AD