Messaging through a service bus in .NET using MassTransit part 6: message types and inheritance support

SEPTEMBER 29, 2016 1 COMMENT (HTTPS://DOTNETCODR.COM/2016/09/29/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-6-MESSAGE-TYPES-AND-INHERITANCE-SUPPORT/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/28/messaging-through-a-service-bus-in-net-using-masstransit-part-5-failures/) we went through how exceptions are handled in MassTransit. By default the only mechanism is that if a registered consumer throws an exception while processing a message then that message ends up in an error queue. The error queue is named after the queue name where the consumer is listening with "_error" attached to it. By default MassTransit won't try to relay the message again. However, it's simple to add various retry policies with the UseRetry extension method. We can configure a wide range of retry policies: incremental, exponential, exception-based ones and other types. In addition MassTransit publishes a Fault message if all retries have been exhausted without success. The fault address or response address can specify a different queue where the fault message will be delivered. A different consumer can then monitor the fault queue and do something meaningful with the fault message.

In this post we'll look at how message type inheritance is supported in MassTransit.

Inheritance

By now we know that MassTransit has a great seralisation mechanism. We can publish and consume strongly typed objects without worrying about JSON/XML etc. strings. MassTransit is actually even more intelligent than we first might think.

Imagine that there's a new requirement that all commands must include two new properties: a target and an importance level. One way to achieve that is first adding the following interface to the MyCompany.Messaging C# library:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyCompany.Messaging
{
    public interface IRegisterDomain
    {
        string Target { get; }
        int Importance { get; }
    }
}
```

Then we modify IRegisterCustomer to implement this interface:

```
namespace MyCompany.Messaging
{
    public interface IRegisterCustomer : IRegisterDomain
    {
        //properties ignored
    }
}
```

Next we add the new properties to the anonymous object when sending an IRegisterCustomer from MassTransit.Publisher:

```
Task sendTask = sendEndpoint.Send<IRegisterCustomer>(new
{
    Address = "New Street",
    Id = Guid.NewGuid(),
    Preferred = true,
    RegisteredUtc = DateTime.UtcNow,
    Name = "Nice people LTD",
    Type = 1,
    DefaultDiscount = 0,
    Target = "Customers",
    Importance = 1
});
```

Say we need a consumer that captures the common information over all domain registration events. Add the following consumer to MassTransit.Receiver:

```
using MyCompany.Messaging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MassTransit.Receiver
{
    public class RegisterDomainConsumer : IConsumer<IRegisterDomain>
    {
        public Task Consume(ConsumeContext<IRegisterDomain> context)
        {
            Console.WriteLine(string.Concat("New domain registered. Target and importance: ",
                context.Message.Target, " / ", context.Message.Importance));
            return Task.FromResult<IRegisterDomain>(context.Message);
        }
    }
}
```

For simplicity we just add this consumer to the existing ReceiveEndpoint configuration of Program.cs in MassTransit.Receiver:

```
rabbit.ReceiveEndpoint(rabbitMqHost, "mycompany.domains.queues", conf =>
{
    conf.Consumer<RegisterCustomerConsumer>(container);
    conf.Consumer<RegisterDomainConsumer>();
    //any retry policy code ignored
});
```

If you still have the exception throwing code in RegisterCustomerConsumer from the previous post then you can remove it.

Start the demo as usual: start MassTransit.Receiver and then MassTransit.Publisher. You should see that both consumers, i.e. RegisterCustomerConsumer and RegisterDomainConsumer successfully received the message. The consumer command window should show the message from the RegisterDomainConsumer along with that from RegisterCustomerConsumer:

“A new customer has signed up, it’s time to register it in the command receiver. Details:
New Street
Nice people LTD
9ce4f92a-ad92-4c53-aa74-cd2405c6c814
True
The concrete customer repository was called for customer Nice people LTD
New domain registered. Target and importance: Customers / 1”

There are a couple of important points to note here:

- We sent an IRegisterCustomer from the publisher, not an IRegisterDomain. However, MassTransit was still able to deseralise the properties that belong to IRegisterDomain and send the message to the RegisterDomainConsumer. Hence MassTransit can understand inheritance
- We can set up consumers for generic interfaces like IRegisterDomain if we want to capture all common commands and events in our application. It’s not necessary to separately publish or send messages of those generic types, it’s enough that they implement

or extend them. MassTransit will be able to work with inheritance.
- This is great for message versioning if we want to extend the message type with new properties. We can add new interfaces and implement them, the existing functionality won't break
- Not related to inheritance, but we saw that we could safely set up multiple consumers for the same receive endpoint like we did above and all will be notified of the message

MassTransit exchanges and queues

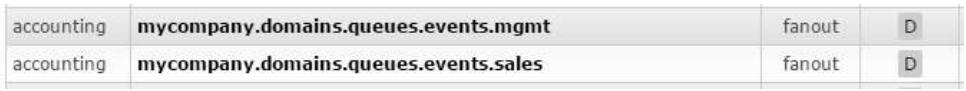
It's an interesting implementation detail to see how MassTransit handles these message types in RabbitMq. First of all an exchange is created for each message type we send or publish. Here are the ones we have worked with in this series so far:

(https://dotnetcodr.files.wordpress.com/2016/09/message-type-exchanges-in-rabbitmq-masstransit.png)

Each exchange is linked to one or more other exchanges depending on the consumers of the type. E.g. we had 2 consumers for the ICustomerRegistered event: CustomerRegisteredConsumerMgmt where we declared a queue called "mycompany.domains.queues.events.mgmt" and CustomerRegisteredConsumerSls with a queue named "mycompany.domains.queues.events.sales". If you click on the details of the ICustomerRegistered exchange then you'll see that it's linked with 2 other exchanges:

(https://dotnetcodr.files.wordpress.com/2016/09/message-type-exchange-linked-to-listeners-exchanges-in-rabbitmq-masstransit.png)

These exchanges have the same names as the queues we have declared. They are also listed on the exchanges list like the ones based on the message type:



(https://dotnetcodr.files.wordpress.com/2016/09/message-queue-name-exchanges-in-masstransit-rabbitmq.png)

In addition each exchange named after the queues declared in code have a queue attached to them with the same name. I.e. the exchange called "mycompany.domains.queues.events.mgmt" has a queue called "mycompany.domains.queues.events.mgmt" linked to it. The same is true of the "mycompany.domains.queues.events.sales" exchange. The queues are listed on the Queues page of course:

(https://dotnetcodr.files.wordpress.com/2016/09/queues-belonging-to-their-exchanges-in-masstransit-rabbitmq.png)

So the message type is a very important ingredient for the MassTransit implementation details when creating new exchanges and queues.

We'll continue in the next post with message interception (https://dotnetcodr.com/2016/10/04/messaging-through-a-service-bus-in-net-using-masstransit-part-7-intercepting-messages/).

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.

One Response to *Messaging through a service bus in .NET using MassTransit part 6: message types and inheritance support*

Rory Thompson says:
December 5, 2016 at 8:53 pm
I'm having a different issue. We deployed our masstransit bus onto a dev server (remote machine now) along with rabbitmq. From another machine, we send the request using _client.Request interface. When the bus picks up the message, it does it's work and then does context.Respond(…). When this happens, it replies back to the same queue on the same machine, and then ends up picking it up again. It's also creating error messages into the error queue saying it cannot find the respondEndpoint that was on the original message. Any idea why this could be happening?

When both the client and the bus were on the same machine (localhost), the response/request was working just fine and as expected.

Reply