Exercises in .NET with Andras Nemes

Tips and tricks in C# .NET

Messaging through a service bus in .NET using MassTransit part 5: failures

SEPTEMBER 28, 2016 LEAVE A COMMENT (HTTPS://DOTNETCODR.COM/2016/09/28/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-5-FAILURES/#RESPOND)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/22/messaging-through-a-service-bus-in-net-using-masstransit-part-4-dependency-injection-with-structuremap/) we explored how to inject a dependency into the registered consumer class in MassTransit. Consumer classes will often have at least some sort of dependency such as a repository interface or another abstraction to propagate the changes made. Good software engineering dictates that a class should indicate what dependencies it needs through e.g. its constructor. This is the contrary of control-freak objects that construct all their dependencies hidden in their implementations.

In this post we'll take a look at various failure handling options in MassTransit.

Exceptions in the consumer class

It happens that an exception is thrown in the consumer class so that it cannot acknowledge the message. What happens then? Let's see.

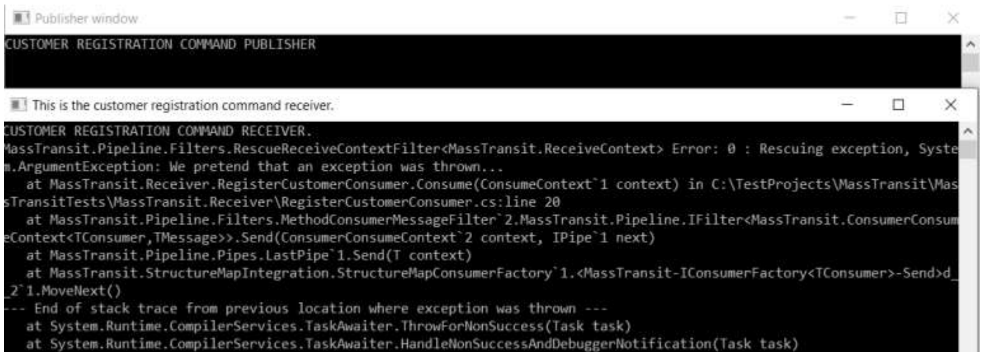
Currently we have a consumer called RegisterCustomerConsumer in our MassTransit.Receiver console demo application. Its implementation starts like this:

```
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 r
```

Insert the following line just before newCustomer is declared:

```
throw new ArgumentException("We pretend that an exception was thrown...");
```

Visual Studio will warn you that the rest of the implementation is unreachable, but we don't care for now. Start the receiver application and then also start MassTransit.Publisher. The publisher will send the usual IRegisterCustomer object. The exception is thrown in the consumer and you'll see the stacktrace printed in the consumer's command window:

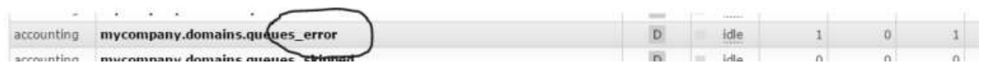


```
CUSTOMER REGISTRATION COMMAND PUBLISHER
```

```
CUSTOMER REGISTRATION COMMAND RECEIVER.
MassTransit.Pipeline.Filters.RescueReceiveContextFilter<MassTransit.ReceiveContext> Error: 0 : Rescuing exception, Syste
m.ArgumentException: We pretend that an exception was thrown...
   at MassTransit.Receiver.RegisterCustomerConsumer.Consume(ConsumeContext`1 context) in C:\TestProjects\MassTransit\Mas
sTransitTests\MassTransit.Receiver\RegisterCustomerConsumer.cs:line 20
   at MassTransit.Pipeline.Filters.MethodConsumerMessageFilter`2.MassTransit.Pipeline.IFilter<MassTransit.ConsumerConsum
eContext<TConsumer,TMessage>>.Send(ConsumerConsumeContext`2 context, IPipe`1 next)
   at MassTransit.Pipeline.Pipes.LastPipe`1.Send(T context)
   at MassTransit.StructureMapIntegration.StructureMapConsumerFactory`1.<MassTransit-IConsumerFactory<TConsumer>-Send>d_
_2`1.MoveNext()
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
```

(https://dotnetcodr.files.wordpress.com/2016/09/consumer-throws-exception-in-masstransit.png)

So what happened to the message? By default it ends up in another queue. This queue is named after the queue that the receiver is listening to with "_error" attached to it. Here's our message:



accounting	mycompany.domains.queues_error	D		idle	1	0	1
accounting	mycompany.domains.queues_skipped	D		idle	0	0	0

(https://dotnetcodr.files.wordpress.com/2016/09/error-queue-for-unacked-messages-in-masstransit.png)

As extra information we can mentioned that there are also "_skipped" queues, like "mycompany.domains.queues_skipped" in our case. Skipped queues store messages that cannot be routed to the queue for some reason. Therefore it's important to check the _error and _skipped queues periodically. You can even set up consumers for these queues if you to process them further, e.g. log them for later inspection.

Exception handling strategies

We saw above that the publisher published a message and the message ended up in the error queue after a single try. We can declare various retry policies in MassTransit but by default there is no retry policy at all. We can easily declare a retry policy for a specific consumer or for the service bus as a whole.

Probably the most basic retry policy is to tell MassTransit to resend a message a number of times before giving up, i.e. sending the message to the error queue. The retry policy for the receiver is declared in the ReceiveEndpoint method as follows:

```
rabbit.ReceiveEndpoint(rabbitMqHost, "mycompany.domains.queues", conf =>
{
    conf.Consumer<RegisterCustomerConsumer>(container);
    conf.UseRetry(Retry.Immediate(5));
});
```

The UseRetry extension method accepts an IRetryPolicy interface where the Retry class offers a number of shortcuts to build an object that implements the interface. The Immediate method accepts an integer and sets up a policy to resend a message that many times with no delay in between. The same UseRetry extension method is available within the Action supplied to the CreateUsingRabbitMq function:

```
IBusControl rabbitBusControl = Bus.Factory.CreateUsingRabbitMq(rabbit =>
{
        //rest of code ignored
    rabbit.UseRetry(...)
});
```

This assigns the retry policy to the bus, so it's a more general policy than on the consumer level. If you now run the same test then you'll see that the same command message is re-sent 5 times in quick succession before it ends up in the _error queue.

The Retry class exposes a wide range of retry policies. If you type "Retry." in the editor then IntelliSense will show you several options. The Except exception will run the retry policy EXCEPT for the exception type specified:

```
conf.UseRetry(Retry.Except(typeof(ArgumentException)).Immediate(5));
```

If you test the code now you'll see that the Immediate policy is bypassed since we want MassTransit to ignore argument exceptions. The Except function accepts an array of Exception types so you can specify as many as you want.

The opposite case is the Selected function where we can provide the exception types for which the retry policy should be applied:

```
conf.UseRetry(Retry.Selected(typeof(ArgumentException)).Immediate(5));
```

Retry.All() will include all exception types in the retry policy. Finally we have a Filter method which accepts a Func that returns a bool. Here we can fine grain our exception filtering logic. The Func has the exception as its input parameter. Here's an example:

```
conf.UseRetry(Retry.Filter<Exception>(e => e.Message.IndexOf("We pretend that an exception was thrown"
```

Filter, Selected, Except and All were all exception related retry policy filters. On the other hand we have the time and frequency based retry policies like Immediate. This latter group includes a number of other functions.

Using the Exponential policy builder we can specify a min and max interval for the time between retries as follows:

```
conf.UseRetry(Retry.Exponential(5, TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(30), TimeSpan.FromSec
```

The first integer is the max number of retries. Then come the min and max intervals followed by the delta. If you test the code with this policy you'll see that the wait times between retries keeps increasing exponentially. Exponential has an overload without the max number of retries if you want MassTransit to keep retrying forever.

A similar policy is Incremental where we can provide a max number of retries, an initial delay and an interval increment:

```
conf.UseRetry(Retry.Incremental(5, TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(3)));
```

With the Interval function we can supply a retry count and an interval between each retry:

```
conf.UseRetry(Retry.Interval(5, TimeSpan.FromSeconds(5)));
```

Finally we have the Intervals function which accepts an array of retry intervals:

```
conf.UseRetry(Retry.Intervals(TimeSpan.FromSeconds(3), TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(4
```

So we have a number of interesting options here.

Faults

When MassTransit has finished retrying with no success then it issues an object of type Fault of T where T is the type of command or event for which all retries have failed. We can set up a consumer for it as follows:

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

namespace MassTransit.Receiver
{
    public class RegisterCustomerFaultConsumer : IConsumer<Fault<IRegisterCustomer>>
    {
        public Task Consume(ConsumeContext<Fault<IRegisterCustomer>> context)
        {
            IRegisterCustomer originalFault = context.Message.Message;
            ExceptionInfo[] exceptions = context.Message.Exceptions;
            return Task.FromResult(originalFault);
        }
    }
}
```

We can register the above consumer like we did before:

```
rabbit.ReceiveEndpoint(rabbitMqHost, "mycompany.queues.errors.newcustomers", conf =>
{
    conf.Consumer<RegisterCustomerFaultConsumer>();
});
```

For the above to work properly we have to set up a fault address when sending or publishing the message in the publisher. Here's an example:

```
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
}, c => c.FaultAddress = new Uri("rabbitmq://localhost:5672/accounting/mycompany.queues.errors.newcus
```

The same FaultAddress property can be configured in the Publish method as well. An alternative to the FaultAddress is the ResponseAddress which MassTransit will inspect if a FaultAddress is not present.

If you test the above code then you'll see that the fault consumer receives the Fault object after all retries have been exhausted.

You can find the exception handling documentation of MassTransit here (http://docs.masstransit-project.com/en/latest/usage/exceptions.html).

We'll continue with exploring how MassTransit handles types in the next post (https://dotnetcodr.com/2016/09/29/messaging-through-a-service-bus-in-net-using-masstransit-part-6-message-types-and-inheritance-support/).

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.