Exercises in .NET with Andras Nemes

Tips and tricks in C# .NET

Messaging through a service bus in .NET using MassTransit part 2: starting with some code

SEPTEMBER 14, 2016 1 COMMENT (HTTPS://DOTNETCODR.COM/2016/09/14/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-2-STARTING-WITH-SOME-CODE/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/08/messaging-through-a-service-bus-in-net-using-masstransit-part-1-foundations/) we presented the topic of this series: using the MassTransit service bus implementation in .NET. We also discussed service buses in general, what they are, what they do and how they can help the developer concentrate on the "important" stuff in a project based around messaging instead of spending time on low-level messaging operations. This is not to say that a service bus is a must for a distributed system to work properly. The system may as well work with RabbitMq only without an extra level of abstraction on top of it. Therefore you must be careful with your design choices. This is especially true of full-blown ESBs – enterprise service buses – that cost a lot of money and introduce a high level of complexity in a distributed architecture.

In this post we'll start writing code and send an object over the wire using MassTransit.

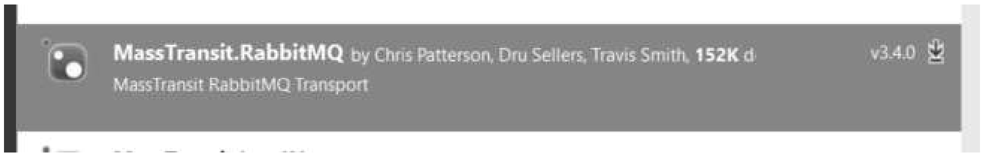
Note that I originally wanted to start with the basic in-memory version of MassTransit, i.e. without a backing message broker. However, that type of messaging is not supported across different processes (http://masstransit.readthedocs.io/en/master/configuration/transports/in_memory.html). This means that if we send a message in one console project and try to catch it in an independent one then it won't work. I tried a couple of ways to get round that limitation but didn't succeed. On the other hand we have a good basic knowledge of RabbitMq from the previous series and using the in-memory pipes for messaging is never good for a production system anyway. Therefore it's still a good idea to jump into the RabbitMq-based code right away.

The demo project skeleton

Create a blank solution in Visual Studio and add 3 projects to it:

- A console project called MassTransit.Publisher
- A console project called MassTransit.Receiver
- A class library called MyCompany.Messaging

Add the following NuGet package to the console projects:


MassTransit.RabbitMQ by Chris Patterson, Dru Sellers, Travis Smith, 152K d v3.4.0
MassTransit RabbitMQ Transport

(https://dotnetcodr.files.wordpress.com/2016/09/masstransit-nuget-package.png)

This will add other dependencies such as RabbitMq.Client as well.

Also, add a reference to MyCompany.Messaging from both the publisher and the receiver console applications.

The message

MassTransit handles serialisation and deserialisation of objects automatically. Normally we don't need to send JSON, XML etc. messages around with MassTransit, it's perfectly fine to work with objects. The relevant documentation (http://masstransit.readthedocs.io/en/master/usage/messages.html) suggests that we wrap the properties of the message in an interface rather than a class. Here's a warning from the creators of MassTransit:

"A common mistake when engineers are new to messaging is to create a base class for messages, and try to dispatch that base class in the consumer – including the behavior of the subclass. Ouch. This always leads to pain and suffering, so just say no to base classes."

In addition there are two types of message and each comes with a naming convention:

- **Commands**: these tell the consumer to perform some action. Commands should be **sent** to a single endpoint. The consumer of that endpoint is expected to perform the action. Command names follow the verb-noun convention such as RegisterCustomer or UpdateOrder.
- **Events**: events are actions that have happened in the system. Events are **published** to whoever is interested in them. Events follow the naming convention noun-verb in past tense such as CustomerRegistered or OrderUpdated

In this post we'll concentrate on commands. We'll follow the above advice and create an interface for our first command message. Insert the following interface into MyCompany.Messaging:

```
namespace MyCompany.Messaging
{
    public interface IRegisterCustomer
    {
        Guid Id { get; }
        DateTime RegisteredUtc { get; }
        int Type { get; }
        string Name { get; }
        bool Preferred { get; }
        decimal DefaultDiscount { get; }
        string Address { get; }
    }
}
```

That should be fairly simple. Note that it's perfectly fine to have your "normal" domain objects such as Customer as classes. It's only the message contracts that take on an interface format.

The consumer

We need to register a consumer with the service bus. One way to achieve it is to first implement the IConsumer of T interface where T is the type of the message contract, in our case IRegisterCustomer. The interface has a single method called Consume which takes a ConsumeContext parameter. The context will include the deserialised message. Add the following class to MassTransit.Receiver:

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

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

The Consume function returns a Task, i.e. it is meant to be used in an asynchronous fashion. We're in a console application which doesn't play well with the await and async keywords so we'll skip that.

Next we'll build a bus control using the Bus class and its Factory property. This factory has an extension method called CreateUsingRabbitMq where we can configure the RabbitMq endpoint. We can configure a whole range of other things too but we'll concentrate on two things to begin with: configuring the RabbitMq host, password and username, and registering the RegisterCustomerConsumer consumer. The bus control object also enables us to start and stop the service bus. Here's the Program.cs file of the Receiver console application:

```
using MassTransit.RabbitMqTransport;
using MassTransit.Transports.InMemory;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MassTransit.Receiver
{
    class Program
    {
        static void Main(string[] args)
        {
            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", conf =>
                {
                    conf.Consumer<RegisterCustomerConsumer>();
                });
            });

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

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

Note the URI format in the Host method. It's different from what we had when we worked directly with RabbitMq: amqp://accountant:accountant@localhost:5672/accounting. The MassTransit URI doesn't allow setting the username and password directly in the connection string. We can only set the domain name and the virtual host. The virtual host is "accounting" in the above example which we created in the previous series on RabbitMq.

We can provide the queue name in the ReceiveEndpoint function. In this case it is "mycompany.domains.queues". Note how the RegisterCustomerConsumer is registered using the Consumer function.

Finally we start the service bus and don't let it stop until the user presses a button.

The publisher

The bus control creation bit is identical except that we don't register a consumer. Since RegisterCustomer is a command we'll send it to a single queue. The bus control lets us build an endpoint for that. The endpoint, represented by the ISendEndpoint interface allows us to perform the actual sending action. The following example shows how to do it. This is from Program.cs of the Publisher console application

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

namespace MassTransitTests
{
    class Program
    {
        public static object RegisterNewOrderConsumer { get; private set; }

        static void Main(string[] args)
        {
            RunMassTransitPublisherWithRabbit();
        }

        private static void RunMassTransitPublisherWithRabbit()
        {
            string rabbitMqAddress = "rabbitmq://localhost:5672/accounting";
            string rabbitMqQueue = "mycompany.domains.queues";
            Uri rabbitMqRootUri = new Uri(rabbitMqAddress);

            IBusControl rabbitBusControl = Bus.Factory.CreateUsingRabbitMq(rabbit =>
            {
                rabbit.Host(rabbitMqRootUri, settings =>
                {
                    settings.Password("accountant");
                    settings.Username("accountant");
                });
            });

            Task<ISendEndpoint> sendEndpointTask = rabbitBusControl.GetSendEndpoint(new Uri(string.Co
            ISendEndpoint sendEndpoint = sendEndpointTask.Result;

            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
            });
            Console.ReadKey();
        }
    }
}
```

Note how we implemented the interface using an anonymous object.

Testing the components

You can set various breakpoints if you want to step through the code yourself. At a minimum place a breakpoint within the Consume function of the IConsumer so that you won't miss when it's triggered. First start the receiver application and let it run complete. Execution will will wait for the Console.ReadKey function. At that point the service bus should be running with a consumer registered. Next start the publisher and let it also run until Console.ReadKey() blocks the application from exiting. You should see that the Consume function is triggered and you should see a message similar to the following in the Debug window:

A new customer has signed up, it's time to register it. Details:
New Street
Nice people LTD

```
128d8fa1-7d50-4196-9f40-da6449b14c7e
True
```

You can even start multiple instances of the publisher application without turning off the consumer. The message should be delivered to the consumer as expected.

We'll continue coding in the next post (https://dotnetcodr.com/2016/09/20/messaging-through-a-service-bus-in-net-using-masstransit-part-3-publishing-messages-to-multiple-consumers/).

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 2: starting with some code*

tatsean says:
August 16, 2017 at 8:31 am
I have a question. I set a breakpoint at "sendEndpoint.Send(new", and when the execution hits the breakpoint, i stop the RabbitMQ window service, and after that press F5 to continue to the code execution. However, the program continues without any exception but problem is since i already stop the RabbitMQ windows service, the message indeed not pump into the queue. So, the biggest issue here is, why sendEndpoint.Send does not throw any exception in this case, any insight?

Reply