Messaging through a service bus in .NET using MassTransit part 4: dependency injection with StructureMap

SEPTEMBER 22, 2016 LEAVE A COMMENT (HTTPS://DOTNETCODR.COM/2016/09/22/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-4-DEPENDENCY-INJECTION-WITH-STRUCTUREMAP/#RESPOND)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/20/messaging-through-a-service-bus-in-net-using-masstransit-part-3-publishing-messages-to-multiple-consumers/) we saw how to publish messages with MassTransit. Our demo register customer command listener publishes a customer registered event that any interested party can sign up to. An advantage with the setup is that the publisher has no knowledge of the consumers, they are completely decoupled. The opposite and more frequent case is often a manager application that knows exactly which parties should be notified and sends them a message directly.

In this post we'll diverge a little from messaging techniques and look at dependency injection instead. We're inte particular interested in how to inject a dependency into a registered command or event consumer.

Never heard of dependency injection?

If you are working with relatively advanced topics such as a .NET service bus then I assume that you have also at least come across terms like dependency injection, inversion of control or dependency inversion. They are often used to mean the same thing: a class which depends on another class or interface to perform its functions should be supplied that dependency by the caller. This often manifests itself in constructors which inform the callers what dependencies they need. Dependency injection is not some fancy C# language construct. The challenging part is most often to identify the tightly coupled dependencies and factor them out into separate and independent pieces.

The following is an example of a constructor injection, i.e. where a class indicates its "needs" via the constructor. This is by far the most common form of dependency injection:

```
public class PublicMessage
{
    private readonly IMessageCollector _messageCollector;
    private readonly ITextWriter _textWriter;

    public PublicMessage(IMessageCollector messageCollector, ITextWriter textWriter)
    {
        if (messageCollector == null) throw new ArgumentNullException("Message collector");
        if (textWriter == null) throw new ArgumentNullException("Text writer");
        _messageCollector = messageCollector;
        _textWriter = textWriter;
    }

    public void Shout()
    {
        string message = _messageCollector.CollectMessageFromUser();
        _textWriter.WriteText(message);
    }
}
```

There's a number of posts on this blog dedicated to dependency injection. Go ahead and read them if you want to refresh your memory:

- SOLID design principles in .NET: the Dependency Inversion Principle and the Dependency Injection pattern (https://dotnetcodr.com/2013/08/26/solid-design-principles-in-net-the-dependency-inversion-principle-and-the-dependency-injection-pattern/)

- SOLID principles in .NET revisited part 7: the Dependency Inversion Principle (https://dotnetcodr.com/2015/05/14/solid-principles-in-net-revisited-part-7-the-dependency-inversion-principle/)
- A whole range of practical DI examples starting here (https://dotnetcodr.com/2014/09/01/externalising-dependencies-with-dependency-injection-in-net-part-1-setup/)

A dependency in the RegisterCustomerConsumer

Currently our RegisterCustomerConsumer only has an empty constructor. It has no external dependencies since it doesn't carry out much work anyway other than printing a couple of messages. Mind you the Console.WriteLine calls could be abstracted away into some interface like IMessageWriter so that callers can control where the messages are printed. So we can identify a dependency even in this simple class.

We'll do something different but the solution presented here is the same. We want to add a repository that the RegisterCustomerConsumer can delegate the save procedure to. Add a new C# class library called MyCompany.Domains with the following domain and a matching repository interface:

```
using System;

namespace MyCompany.Domains
{
    public class Customer
    {
        private readonly Guid _id;
        private readonly string _name;
        private readonly string _address;

        public Customer(Guid id, string name, string address)
        {
            _id = id;
            _name = name;
            _address = address;
        }
        public Guid Id { get { return _id; } }
        public string Address { get { return _address; } }
        public string Name { get { return _name; } }
        public DateTime RegisteredUtc { get; set; }
        public int Type { get; set; }
        public bool Preferred { get; set; }
        public decimal DefaultDiscount { get; set; }

    }
}
```

```
namespace MyCompany.Domains
{
    public interface ICustomerRepository
    {
        void Save(Customer customer);
    }
}
```

Add a project reference to MyCompany.Domains from MassTransit.Receiver.

Here's the updated RegisterCustomerConsumer class with an extended constructor:

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

namespace MassTransit.Receiver
{
    public class RegisterCustomerConsumer : IConsumer<IRegisterCustomer>
    {
        private readonly ICustomerRepository _customerRepository;

        public RegisterCustomerConsumer(ICustomerRepository customerRepository)
        {
            if (customerRepository == null) throw new ArgumentNullException("Customer repository");
            _customerRepository = customerRepository;
        }

        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);

            _customerRepository.Save(new Customer(newCustomer.Id, newCustomer.Name, newCustomer.Addre
            {
                DefaultDiscount = newCustomer.DefaultDiscount,
                Preferred = newCustomer.Preferred,
                RegisteredUtc = newCustomer.RegisteredUtc,
                Type = newCustomer.Type
            });

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

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

We add the dependency through the constructor and call the Save method in the Consume function, there's nothing magic in that. However, you'll see that Visual Studio shows a compilaton error:

'RegisterCustomerConsumer' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'TConsumer' in the generic type or method 'ConsumerExtensions.Consumer(IReceiveEndpointConfigurator, Action<IConsumerConfigurator>)'

Unfortunately we cannot just supply a concrete implementation of the interface like that.

The concrete repository

Before we continue let's just implement the ICustomerRepository interface. Add a new C# class library called MyCompany.Repository.Dummy with the following implementation:

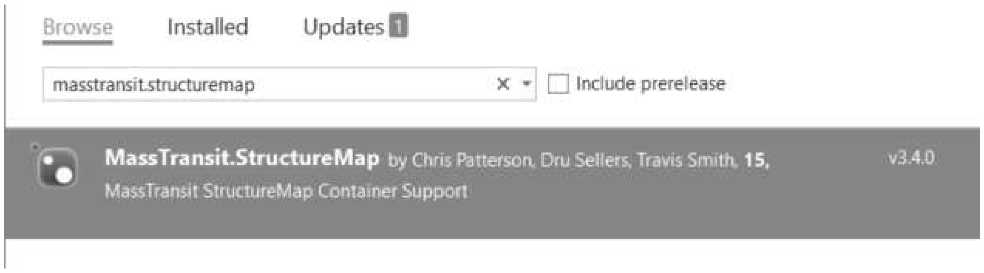
```
using MyCompany.Domains;
using System;

namespace MyCompany.Repository.Dummy
{
    public class CustomerRepository : ICustomerRepository
    {
        public void Save(Customer customer)
        {
            Console.WriteLine(string.Concat("The concrete customer repository was called for customer
        }
    }
}
```

StructureMap IoC

We'll have to use an inversion-of-control container here. MassTransit supports a number of IoC containers including Ninject, Unity, Autofac, Castle Windsor and StructureMap. I'm most familiar with StructureMap (http://structuremap.github.io/) so I'll use that in this example. I have used the code example on the relevant MassTransit documentation page (http://masstransit.readthedocs.io/en/master/usage/containers/structuremap.html) as a starting point, but the code code presented below is slightly different.

Add the following NuGet package to MassTransit.Receiver:



MassTransit.StructureMap by Chris Patterson, Dru Sellers, Travis Smith, 15, v3.4.0
MassTransit StructureMap Container Support

(https://dotnetcodr.files.wordpress.com/2016/09/structuremap-library-from-nuget-to-be-used-with-masstransit.png)

This library will add a number of extension methods to MassTransit. Here's the updated Program.cs of MassTransit.Receiver:

```
using MassTransit.RabbitMqTransport;
using MyCompany.Domains;
using MyCompany.Repository.Dummy;
using StructureMap;
using System;

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

        private static void RunMassTransitReceiverWithRabbit()
        {
            var container = new Container(conf =>
            {
                conf.For<ICustomerRepository>().Use<CustomerRepository>();
            });
            string whatDoIHave = container.WhatDoIHave();

            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>(container);
                });
            });

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

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

For the above code to compile we have to add a project reference to MyCompany.Repository.Dummy from MassTransit.Receiver.

At the start of RunMassTransitReceiverWithRabbit we configure the StrutureMap container. We declare which concrete implementations we want to use. We only have one so we tell StructureMap to construct a CustomerRepository if it finds a dependency on ICustomerRepository. The call to WhatDoIHave is not necessary for the code to work. I only wanted to show that as a tip. The function returns a string in tabular format showing which abstractions and concrete implementations it has in its registry. Take a look at the string and you'll find the ICustomerRepository/CustomerRepository pair.

Later on we send in the container to the Consumer method where we register RegisterCustomerConsumer. That's the same function we saw before but the overload where a StructureMap container could be passed in is new. Before the NuGet package was added there were 3 overloads of Consumer, now there's a fourth one.

That's it, you can run the demo the same way as in the previous post. You can set a breakpoint within RegisterCustomerConsumer to check whether ICustomerRepository repository was correctly resolved. Start the receivers first and then the publisher. You'll see that the system works like before and that ICustomerRepository was correctly resolved to a CustomerRepository. There will be an extra message in the MassTransit.Receiver command window:

"The concrete customer repository was called for customer Nice people LTD"

So we have successfully added a dependency to the RegisterCustomerConsumer class. To be honest I would have been happier to get this done without an IoC container since dependency injection should not depend on such a mechanism in a simple project such as a console application. However, I didn't find any other way to inject a dependency into the consumer class. If you know a better way then let everyone know in the comments section.

We'll continue in the next post (https://dotnetcodr.com/2016/09/28/messaging-through-a-service-bus-in-net-using-masstransit-part-5-failures/).

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.