Messaging through a service bus in .NET using MassTransit part 1: foundations

SEPTEMBER 8, 2016 3 COMMENTS (HTTPS://DOTNETCODR.COM/2016/09/08/MESSAGING-THROUGH-A-SERVICE-BUS-IN-NET-USING-MASSTRANSIT-PART-1-FOUNDATIONS/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/05/messaging-with-rabbitmq-and-net-review-part-11-various-other-topics/) we completed our updated series of using RabbitMq in .NET. We looked at a couple of miscellaneous topics such as getting a confirmation from RabbitMq whether a messages was actually queued or how a consumer can "not-acknowledge" a message. We also went through some other concepts like data serialisation where a link was provided to the relevant posts in the original series.

This post will start a new series but the topic will be very similar. We'll stay in the world of messaging. In fact we'll be building upon the previous series and take a step upwards in the abstraction ladder. We'll be looking at the basics of using a service bus in .NET using the open-source MassTransit project (http://masstransit-project.com/).

Pre-requisites

It is highly recommended that you have at least some basic knowledge of RabbitMq since we'll use it as the underlying messaging mechanism for the service bus. A service bus is best used with an underlying concrete messaging mechanism to wire messages between independent components and RabbitMq is an ideal and reliable choice for that. If you haven't used RabbitMq before then you can start going through the previous series here (https://dotnetcodr.com/2016/08/02/messaging-with-rabbitmq-and-net-review-part-1-foundations-and-terminology/). It's enough to go through the following posts in order to continue with this series:

- RabbitMq: foundations and terminology (https://dotnetcodr.com/2016/08/02/messaging-with-rabbitmq-and-net-review-part-1-foundations-and-terminology/)
- RabbitMq: installation and setup (https://dotnetcodr.com/2016/08/03/messaging-with-rabbitmq-and-net-review-part-2-installation-and-setup/)
- RabbitMq: the .NET client and some initial code (https://dotnetcodr.com/2016/08/05/messaging-with-rabbitmq-and-net-review-part-3-the-net-client-and-some-initial-code/)
- RabbitMq: one way messaging with a basic consumer (https://dotnetcodr.com/2016/08/08/messaging-with-rabbitmq-and-net-review-part-4-one-way-messaging-with-a-basic-consumer/)

If you're eager to continue with this course then you can leave the other posts in the RabbitMq series for later. However, make sure that have installed RabbitMq and can access its management UI on http://localhost:15672/ (http://localhost:15672/) and that you have a valid user other than the default guest. In the above posts we created a user called "accountant" and we'll use that in the code examples later on.

In fact we'll first look at using MassTransit on its own with the built-in in-memory message protocol so that we get acquainted with the most important concepts. However, that's not how service buses are used in practice so we'll eventually add the RabbitMq-related components to the code.

What's a service bus?

A service bus (https://en.wikipedia.org/wiki/Enterprise_service_bus) is a layer of abstraction above a messaging system like RabbitMq. Service buses come in a varying degree of abstraction and functionality where Enterprise Service Bus (ESB) is the most abstract and complex implementation. ESBs also tend to be quite expensive. We won't go there in this series, we'll be looking into a light-weight implementation instead.

A service bus helps us implement messaging between disparate applications in a distributed system without worrying about low level implementation details like exchanges, queues and bindings in RabbitMq. Ideally a service bus hides the underlying message broker from the client. If this is not the case then the clients may have bypass the service bus and directly call the services of the broker.

However, should you need to use the low level features of the message broker then the service bus library should have the necessary functions to do that. At the same time it should be relatively easy to switch the underlying message broker. A service bus that fulfils these criteria is called **platform-agnostic**. If you use RabbitMq directly in your code and want to switch to another message broker later on then you may have to do a lot of rewrite. With a well designed service bus the switch will be a lot smoother.

We can draw a parallel with a logging framework such as log4net (https://logging.apache.org/log4net/). Say you want to send your log messages to 3 targets: a file, Graylog (https://www.graylog.org/) and the standard output. You can surely build your own implementations using the available classes in .NET and the Graylog client library. However, it's more efficient to use a well-tested logging framework which already does that for you. All you need is a configuration file and the log4net client library to send the log messages to all your configured targets. You can forget about low level operations such as file-related calls and other resource intensive actions, log4net will take care of it for you. Similarly a service bus will shield the developer from the intricacies of exchanges, queues, bindings, acknowledgements, retries, concurrency etc. The developer can concentrate on sending and receiving the messages, the service bus will provide a lot of functionality out of the box.

There are a couple of disadvantages of using a service bus as well. Those may depend on the concrete service bus implementation though. If it hides a lot of functionality that the underlying message broker offers then the developers will need to bypass the bus. It might also put some overhead on top of the message broker operations. We can also mention that ESBs are quite complex and may make your system unnecessarily complex as well. ESBs generally warrant large systems with a lot of nodes. We are developers, we are easily carried away by a new exciting technology just because it looks great. However, if all you need is a messaging interface in a small system with 2-3 nodes you may be better off building a public API to those systems that can be called over HTTP.

MassTransit for .NET

MassTransit (http://masstransit.readthedocs.io/en/master/index.html) is an example of a service bus implementation. It is a "free, open source, lightweight message bus for creating distributed applications using the .NET framework." In their own words MassTransit is "not an Enterprise Service Bus (ESB)", it is more light-weight than a full-blown ESB like the Oracle Service Bus (http://www.oracle.com/technetwork/middleware/service-bus/overview/index.html) or the Mule ESB (https://www.mulesoft.com/platform/soa/mule-esb-open-source-esb). MassTransit is free even on production systems, you don't need to buy a licence. It provides a type system for .NET projects, i.e. we don't need to serialise and deserialise our C# objects using e.g. JSON, MassTransit will take of it for us. You can read about the history of this application here (http://masstransit.readthedocs.io/en/master/overview/backstory.html).

MassTransit has a client library that can be downloaded from NuGet. It doesn't require any special installation or a management UI. It can be readily used in a .NET project.

Another service bus implementation: NServiceBus

MassTransit is of course not the only service bus implementation out there. You may have come across NServiceBus (https://particular.net/nservicebus) which is also targeted at .NET projects. NServiceBus and MassTransit are similar products and there's a number of sites out there that compare and contrast the two. Here are a couple of useful links:

- Comparing nServiceBus and MassTransit: Do we still need .Net integration frameworks? (http://www.ben-morris.com/comparing-nservicebus-and-mass-transit-do-we-still-need-net-integration-frameworks/)
- NServiceBus versus MassTransit on StackOverflow (http://stackoverflow.com/questions/13647423/nservicebus-vs-masstransit)
- A fight between NServiceBus and MassTransit (http://looselycoupledlabs.com/2014/11/masstransit-versus-nservicebus-fight/)

We'll continue in the next post (https://dotnetcodr.com/2016/09/14/messaging-through-a-service-bus-in-net-using-masstransit-part-2-starting-with-some-code/) soon.

View the list of posts on Messaging here (https://dotnetcodr.com/messaging/).
FILED UNDER .NET, MESSAGING TAGGED WITH C#, MASSTRANSIT, MESSAGING

About Andras Nemes
I'm a .NET/Java developer living and working in Stockholm, Sweden.

3 Responses to *Messaging through a service bus in .NET using MassTransit part 1: foundations*

Koray says:
September 8, 2016 at 7:30 am

Fine start, looking forward reading next.

Reply
Saurin says:
October 25, 2017 at 7:18 am
More then awesome

Reply
Artur Karbone (@ArturKarbone) says:
May 25, 2018 at 6:20 pm
I like the analogy with log4net and its providers

Reply