Messaging with RabbitMQ and .NET review part 3: the .NET client and some initial code

AUGUST 5, 2016 6 COMMENTS (HTTPS://DOTNETCODR.COM/2016/08/05/MESSAGING-WITH-RABBITMQ-AND-NET-REVIEW-PART-3-THE-NET-CLIENT-AND-SOME-INITIAL-CODE/#COMMENTS)

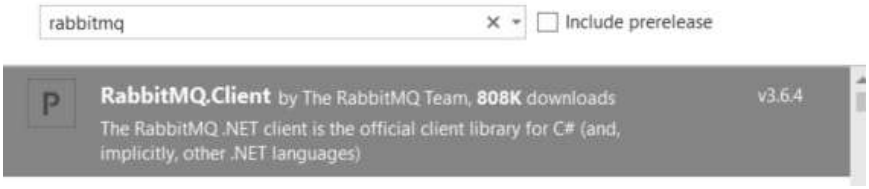
Introduction

In the previous post (https://dotnetcodr.com/2016/08/03/messaging-with-rabbitmq-and-net-review-part-2-installation-and-setup/) we installed the RabbitMq service on Windows. I think you'll agree that it wasn't a very complicated process. We then logged into the management GUI using the default "guest" administrator user. We finally looked at how to create users and virtual hosts. We said that a virtual host was a container or namespace to delimit groups of resources within RabbitMq, such as "sales" or "accounting". We also created a new user called "accountant".

In this post we'll start working with RabbitMq in Visual Studio. We'll in particular start exploring the RabbitMq .NET client library.

The RabbitMq .NET driver and some initial code

Open Visual Studio 2015 or an earlier version and create a new Console application. I'll call my instance RabbitMqNetTests. Import the following NuGet package:



(https://dotnetcodr.files.wordpress.com/2016/08/rabbitmq-nuget-package-for-net.png)

Add the following using statement to Program.cs:

```
1 using RabbitMQ.Client;
```

Let's create a connection to the RabbitMQ server in Main. The ConnectionFactory object will help us build an IConnection object which represents a connection to the RabbitMq message broker. There are at least 2 ways to build the connection. One is to provide a connection URI. The following will connect to the RabbitMq server with the user "accountant", password "accountant", port 5672, virtual host "accounting" using the connection string kind of syntax:

```
1 ConnectionFactory connectionFactory = new ConnectionFactory();
2 connectionFactory.Uri = "amqp://accountant:accountant@localhost:5672/accounting";
3 IConnection connection = connectionFactory.CreateConnection();
4 Console.WriteLine(string.Concat( "Connection open: ", connection.IsOpen));
```

Note that although the management UI opens on port 15672 the correct port number for access in code is 5672 for regular connections. If you set up TLS for your RabbitMq instance then the correct port will be 5671.

You can test the above code in the Main method of the console app. If the code executes without exceptions and you see "Connection open: True" in the console window then the connection was made successfully. If you change the connection string to a bad port number, e.g. 5670 then you'll get the following exception:

An unhandled exception of type 'RabbitMQ.Client.Exceptions.BrokerUnreachableException' occurred in RabbitMQ.Client.dll
Additional information: None of the specified endpoints were reachable

This message usually indicates a problem with the connection string.

Another way of connecting to the message broker is by building up the connection factory object bit by bit using its public properties as follows:

```
ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.Port = 5672;
connectionFactory.HostName = "localhost";
connectionFactory.UserName = "accountant";
connectionFactory.Password = "accountant";
connectionFactory.VirtualHost = "accounting";
IConnection connection = connectionFactory.CreateConnection();
```

Remember that the IConnection object is thread safe and can be shared among multiple applications.

The channel

We also mentioned that an application communicates with RabbitMq through a channel which exists within a connection. A connection can have multiple channels to enable parallel communication of multiple threads. Each thread should have its own channel.

For some reason the channel is called IModel in the .NET client. An IModel represents a channel to the AMQP server:

```
IModel model = connection.CreateModel();
```

From IModel we can access methods to send and receive messages and much more. Here's how to close a channel:

```
channel.Close();
```

There's also a boolean property to check whether the channel has been closed:

```
bool channelIsClosed = channel.IsClosed;
```

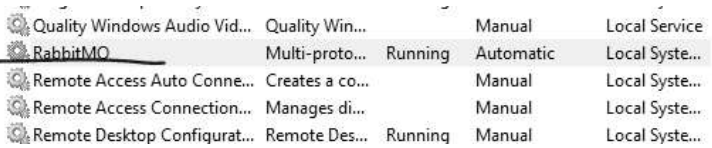
Durability basics

There are two types of queues, exchanges and messages from a persistence point of view:

- Durable: messages and other resources are saved to disk so they are available even after a server restart. There's some overhead incurred while reading and saving messages. If durability is set to true then the resource will be persisted to disk and not only exist in memory. You'll probably want to have the durable parameter set to true in a real life messaging scenario
- Non-durable: the resources are persisted in memory only. They disappear after a server restart but offer a faster service

Keep these advantages and disadvantages in mind when you're deciding which persistence strategy to go for.

The RabbitMq service is run by a Windows service visible in the Services window:



(https://dotnetcodr.files.wordpress.com/2016/08/rabbitmq-windows-service.png)

Later on you can always simulate a RabbitMq server restart by right-clicking on the above node and selecting the Restart option. All non-durable resources will be wiped out from memory but the durable ones will survive, including durable messages that have not yet been acknowledged by any client at the time of the server crash.

Creating queues, exchanges and bindings at runtime

The IModel object will let us create all the necessary resources for exchanging messages. The ExchangeDeclare method will create an exchange. The following is an example for the exchange type direct. The exchange name is the first parameter:

```
channel.ExchangeDeclare("my.first.exchange", ExchangeType.Direct);
```

The above is the most basic overload of ExchangeDeclare. Here is a the longest overload:

```
channel.ExchangeDeclare("my.first.exchange", ExchangeType.Direct, true, false, null);
```

The third parameter, i.e. “true” sets durability to true. Then the “false” stands for the auto-delete option. Setting auto-delete to true for exchanges and queues means that they are automatically deleted as soon as they are no longer used, i.e. there’s no channel using them. The last parameter is a dictionary object where we can pass in some further options. We won’t worry about that right now so we set it to null.

Next we create a queue using the QueueDeclare function:

```
channel.QueueDeclare("my.first.queue", true, false, false, null);
```

We have the following parameters to the function:

- The queue name
- Whether it is durable
- Whether it is exclusive, which means whether the queue is exclusively used for the connection
- Whether it should be auto-deleted
- The same dictionary object with custom options as in ExchangeDeclare

The last piece that’s missing is the binding. We use the QueueBind method for that:

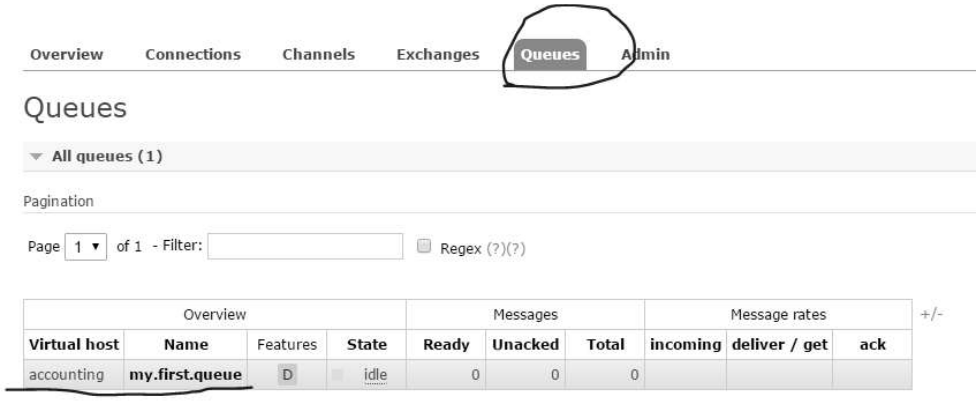
```
channel.QueueBind("my.first.queue", "my.first.exchange", "");
```

Here we have the queue name, the exchange name and then the routing key. We set the routing key to an empty string to indicate that we don’t need any.

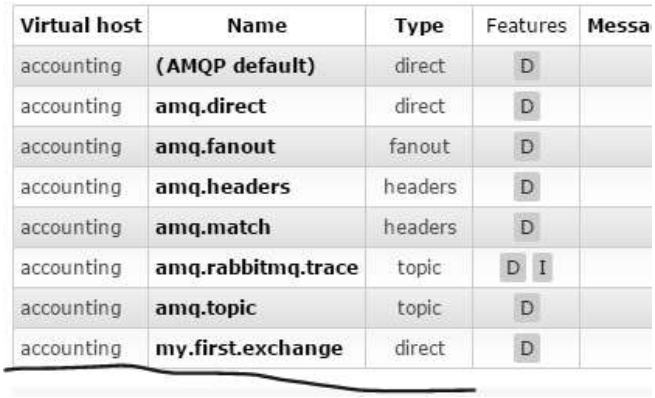
Alright, let’s put all of this together in the Main method of the console application:

```
static void Main(string[] args)
{
    ConnectionFactory connectionFactory = new ConnectionFactory();

    connectionFactory.Port = 5672;
    connectionFactory.HostName = "localhost";
    connectionFactory.UserName = "accountant";
    connectionFactory.Password = "accountant";
    connectionFactory.VirtualHost = "accounting";

    IConnection connection = connectionFactory.CreateConnection();
    IModel channel = connection.CreateModel();
    Console.WriteLine(string.Concat( "Connection open: ", connection.IsOpen));

    channel.ExchangeDeclare("my.first.exchange", ExchangeType.Direct, true, false, null);
    channel.QueueDeclare("my.first.queue", true, false, false, null);
    channel.QueueBind("my.first.queue", "my.first.exchange", "");

    channel.Close();
        connection.Close();
    Console.WriteLine(string.Concat("Channel is closed: ", channel.IsClosed));

    Console.WriteLine("Main done...");
    Console.ReadKey();
}
```

Run the above code. You shouldn’t get any exceptions thrown. Let’s check whether the resources are visible in the management UI:



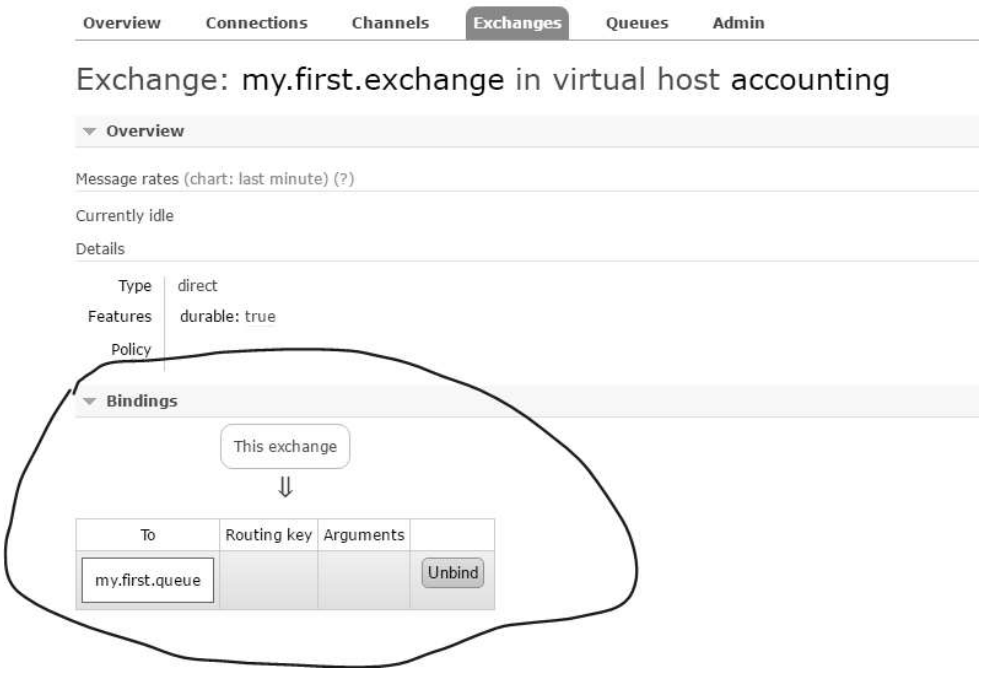
(https://dotnetcodr.files.wordpress.com/2016/08/first-rabbitmq-queue-created-in-code.png)



Virtual host	Name	Type	Features	Messa
accounting	(AMQP default)	direct	D	
accounting	amq.direct	direct	D	
accounting	amq.fanout	fanout	D	
accounting	amq.headers	headers	D	
accounting	amq.match	headers	D	
accounting	amq.rabbitmq.trace	topic	D I	
accounting	amq.topic	topic	D	
accounting	my.first.exchange	direct	D	

(https://dotnetcodr.files.wordpress.com/2016/08/first-rabbitmq-exchange-created-in-code.png)

If you click on the exchange name you can also view the existing bindings:



(https://dotnetcodr.files.wordpress.com/2016/08/rabbitmq-binding-created-in-code.png)

The code seems to have worked well. The great thing is that you can run this code as often you want. If the declared resource already exists then the code will not do anything, i.e. you won't get an exception if you declare the same queue twice or more often.

You can also restart the RabbitMq service in the Services window to simulate a server restart. The resources should still be there.

Publishing a message

The last bit we'll look at in this post is how to publish a message to our newly created queue. Add the following code after the QueueBind method:

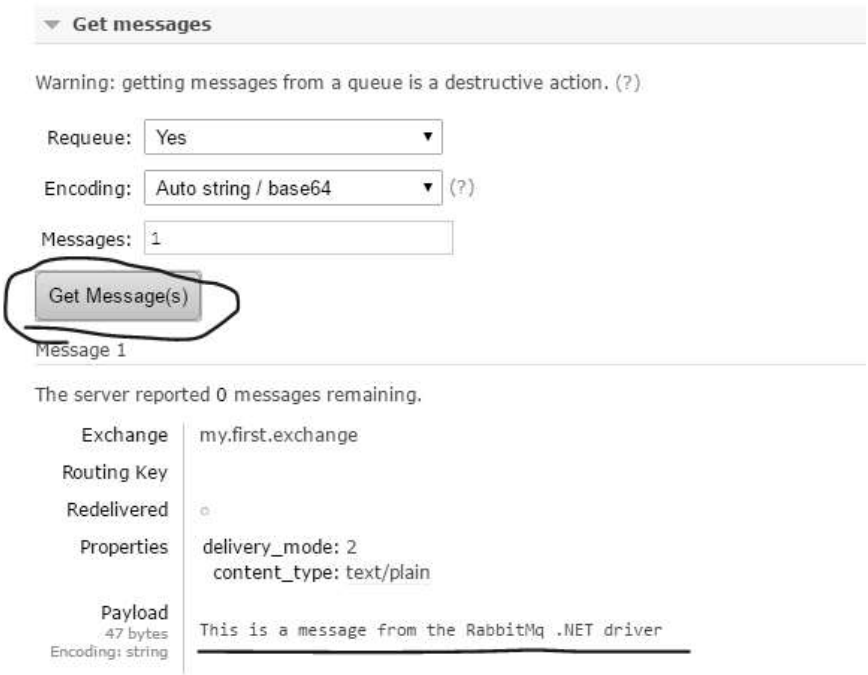
```
1  IBasicProperties properties = channel.CreateBasicProperties();
2  properties.Persistent = true;
3  properties.ContentType = "text/plain";
4  PublicationAddress address = new PublicationAddress(ExchangeType.Direct, "my.first.exchange", "");
5  channel.BasicPublish(address, properties, Encoding.UTF8.GetBytes("This is a message from the RabbitMq
```

IBasicProperties lets us set a wide range of properties for the message. Here we only use two of them. We want the message to be durable, i.e. persistent. We also declare the MIME type. We then build a PublicationAddress object with the exchange name, the exchange type and the empty routing key. We finally publish a message which must first be transformed into a byte array.

Run the above code. If everything went as expected then there should be a message in the queue:

(https://dotnetcodr.files.wordpress.com/2016/08/first-message-registered-in-rabbitmq-queue.png)

Before we do anything else we can again simulate a server crash and restart the RabbitMq service like we did above. The message should still be there. We can retrieve the message in the UI in the "Get messages" section of the queue. Click on the queue name and get the message by pressing the get messages button:



(https://dotnetcodr.files.wordpress.com/2016/08/message-retrieved-from-the-rabbitmq-queue.png)

That's enough for now. We'll continue our exploration in the next post (https://dotnetcodr.com/2016/08/08/messaging-with-rabbitmq-and-net-review-part-4-one-way-messaging-with-a-basic-consumer/) by looking at how to get messages in code.

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

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

6 Responses to *Messaging with RabbitMQ and .NET review part 3: the .NET client and some initial code*

ramessesx says:
August 7, 2016 at 2:13 pm
Hi Andras, thanks again for posting. I got everything to work with the '/' virtual host, but not with the new Vitual Host (accounting), do you know of any reasons why this is ?

Reply

Andras Nemes says:
August 7, 2016 at 6:00 pm
Hello, what kind of exception do you get? Does the virtual host really exist? //Andras

Reply

ramessesx says:
August 8, 2016 at 9:50 am
Hi Andras. I was not getting any exceptions as such, I simply was not seeing the newly created exchange or queue on the RabbitMQ Managment console, but I have just figured out what the problem was => Permissions.

I was signed in as 'guest', not 'accountant', and as such it would not allow me to view the exchanges or queues for the 'accounting' virtual host. After setting permission for 'guest' on the 'accounting' host, I was able to view everything.

Sorry for troubling you with such a 'vague' question, and thanks for taking the time to respond.

Love the blog, look forward to more …

Andras Nemes says:
August 8, 2016 at 4:14 pm
Hello, it's ok, don't worry, I'm glad you find this blog useful. //Andras

ict22 says:
August 15, 2016 at 6:42 pm
Hi Andras, many thanks for the update. Great and very useful blog!

Reply

Andrei Timofte says:
July 2, 2019 at 11:54 am
Great series! Thank you for explaining in such detail!

Reply