Messaging with RabbitMQ and .NET review part 6: the fanout exchange type

AUGUST 15, 2016 4 COMMENTS (HTTPS://DOTNETCODR.COM/2016/08/15/MESSAGING-WITH-RABBITMQ-AND-NET-REVIEW-PART-6-THE-FANOUT-EXCHANGE-TYPE/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/08/10/messaging-with-rabbitmq-and-net-review-part-5-one-way-messaging-with-an-event-based-consumer/) we looked at an alternative way to consume messages from a queue in RabbitMq. In particular we discussed the usage of the EventingBasicConsumer which is an event and delegate based alternative to the DefaultBasicConsumer class. The outcome is the same in both cases, i.e. the consumer monitors the assigned queue and pulls messages from it.

In this post we'll discuss how to work with the fanout exchange type.

Fanout exchange and the publish-subscribe MEP

So far in this series we've mentioned two message exchange patterns (MEPs): one way messaging and worker queues. In the publish-subscribe MEP a message is sent to an exchange and the exchange distributes the message to all queues bound to it. Each queue will have its listener to process the message. This sounds like the fanout variant of the exchange types we briefly discussed earlier. We'll work with the test console application we started before in Visual Studio. Currently there are two projects: one message sender and one receiver. We'll reuse the sender, there's no point in setting up a new project just for that, the code is so similar for that part.

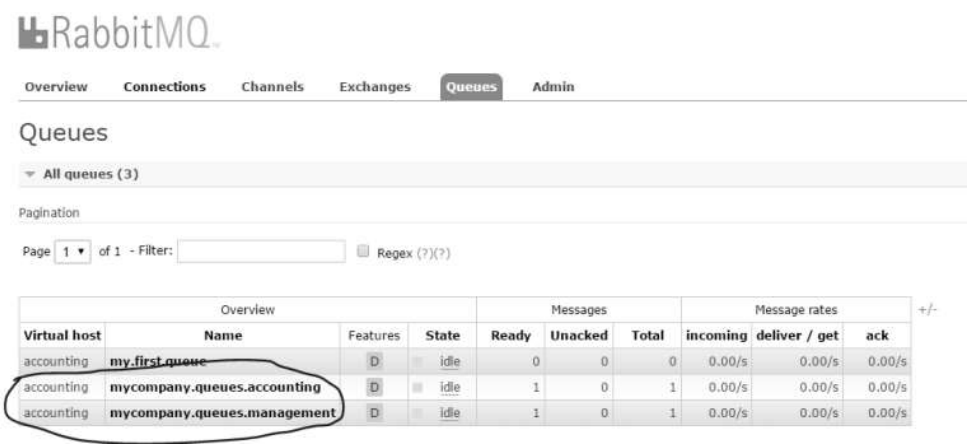
At this point we have some code that sets up an exchange of type direct in the Main method of the sender app. You can cut and paste all that code into a private method so that you have it as reference. You can call it SetUpDirectExchange or something like that.

We'll first set up the fanout exchange and two queues. We'll simulate the scenario where two listeners are polling messages from two queues but both queues are fed by the same exchange. The two listeners will be the accounting and management departments of the company. We'll also send a message to the exchange in the same code snippet:

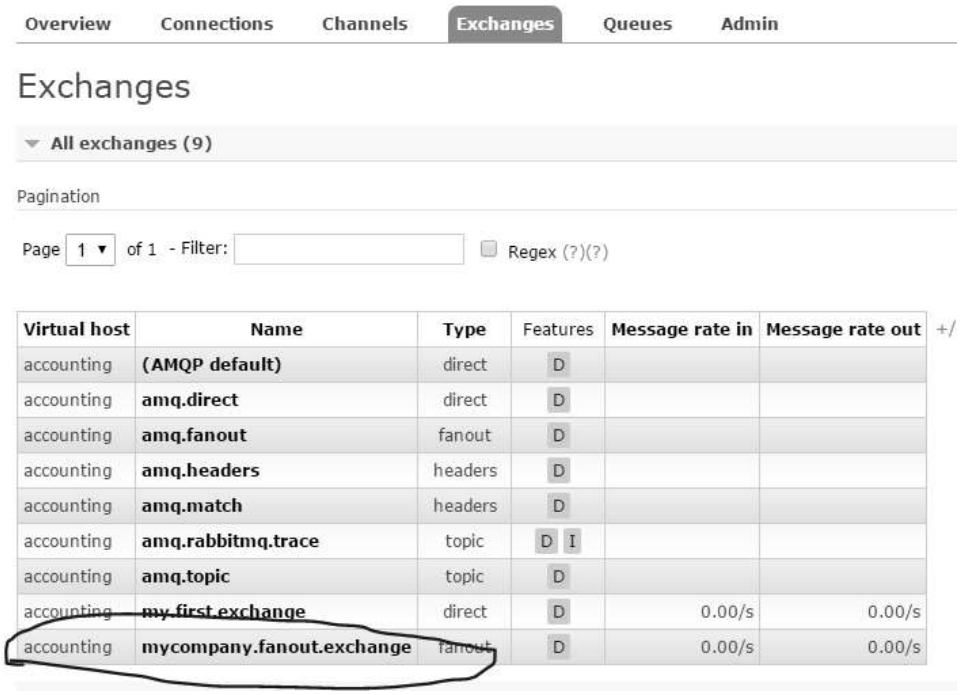
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RabbitMQ.Client;

namespace RabbitMqNetTests
{
    class Program
    {
        static void Main(string[] args)
        {
            SetUpFanoutExchange();
        }

        private static void SetUpFanoutExchange()
        {
            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();

            channel.ExchangeDeclare("mycompany.fanout.exchange", ExchangeType.Fanout, true, false, nu
            channel.QueueDeclare("mycompany.queues.accounting", true, false, false, null);
            channel.QueueDeclare("mycompany.queues.management", true, false, false, null);
            channel.QueueBind("mycompany.queues.accounting", "mycompany.fanout.exchange", "");
            channel.QueueBind("mycompany.queues.management", "mycompany.fanout.exchange", "");

            IBasicProperties properties = channel.CreateBasicProperties();
            properties.Persistent = true;
            properties.ContentType = "text/plain";
            PublicationAddress address = new PublicationAddress(ExchangeType.Fanout, "mycompany.fanou
            channel.BasicPublish(address, properties, Encoding.UTF8.GetBytes("A new huge order has ju

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

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

        private static void SetUpDirectExchange()
        {
            //code ignored
        }
    }
}
```

You'll see that the code is virtually identical to what we had for the one-way direct message exchange type. The only major difference is the exchange type set in the ExchangeType class: it's Fanout instead of Direct. Another difference is that we bind two different queues to the same exchange. Otherwise the arguments to the various functions is the same as before.

Run the code in Visual Studio, it should succeed normally. The new resources will be visible in the management GUI:

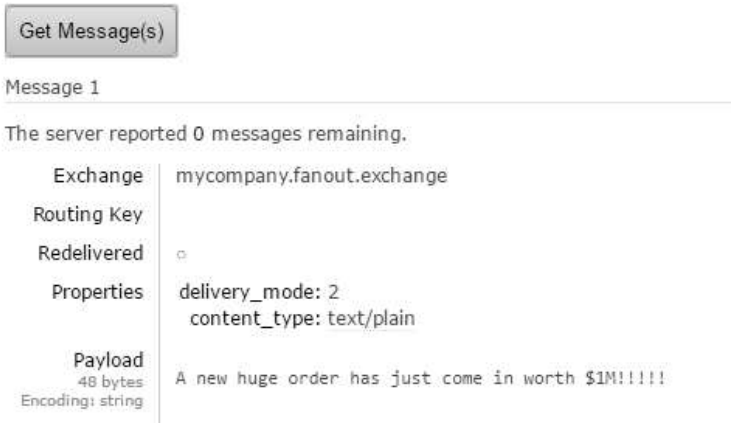


(https://dotnetcodr.files.wordpress.com/2016/08/fanout-queues-visible-in-rabbitmq-management-gui.png)



(https://dotnetcodr.files.wordpress.com/2016/08/fanout-exchange-type-visible-in-rabbitmq-management-console.png)

Even the message is visible on both queues, I'll just show one of them here:



(https://dotnetcodr.files.wordpress.com/2016/08/fanout-message-visible-on-rabbitmq-queue-in-the-management-console.png)

The consumers

In this section we'll set up the two consumers. You'll see that the implementation is again almost identical to what we saw in the case of direct messages. I'll go for the solution based on the EventingBasicConsumer class so that we don't need to create another class for the consumer. However, feel free to derive from DefaultBasicConsumer, it's equally good.

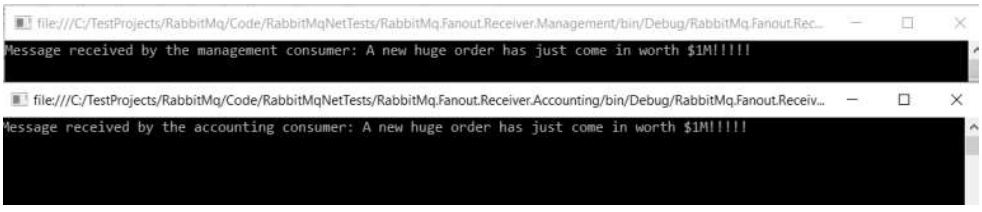
Add two new console projects to the demo solution: RabbitMq.Fanout.Receiver.Accounting and RabbitMq.Fanout.Receiver.Management. Add the RabbitMq client API from NuGet to both projects. Here's the code for Program.cs of the accounting consumer:

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RabbitMq.Fanout.Receiver.Accounting
{
    class Program
    {
        static void Main(string[] args)
        {
            ReceiveFanoutMessages();
        }

        private static void ReceiveFanoutMessages()
        {
            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();
            channel.BasicQos(0, 1, false);
            EventingBasicConsumer eventingBasicConsumer = new EventingBasicConsumer(channel);

            eventingBasicConsumer.Received += (sender, basicDeliveryEventArgs) =>
            {
                IBasicProperties basicProperties = basicDeliveryEventArgs.BasicProperties;

                Debug.WriteLine(string.Concat("Message received from the exchange ", basicDeliveryEve
                Debug.WriteLine(string.Concat("Content type: ", basicProperties.ContentType));
                Debug.WriteLine(string.Concat("Consumer tag: ", basicDeliveryEventArgs.ConsumerTag));
                Debug.WriteLine(string.Concat("Delivery tag: ", basicDeliveryEventArgs.DeliveryTag));
                string message = Encoding.UTF8.GetString(basicDeliveryEventArgs.Body);
                Debug.WriteLine(string.Concat("Message: ", Encoding.UTF8.GetString(basicDeliveryEvent
                Console.WriteLine(string.Concat("Message received by the accounting consumer: ", mess
                channel.BasicAck(basicDeliveryEventArgs.DeliveryTag, false);
            };

            channel.BasicConsume("mycompany.queues.accounting", false, eventingBasicConsumer);
        }
    }
}
```

This should be easy to follow based on what we saw previously. We connect the consumer to the accounting queue. The management consumer will have the same code but the queue name will be adjusted accordingly:

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RabbitMq.Fanout.Receiver.Management
{
    class Program
    {
        static void Main(string[] args)
        {
            ReceiveFanoutMessages();
        }

        private static void ReceiveFanoutMessages()
        {
            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();
            channel.BasicQos(0, 1, false);
            EventingBasicConsumer eventingBasicConsumer = new EventingBasicConsumer(channel);

            eventingBasicConsumer.Received += (sender, basicDeliveryEventArgs) =>
            {
                IBasicProperties basicProperties = basicDeliveryEventArgs.BasicProperties;

                Debug.WriteLine(string.Concat("Message received from the exchange ", basicDeliveryEve
                Debug.WriteLine(string.Concat("Content type: ", basicProperties.ContentType));
                Debug.WriteLine(string.Concat("Consumer tag: ", basicDeliveryEventArgs.ConsumerTag));
                Debug.WriteLine(string.Concat("Delivery tag: ", basicDeliveryEventArgs.DeliveryTag));
                string message = Encoding.UTF8.GetString(basicDeliveryEventArgs.Body);
                Debug.WriteLine(string.Concat("Message: ", Encoding.UTF8.GetString(basicDeliveryEvent
                Console.WriteLine(string.Concat("Message received by the management consumer: ", mess
                channel.BasicAck(basicDeliveryEventArgs.DeliveryTag, false);
            };

            channel.BasicConsume("mycompany.queues.management", false, eventingBasicConsumer);
        }
    }
}
```

You can start both consumers in Visual Studio using the same technique we used before: right-click, Debug, Start new instance. You should see that both consumers have pulled the same message:



(https://dotnetcodr.files.wordpress.com/2016/08/both-fanout-queue-consumers-receive-the-message-from-the-rabbitmq-exchange.png)

That was a very basic implementation of fanout messaging in RabbitMq. In the next post (https://dotnetcodr.com/2016/08/18/messaging-with-rabbitmq-and-net-review-part-7-two-way-messaging/) we'll look at two-way messaging.

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.

4 Responses to *Messaging with RabbitMQ and .NET review part 6: the fanout exchange type*

ramessesx says:
August 15, 2016 at 9:51 am
Thanks Andras

Reply

ict22 says:
August 16, 2016 at 12:58 pm
Thank you for the updated version. Very informative.

Reply

Abu Sazzad says:
April 10, 2017 at 1:45 pm
Hi Andras,

Thanks for this article. However if I try to run the consumer application before running publisher application, I'm getting the following exception:
RabbitMQ.Client.Exceptions.OperationInterruptedException: 'The AMQP operation was interrupted: AMQP close-reason, initiated by Peer, code=404, text="NOT_FOUND – no queue 'mycompany.queues.accounting' in vhost '/'", classId=60, methodId=20, cause='

So is it by design where publisher application needs to be run first so that the queue is created, then consumer application would eventually found the queue and run as usual.

Please let me know your thoughts on this.

Thanks again.

Reply

ahmet says:
November 17, 2019 at 11:16 am
when i delete below line of code, its still working.
channel.ExchangeDeclare("mycompany.fanout.exchange", ExchangeType.Fanout, true, false, null);

Reply