RabbitMQ for Windows: Direct Exchanges

2 April, 2012. It was a Monday.

Posts In This Series

This is the fifth installment to the series: RabbitMQ for Windows. In the last installment, we took a look at the four exchange types provided by RabbitMQ: Direct, Fanout, Topic, and Headers. In this installment we'll walk through an example which uses a direct exchange type directly and we'll take a look at the push API.

In the Hello World example from the second installment of the series, we used a direct exchange type implicitly by taking advantage of the automatic binding of queues to the default exchange using the queue name as the routing key. The example we'll work through this time will be similar, but we'll declare and bind to the exchange explicitly.

This time our example will be a distributed logging application. We'll create a Producer console application which publishes a logging message for some noteworthy action and a Consumer console application which displays the message to the console.

Beginning with our Producer app, we'll start by establishing a connection using the default settings, create the connection, and create a channel:

```
using RabbitMQ.Client;

namespace Producer
{
  class Program
  {
    static void Main(string[] args)
    {
      var connectionFactory = new ConnectionFactory();
      IConnection connection = connectionFactory.CreateConnection();
      IModel channel = connection.CreateModel();
    }
  }
}
```

Next, we need to declare the exchange we'll be publishing our message to. We need to give our exchange a name in order to reference it later, so let's use "direct-exchange-example":

```
channel.ExchangeDeclare("direct-exchange-example", ExchangeType.Direct);
```

The second parameter indicates the exchange type. For the official RabbitMQ .Net client, this is just a simple string containing one of the values: direct, fanout, topic, or headers. The type RabbitMQ.Client.ExchangeType defines each of the exchange types as a constant for convenience.

Next, let's call some method which might produce a value worthy of interest. We'll call the method DoSomethingInteresting() and have it return a string value:

```
string value = DoSomethingInteresting();
```

For the return value, the implementation of DoSomethingInteresting() can just return the string value of a new Guid:

```
static string DoSomethingInteresting()
{
  return Guid.NewGuid().ToString();
}
```

Next, let's use the returned value to create a log message containing a severity level of Information:

```
string logMessage = string.Format("{0}: {1}", TraceEventType.Information, value);
```

Next, we need to convert our log message to a byte array and publish the message to our new exchange:

```


  byte[] message = Encoding.UTF8.GetBytes(logMessage);
```

```
  channel.BasicPublish("direct-exchange-example", "", null, message);
```

Here, we use an empty string as our routing key and null for our message properties.

We end our Producer by closing the channel and connection:

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

Here's the full listing:

```
using System;
using System.Diagnostics;
using System.Text;
using System.Threading;
using RabbitMQ.Client;

namespace Producer
{
  class Program
  {
    static void Main(string[] args)
    {
      Thread.Sleep(1000);
      var connectionFactory = new ConnectionFactory();
      IConnection connection = connectionFactory.CreateConnection();
      IModel channel = connection.CreateModel();

      channel.ExchangeDeclare("direct-exchange-example", ExchangeType.Direct);
```

```
      string value = DoSomethingInteresting();
      string logMessage = string.Format("{0}: {1}", TraceEventType.Information, value);

      byte[] message = Encoding.UTF8.GetBytes(logMessage);
      channel.BasicPublish("direct-exchange-example", "", null, message);

      channel.Close();
      connection.Close();
    }

    static string DoSomethingInteresting()
    {
      return Guid.NewGuid().ToString();
    }
  }
}
```

Note that our logging example's Producer differs from our Hello World's Producer in that we didn't declare a queue this time. In our Hello World example, we needed to run our Producer before the Consumer since the Consumer simply retrieved a single message and exited. Had we published to the default exchange without declaring the queue first, our message would simply have been discarded by the server before the Consumer had an opportunity to declare and bind the queue.

Next, we'll create our Consumer which starts the same way as our Producer code:

```
using RabbitMQ.Client;

namespace Consumer
{
  class Program
  {
```

```
    static void Main(string[] args)
    {
      var connectionFactory = new ConnectionFactory();
      IConnection connection = connectionFactory.CreateConnection();
      IModel channel = connection.CreateModel();

      channel.ExchangeDeclare("direct-exchange-example", ExchangeType.Direct);
    }
  }
}
```

Next, we need to declare a queue to bind to our exchange. Let's name our queue "logs":

```
channel.QueueDeclare("logs", false, false, true, null);
```

To associate our logs queue with our exchange, we use the QueueBind() method providing the name of the queue, the name of the exchange, and the binding key to filter messages on:

```
channel.QueueBind("logs", "direct-exchange-example", "");
```

At this point we could consume messages using the pull API method BasicGet() as we did in the Hello World example, but this time we'll use the push API. To have messages pushed to us rather than us pulling messages, we first need to declare a consumer:

```
var consumer = new QueueingBasicConsumer(channel);
```

To start pushing messages to our consumer, we call the channel's BasicConsume() method and tell it which consumer to start pushing messages to:

```
channel.BasicConsume(“logs”, true, consumer);
```

Here, we specify the queue to consume messages from, a boolean flag instructing messages to be auto-acknowledged (see discussion in the Getting the Message section of Hello World Review), and the consumer to push the messages to.

Now, any messages placed on the queue will automatically be retrieved and placed in a local in-memory queue. To dequeue a message from the local queue, we call the Dequeue() method on the consumer's Queue property:

```
var eventArgs = (BasicDeliverEventArgs)consumer.Queue.Dequeue();
```

This method call blocks until a message is available to be dequeued, or until an EndOfStreamException is thrown indicating that the consumer was cancelled, the channel was closed, or the connection otherwise was terminated.

Once the Dequeue() method returns, the BasicDeliverEventArgs contains the bytes published from the Producer in the Body property, so we can convert this value back into a string and print it to the console:

```
var message = Encoding.UTF8.GetString(eventArgs.Body);
Console.WriteLine(message);
```

We end our Consumer by closing the channel and connection:

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

Here's the full listing:

```
using System;
using System.Text;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

namespace Consumer
{
```

```
  class Program
  {
    static void Main(string[] args)
    {
      var connectionFactory = new ConnectionFactory();
      IConnection connection = connectionFactory.CreateConnection();
      IModel channel = connection.CreateModel();

      channel.ExchangeDeclare("direct-exchange-example", ExchangeType.Direct);
      channel.QueueDeclare("logs", false, false, true, null);
      channel.QueueBind("logs", "direct-exchange-example", "");

      var consumer = new QueueingBasicConsumer(channel);
      channel.BasicConsume("logs", true, consumer);

      var eventArgs = (BasicDeliverEventArgs) consumer.Queue.Dequeue();

      string message = Encoding.UTF8.GetString(eventArgs.Body);
      Console.WriteLine(message);

      channel.Close();
      connection.Close();
      Console.ReadLine();
    }
  }
}
```

If we run the resulting Consumer.exe at this point, it will block until a message is routed to the queue. Running the Producer.exe from another shell produces a message on the consumer console similar to the following:

```
Information: 610fe447-bf31-41d2-ae29-414b2d00087b
```

Note: For a convenient way to execute both the Consumer and Producer from within Visual Studio, go to the solution properties and choose “Set StartUp Projects …”. Select the “Multiple startup projects:” option and set both the Consumer and Producer to the Action: Start. Use the arrows to the right of the projects to ensure the Consumer is started before the Producer. In some cases, this can still result in the Producer publishing the message before the Consumer has time to declare and bind the queue, so putting a Thread.Sleep(1000) at the start of your Producer should ensure things happen in the required order. After this, you can run your examples by using Ctrl+F5 (which automatically prompts to exit).

That concludes our direct exchange example. Next time, we’ll take a look at the Fanout exchange type.

← RabbitMQ for Windows: Exchange Types

RabbitMQ for Windows: Fanout Exchanges →

publishing directly to a queue. In reality, producers always publish to an exchange. Queues can be declared by either the producer or the consumer, but queues are generally thought of as being associated with the consumer rather than the producer.

^ | v • Share ›

DavidS ➚ derekgreer • 8 years ago

Hi Derek, thank you for getting back to me and thank you for the clarification as well.

^ | v • Share ›

DavidS ➚ DavidS • 8 years ago

OK got it. It's done within the QueueBind

^ | v • Share ›

DavidS • 8 years ago

Hi, one of the things I've noticed is that when you run the HelloWorld example, the "hello-world-queue" remains active even after the application has terminated. However, that is not the case for the Logging example whereby as soon as I close the Consumer application, the "logs" queue disappears. Can you shed some light over why this is the case?

^ | v • Share ›

DavidS ➚ DavidS • 8 years ago

Ah, I've found what it could possibly be. The fact is that we have got

channel.QueueDeclare("logs", false, false, true, null);

i.e. auto-delete set and the fact that we are using the method BasicConsume means that the queue is autodeleted.

^ | v • Share ›

Recent Author Posts

- Hello, React! - A Beginner's Setup Tutorial
- Exploring TypeScript
- Git on Windows: Whence Cometh Configuration
- Separation of Concerns: Application Builds & Continuous Integration
- Survey of Entity Framework Unit of Work Patterns
- Introducing NUnit.Specifications
- Being Agile
- Expected Objects Custom Comparisons
- RabbitBus: An Example
- Adventures in Debugging: The NHibernate 'don't flush the Session' Error

Recent Site Posts

- Domain-Driven Refactoring: Defactoring and Pushing Behavior Down
- Domain-Driven Refactoring: Extracting Domain Services
- Domain-Driven Refactoring: Long Methods
- Domain-Driven Refactoring: Procedural Beginnings
- Domain-Driven Refactoring: Intro
- Local Development with Azure Service Bus
- Taming the WSL 2 Resource Monster
- Crossing the Generics Divide
- OpenTelemetry 1.0 Extensions Released
- Choosing a ServiceLifetime

Authors

- Andrew Siemer
- Chad Myers
- Chris Missal
- Chris Patterson
- Derek Greer
- Derik Whittaker
- Eric Anderson
- Eric Hexter
- Gabriel Schenker
- Gregory Long
- Hugo Bonacci
- James Gregory
- Jimmy Bogard
- John Teague
- Josh Arnold
- Joshua Flanagan
- Joshua Lockwood
- Keith Dahlby
- Matt Hinze
- Patrick Lioi
- Rod Paddock
- Ryan Rauh
- Ryan Svihla
- Scott Densmore
- Sean Biefeld
- Sean Chambers
- Sharon Cichelli
- Steve Donie