RabbitMQ in .NET: data serialisation I

JUNE 5, 2014 3 COMMENTS (HTTPS://DOTNETCODR.COM/2014/06/05/RABBITMQ-IN-NET-DATA-SERIALISATION/#COMMENTS)

Introduction

We went through the basic messaging concepts in RabbitMQ in a previous series. You'll find the links to all installment on this page (https://dotnetcodr.com/messaging/). In this new series we'll continue our discussion of messaging concepts in RabbitMQ. If you're entirely new to RabbitMq then you should at least skim through the foundations as I won't provide any detailed description of the code that was covered before.

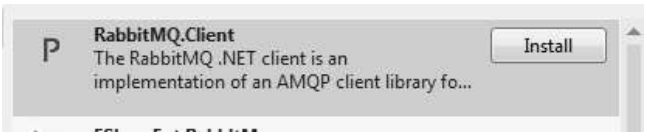
So far we've kept our messages simple in order to concentrate on the key concepts: we only sent simple text messages to RabbitMQ. However, in reality we normally send objects with properties and not only text. The object needs to be serialised into a byte array so that it can be included in the message body. On the receiving end the serialised object needs to be deserialised.

We're going to look at different ways of achieving this goal.

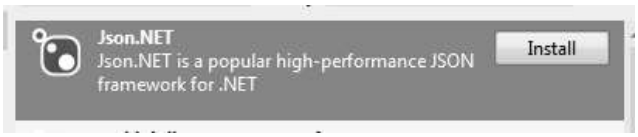
Most of the posts on RabbitMQ on this blog are based on the work of RabbitMQ guru Michael Stephenson (http://geekswithblogs.net/michaelstephenson/Default.aspx).

Demo: JSON serialisation

I'm using Visual Studio 2012 for this demo but you'll probably be fine with VS 2010 and VS 2013 as well. If you followed along the original tutorial then you can open that solution. Otherwise create a new blank solution in Visual Studio and insert a solution folder called Serialisation. Add two Console app projects to this folder: SerialisationSender and SerialisationReceiver. Add the following NuGet packages to both:



(https://dotnetcodr.files.wordpress.com/2014/04/rabbitmq-new-client-package-nuget.png)



(https://dotnetcodr.files.wordpress.com/2014/04/newtonsoft-json-net-nuget-package.png)

Add a class library called SharedObjects and add the same RabbitMq NuGet package to it as above. Insert an object called Customer:

```
1  public class Customer
2  {
3      public string Name { get; set; }
4  }
```

Set a reference to this class library from both the Sender and the Receiver console apps. Insert another class called CommonService to the class library:

```
public class CommonService
{
    private string _hostName = "localhost";
    private string _userName = "guest";
    private string _password = "guest";

    public static string SerialisationQueueName = "SerialisationDemoQueue";

    public IConnection GetRabbitMqConnection()
    {
        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.HostName = _hostName;
        connectionFactory.UserName = _userName;
        connectionFactory.Password = _password;

        return connectionFactory.CreateConnection();
    }
}
```

Next we'll set up the queue for this demo. Add the following code to Program.cs in the Sender app:

```
static void Main(string[] args)
{
    CommonService commonService = new CommonService();
    IConnection connection = commonService.GetRabbitMqConnection();
    IModel model = connection.CreateModel();
    SetupSerialisationMessageQueue(model);
}

private static void SetupSerialisationMessageQueue(IModel model)
{
    model.QueueDeclare(CommonService.SerialisationQueueName, true, false, false, null);
}
```

Run the Sender app and check in the RabbitMQ management console that the queue has been created. Comment out the call to SetupSerialisationMessageQueue in Main. Insert the following method to the Sender:

```
private static void RunSerialisationDemo(IModel model)
{
    Console.WriteLine("Enter customer name. Quit with 'q'.");
    while (true)
    {
        string customerName = Console.ReadLine();
        if (customerName.ToLower() == "q") break;
        Customer customer = new Customer() { Name = customerName };
        IBasicProperties basicProperties = model.CreateBasicProperties();
        basicProperties.SetPersistent(true);
        String jsonified = JsonConvert.SerializeObject(customer);
        byte[] customerBuffer = Encoding.UTF8.GetBytes(jsonified);
        model.BasicPublish("", CommonService.SerialisationQueueName, basicProperties, customerBuffer)
    }
}
```

There's not much magic going on: we enter the customer name, construct the Customer object, build a JSON object out of it, get the byte array out of the JSON string and send it to the message queue. Add a call to this method from Main:

```
RunSerialisationDemo(model);
```

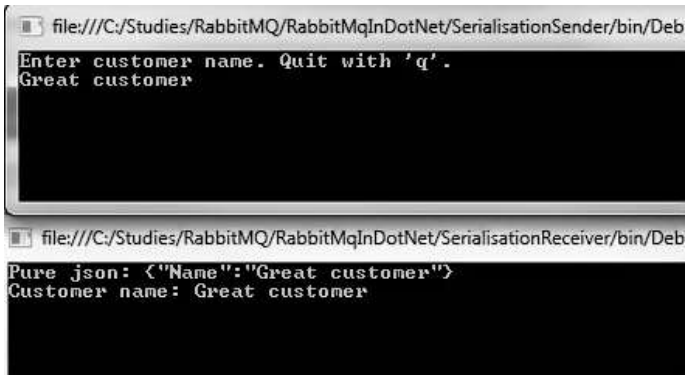
In the SerialisationReceiver add the following code to Program.cs:

```
static void Main(string[] args)
{
    CommonService commonService = new CommonService();
    IConnection connection = commonService.GetRabbitMqConnection();
    IModel model = connection.CreateModel();
    ReceiveSerialisationMessages(model);
}

private static void ReceiveSerialisationMessages(IModel model)
{
    model.BasicQos(0, 1, false);
    QueueingBasicConsumer consumer = new QueueingBasicConsumer(model);
    model.BasicConsume(CommonService.SerialisationQueueName, false, consumer);
    while (true)
    {
        BasicDeliverEventArgs deliveryArguments = consumer.Queue.Dequeue() as BasicDeliverEventArgs;
        String jsonified = Encoding.UTF8.GetString(deliveryArguments.Body);
        Customer customer = JsonConvert.DeserializeObject<Customer>(jsonified);
        Console.WriteLine("Pure json: {0}", jsonified);
        Console.WriteLine("Customer name: {0}", customer.Name);
        model.BasicAck(deliveryArguments.DeliveryTag, false);
    }
}
```

This bit of code should look very familiar to you from the foundations course: we're listening to a specific queue and extract any incoming messages. The only new bit is that we deserialise the JSON string into a Customer object.

The demo is ready to run. Start the Sender app. Then right-click the Receiver project, select Debug, Start new Instance. You'll have two console window up and running. Start sending customer names to the Receiver from the Sender:



(https://dotnetcodr.files.wordpress.com/2014/04/serialised-message-output.png)

The same technique works for more complicated objects with multiple properties and other objects as properties.

Setting the content type

JSON is not the only message type we can send. The other two common message formats are XML and binary. No matter how you format your message, it will need to be sent to the message queue as a byte array. We've already seen how to serialise a JSON message. For XML you can use the following method:

```
private byte[] SerialiseIntoXml(Customer customer)
{
    MemoryStream memoryStream = new MemoryStream();
    XmlSerializer xmlSerialiser = new XmlSerializer(customer.GetType());
    xmlSerialiser.Serialize(memoryStream, customer);
    memoryStream.Flush();
    memoryStream.Seek(0, SeekOrigin.Begin);
        return memoryStream.GetBuffer();
}
```

…and to get the Customer object from a binary format you can use the following method:

```
private byte[] SerialiseIntoBinary(Customer customer)
{
    MemoryStream memoryStream = new MemoryStream();
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    binaryFormatter.Serialize(memoryStream, customer);
    memoryStream.Flush();
    memoryStream.Seek(0, SeekOrigin.Begin);
    return memoryStream.GetBuffer();
}
```

Note that the Customer object must be serialisable for these to work:

```
[Serializable]
public class Customer
{
      public string Name { get; set; }
}
```

So now we can serialise our message in 3 different ways. It now makes sense to denote the content type of the message. The IBasicProperties interface has a property called ContentType where you can set the MIME type using the well-known values below:

- JSON: application/json
- XML: text/xml
- Binary: application/octet-stream

Example:

```
[Serializable]
IBasicProperties basicProperties = model.CreateBasicProperties();
basicProperties.SetPersistent(true);
basicProperties.ContentType = "application/json";
```

The properties are sent along in the BasicPublish method so the MIME type is preserved.

On the client side you can read the MIME type as follows:

```
BasicDeliverEventArgs deliveryArguments = consumer.Queue.Dequeue() as BasicDeliverEventArgs;
string contentType = deliveryArguments.BasicProperties.ContentType;
```

Also, the client will need to deserialise the message. We've already seen how to do that in the case of the JSON format. For XML you can have the following helper method:

```
private Customer DeserialiseFromXml(byte[] messageBody)
{
    MemoryStream memoryStream = new MemoryStream();
    memoryStream.Write(messageBody, 0, messageBody.Length);
    memoryStream.Seek(0, SeekOrigin.Begin);
    XmlSerializer xmlSerialiser = new XmlSerializer(typeof(Customer));
    return xmlSerialiser.Deserialize(memoryStream) as Customer;
}
```

…and for the binary format you use something like this:

```
private Customer DeserialiseFromBinary(byte[] messageBody)
{
    MemoryStream memoryStream = new MemoryStream();
    memoryStream.Write(messageBody, 0, messageBody.Length);
    memoryStream.Seek(0, SeekOrigin.Begin);
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    return binaryFormatter.Deserialize(memoryStream) as Customer;
}
```

Note that the Customer object must be shared between Sender and the Receiver for binary serialisation to work.

Denoting the type of the message

It can happen that the Receiver doesn't know in advance what type of message is coming, i.e. if it's a Customer, an Order, a Product etc.

You can solve this issue using the Type property of the IBasicProperties object when sending the message from the Sender. It is a string property:

```
1 IBasicProperties basicProperties = model.CreateBasicProperties();
2 basicProperties.SetPersistent(true);
3 basicProperties.ContentType = "application/json";
4 basicProperties.Type = "Customer";
```

And then you can read the type in the Receiver as follows:

```
1 BasicDeliverEventArgs deliveryArguments = consumer.Queue.Dequeue() as BasicDeliverEventArgs;
2 string contentType = deliveryArguments.BasicProperties.ContentType;
3 string objectType = deliveryArguments.BasicProperties.Type;
```

You are free to set the value of the Type property. You can do it in a simple way like above, but the Receiver will need to know those values. In the world of open APIs and automatic documentation generators this shouldn't be a serious obstacle. Other solutions:

- Fully qualified name, including the namespace, such as "SharedObjects.Customer". This is easy to retrieve with the typeof keyword: typeof(Customer).ToString(). This approach is mostly viable within the .NET world, where both the Sender and the Receiver can work with .NET objects
- Canonical messages (http://en.wikipedia.org/wiki/Canonical_model) where the object type is described using XSD (http://en.wikipedia.org/wiki/XSD) along with the root element (http://www.w3schools.com/schema/schema_schema.asp). This approach works best if interoperability between disparate systems is a must

We'll look at the first option in the next post (https://dotnetcodr.com/2014/06/09/rabbitmq-in-net-data-serialisation-ii/).

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

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

3 Responses to *RabbitMQ in .NET: data serialisation I*

Vince says:
October 19, 2014 at 1:27 pm
Thanks for your Job Andras…

Reply

Quang Liem says:
September 16, 2015 at 6:05 pm
Reblogged this on liemqv blog.

Reply

Ombasa Mukhwami says:
June 28, 2019 at 9:54 am
very resourceful

Reply