RabbitMQ by Pivotal

1 "Hello World!"

The simplest thing that does *something*

Python Java Ruby PHP C# JavaScript Go Elixir Objective-C Swift Spring AMQP

2 Work queues

Distributing tasks among workers (the competing consumers pattern)

Python Java Ruby PHP C# JavaScript Go Elixir Objective-C Swift Spring AMQP

3 Publish/Subscribe

Sending messages to many consumers at once

Python Java Ruby PHP C# JavaScript Go Elixir Objective-C Swift Spring AMQP

4 Routing

Receiving messages selectively

Python Java Ruby PHP C# JavaScript Go Elixir Objective-C Swift Spring AMQP

5 Topics

Receiving messages based on a pattern (topics)

Python Java Ruby PHP C# JavaScript Go Elixir Objective-C Swift Spring AMQP

6 RPC

Request/reply pattern example

Python Java Ruby PHP C# JavaScript Go Elixir Spring AMQP

7 Publisher Confirms

Reliable publishing with publisher confirms

Java C#

Introduction

RabbitMQ is a message broker: it accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. or Ms. Mailperson will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data – *messages*.

RabbitMQ, and messaging in general, uses some jargon.

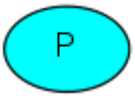
Prerequisites

This tutorial assumes RabbitMQ is installed and running on `localhost` on standard port (`5672`). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

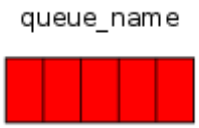
If you're having trouble going through this tutorial you can contact us through the mailing list.

- *Producing* means nothing more than sending. A program that sends messages is a *producer* :

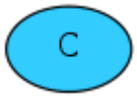


- *A queue* is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a *queue*. A *queue* is only bound by the host's memory & disk limits, it's essentially a large message buffer. Many *producers* can send messages that go to one queue, and many *consumers* can try to receive data from one *queue*. This is how we represent a queue:

can try to receive data from one *queue*. This is how we represent a queue:



- *Consuming* has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages:



Note that the producer, consumer, and broker do not have to reside on the same host; indeed in most applications they don't. An application can be both a producer and consumer, too.

"Hello World"

(using the .NET/C# Client)

In this part of the tutorial we'll write two programs in C#; a producer that sends a single message, and a consumer that receives messages and prints them out. We'll gloss over some of the detail in the .NET client API, concentrating on this very simple thing just to get started. It's a "Hello World" of messaging.

In the diagram below, "P" is our producer and "C" is our consumer. The box in the middle is a queue - a message buffer that RabbitMQ keeps on behalf of the consumer.

The .NET client library

RabbitMQ speaks multiple protocols. This tutorial uses AMQP 0-9-1, which is an open, general-purpose protocol for messaging. There are a number of clients for RabbitMQ in many different languages. We'll use the .NET client provided by RabbitMQ.

The client supports .NET Core as well as .NET Framework 4.5.1+. This tutorial will use RabbitMQ .NET client 5.0 and .NET Core so you will ensure you have it installed and in your PATH.

You can also use the .NET Framework to complete this tutorial however the setup steps will be different.

RabbitMQ .NET client 5.0 and later versions are distributed via nuget.

This tutorial assumes you are using powershell on Windows. On MacOS and Linux nearly any shell will work.

Setup

First lets verify that you have .NET Core toolchain in `PATH`:

```
dotnet --help
```

should produce a help message.

Now let's generate two projects, one for the publisher and one for the consumer:

```
dotnet new console --name Send
mv Send/Program.cs Send/Send.cs
dotnet new console --name Receive
mv Receive/Program.cs Receive/Receive.cs
```

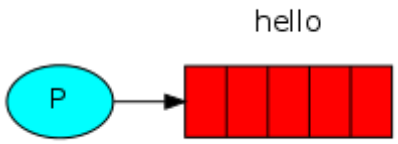
This will create two new directories named `Send` and `Receive`.

Then we add the client dependency.

```
cd Send
dotnet add package RabbitMQ.Client
dotnet restore
cd ../Receive
dotnet add package RabbitMQ.Client
dotnet restore
```

Now we have the .NET project set up we can write some code.

Sending



We'll call our message publisher (sender) `Send.cs` and our message consumer (receiver) `Receive.cs`. The publisher will connect to RabbitMQ, send a single message, then exit.

In `Send.cs`, we need to use some namespaces:

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

Set up the class:

```
class Send
{
    public static void Main()
    {
        ...
    }
}
```

then we can create a connection to the server:

```
class Send
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        {
            using (var channel = connection.CreateModel())
            {
                ...
            }
        }
    }
}
```

The connection abstracts the socket connection, and takes care of protocol version negotiation and authentication and so on for us. Here we connect to a broker on the local machine - hence the *localhost*. If we wanted to connect to a broker on a different machine we'd simply specify its name or IP address here.

Next we create a channel, which is where most of the API for getting things done resides.

To send, we must declare a queue for us to send to; then we can publish a message to the queue:

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

class Send
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello",
                                 durable: false,
                                 exclusive: false,
                                 autoDelete: false,
                                 arguments: null);

            string message = "Hello World!";
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: "",
                                 routingKey: "hello",
                                 basicProperties: null,
                                 body: body);
            Console.WriteLine(" [x] Sent {0}", message);
        }

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }
}
```

Declaring a queue is idempotent - it will only be created if it doesn't exist already. The message content is a byte array, so you can encode whatever you like there.

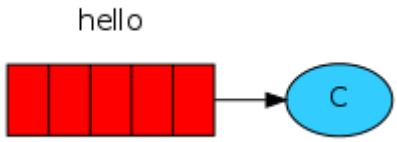
When the code above finishes running, the channel and the connection will be disposed. That's it for our publisher.

Here's the whole Send.cs class.

> **Sending doesn't work!**
>
> If this is your first time using RabbitMQ and you don't see the "Sent" message then you may be left scratching your head wondering what could be wrong. Maybe the broker was started without enough free disk space (by default it needs at least 50 MB free) and is therefore refusing to accept messages. Check the broker logfile to confirm and reduce the limit if necessary. The configuration file documentation will show you how to set `disk_free_limit`.

Receiving

As for the consumer, it listening for messages from RabbitMQ. So unlike the publisher which publishes a single message, we'll keep the consumer running continuously to listen for messages and print them out.



The code (in `Receive.cs`) has almost the same `using` statements as `Send`:

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

Setting up is the same as the publisher; we open a connection and a channel, and declare the queue from which we're going to consume. Note this matches up with the queue that `Send` publishes to.

```
class Receive
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        {
```

```
            using (var channel = connection.CreateModel())
            {
                channel.QueueDeclare(queue: "hello",
                                     durable: false,
                                     exclusive: false,
                                     autoDelete: false,
                                     arguments: null);
                ...
            }

        }
    }
}
```

Note that we declare the queue here as well. Because we might start the consumer before the publisher, we want to make sure the queue exists before we try to consume messages from it.

We're about to tell the server to deliver us the messages from the queue. Since it will push us messages asynchronously, we provide a callback. That is what `EventingBasicConsumer.Received` event handler does.

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

class Receive
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello",
                                 durable: false,
                                 exclusive: false,
                                 autoDelete: false,
                                 arguments: null);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
```

```
            {
                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine(" [x] Received {0}", message);
            };
            channel.BasicConsume(queue: "hello",
                                 autoAck: true,
                                 consumer: consumer);


            Console.WriteLine(" Press [enter] to exit.");
            Console.ReadLine();
        }
    }
}
```

Here's the whole Receive.cs class.

Putting It All Together

Open two terminals.

Run the consumer:

```
cd Receive
dotnet run
```

Then run the producer:

```
cd Send
dotnet run
```

The consumer will print the message it gets from the publisher via RabbitMQ. The consumer will keep running, waiting for messages (Use Ctrl-C to stop it), so try running the publisher from another terminal.

Time to move on to part 2 and build a simple *work queue*.

Production [Non-]Suitability Disclaimer

Please keep in mind that this and other tutorials are, well, tutorials. They demonstrate one new concept at a time and may intentionally oversimplify some things and leave out others. For example topics such as connection management, error handling, connection recovery, concurrency and metric collection are largely omitted for the sake of brevity. Such simplified

code should not be considered production ready.

Please take a look at the rest of the documentation before going live with your app. We particularly recommend the following guides: Publisher Confirms and Consumer Acknowledgements, Production Checklist and Monitoring.

Getting Help and Providing Feedback

If you have questions about the contents of this tutorial or any other topic related to RabbitMQ, don't hesitate to ask them on the RabbitMQ mailing list.

Help Us Improve the Docs <3

If you'd like to contribute an improvement to the site, its source is available on GitHub. Simply fork the repository and submit a pull request. Thank you!