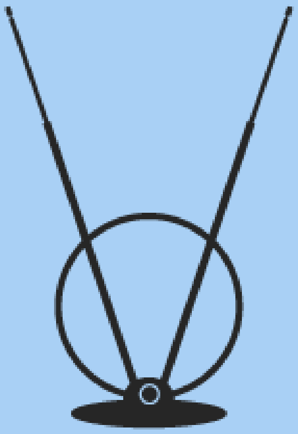


Message Queuing with RabbitMQ Succinctly

by Stephen Haunts

Syncfusion | Technology Resource Portal

Message Queuing with RabbitMQ Succinctly

By

Stephen Haunts

Foreword by Daniel Jebaraj

Table of Contents

The Story behind the *Succinctly* Series of Books **6**

About the Author **8**

Introduction **9**

What This Book Isn't 9

Sample Code Projects 9

Chapter 1 Message Queuing Overview **10**

Common Message Queuing Traits 10

Message Queuing Protocols 11

Chapter 2 RabbitMQ Overview **12**

Chapter 3 AMQP Messaging Standard **14**

Exchanges 15

Direct Exchange 16

Fanout Exchange 17

Topic Exchange 18

Headers Exchange 19

Queues 19

Bindings 20

Consumers 20

Message Acknowledgements 20

Rejecting Messages 21

Chapter 4 Installing and Configuring RabbitMQ **22**

Basic Installation 22

Setting Up the Management Portal 24

Chapter 5 Overview of the Management Plug-in **27**

Chapter 6 Administration via the Command Line 34

Chapter 7 Basic Queue and Message Example 39

Chapter 8 Working Examples 44

RabbitMQ Client Library.. 44

What is Contained in the Client Library? 44

Installing the .NET Client Library 46

Example Code Scenario 47

Common Code Throughout the Examples 47

Payment.cs 48

PurchaseOrder.cs 49

ObjectSerialize... 49

Example 1: Basic Queue 52

Example 2: Worker Queue... 57

Example 3: Publisher and Subscriber.. 67

Example 4: Direct Routing 74

Example 5: Topic-based Publish and Subscribe 84

Example 6: Remote Procedure Call 96

Closing Notes 108

Technical Reviewer: Peter Shaw
Copy Editor: Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet, and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just like everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

About the Author

Stephen Haunts has been developing software and applications professionally since 1996 and as a hobby since he was 10 years old. Stephen has worked across many different industries including computer games, online banking, retail finance, and healthcare and pharmaceuticals. Stephen started out programming in BASIC on machines such as the Dragon 32, Vic 20, and the Amiga, and moved onto C and C++ on the IBM PC. Stephen has been developing software in C# and the .NET framework since first being introduced to it in 2003.

As well as being an accomplished software developer, Stephen is also an experienced development leader and has led, mentored, and coached teams to deliver many high-value, high-impact solutions in finance and healthcare.

Outside of Stephen's day job, he is also an experienced tech blogger who runs a popular blog called Coding in the Trenches on his own website and he is also a training course author for the online training company Pluralsight. Stephen also runs several open-source projects including SafePad, Text Shredder, Block Encrpytor, and the post-deployment testing tool Smoke Tester.

Stephen is also an accomplished electronic musician and sound designer.

Introduction

Never before has the integration of systems in the enterprise been so important. It is common that development teams need to integrate in-house development systems together with third-party systems. The more systems that need to be integrated together, the more moving parts you have, and the more potential for failure and instability to creep in. Choosing the right integration platforms is important. We are spoiled with choices when it comes to both open-source and commercial solutions.

In this book, we will look at a popular integration platform called RabbitMQ. RabbitMQ is a powerful message broker and queuing system that allows you to integrate systems together so that they are robust, reliable, and can scale with your business.

This book will take you from novice to expert in no time. We will cover how to install and set up RabbitMQ, and how to configure it from the control panel to the command line. We will then do a deep dive into the features of RabbitMQ and work through a number of practical code examples.

We will focus on .NET and C# in this book but RabbitMQ is a cross-platform system with client libraries for most popular development platforms. This makes RabbitMQ a compelling solution for integrating systems developed in any number of languages and platforms.

What This Book Isn't

This book is not an exhaustive reference manual for RabbitMQ. There are plenty of other books out there that do this. This book's aim is to get you up and running with a good level of proficiency with RabbitMQ as quickly as possible. This will mostly be through a series of workable examples focusing on real-world scenarios to which you can refer to as you integrate RabbitMQ into your enterprise.

Sample Code Projects

This book uses a sample solution that contains each of the examples discussed in the second half of this book. The examples were created in Visual Studio 2013 and built against RabbitMQ 3.4.4.

I recommend that you download this code and follow along with it while working through this book.

Note: The sample code can be downloaded from https://bitbucket.org/syncfusiontech/message-queuing-with-rabbitmq-succinctly.

Chapter 1 Message Queuing Overview

Message queuing gives you a mechanism to allow an application to asynchronously send a message to a receiver. This means that the sender and receiver do not need to interact with the message at the same time. A message is sent to a queue where it is stored until the receiver retrieves the message.

Message queues can be inter-process where the queue resides in memory on a single server or for integrating systems across multiple servers. This can be done by using in-memory queues but it is also common to use durable queues in which the messages are persisted to disk, meaning that messages are not lost should any system or server go offline for any period of time.

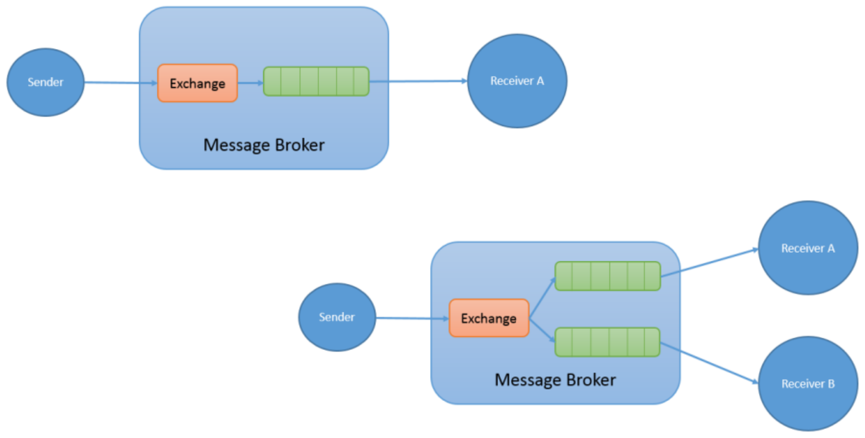


Figure 1 Message queue examples with single and multiple queues.

Message queuing systems come in many forms—both as commercial proprietry products and as open-source products. An example of a commercial solution is IBM MQ. Examples of open-source message queuing systems include RabbitMQ, JBoss Messaging , and Apache ActiveMQ.

The open-source message queuing systems are free to download and use but it is also common for companies to provide commercial support agreements (which you can pay for that allow your organization to reach out for technical support should you run into any production issues). RabbitMQ is no different in this respect and Pivotal Software, Inc., the company that owns RabbitMQ, provides a purchasable support agreement. Having a support agreement is normally a prerequisite for large organizations adopting an open-source platform.

Common Message Queuing Traits

A messaging broker is installed onto a server or a set of servers, and a series of name queues and exchanges are defined where they are registered with the message broker. An application can then send a message to the queue and any number of receiving systems can receive the message and act on its contents.

Queuing systems such as RabbitMQ will typically have the following features available to them:

- **Durability**: Messages can either be kept in memory or persisted to disk. If they are persisted to disk, then the message will be preserved should the server containing the queues crash. If this happens, once the server comes back up, the receiving applications can pick up where they left off.
- **Security**: A message broker should be able to let you set up different security permissions for different users. Some applications may have permission to write but not read from the queue, or another application could just have permissions to read from the queue.
- **Time to Live**: A message may have a time to live property set on it. This will mean a message can be set to expire after a certain timespan.
- **Filtering and Routing**: A messaging system might have the ability to filter the messages to decide to which queue a message should be delivered.
- **Batching**: Batching allows messages to be saved up and then delivered all at the same time as part of a larger batch; this is typically used when a destination server is not always online.
- **Receipt Acknowledgement**: A message publisher may require an acknowledgement from the message broker that the message has been successfully delivered.

Message Queuing Protocols

Many commercial message queuing systems are based on proprietary protocols. These protocols are kept closed so that the vendor can restrict what operating systems (OSes) and development platforms can interact with the message broker. This closed nature means that, if any customer of the messaging broker provider wanted it to work with another platform, they might have to pay a high consultancy fee to get the support they need.

With the advent of open-source messaging systems, a number of open queuing protocol standards have been developed. The two most predominant standards are:

1. **Advanced Message Queuing Protocol (AMQP)**: This is a feature-rich message queuing protocol and it is the protocol used by RabbitMQ.
2. **Streaming Text-Oriented Messaging Protocol (STOMP)**: Stomp is a simple text-based messaging protocol.

Later on in this book, we will look in more detail at the AMQP.

Chapter 2 RabbitMQ Overview

RabbitMQ is an open-source messaging system that allows you to integrate applications together by using messages and queues. RabbitMQ implements the AMQP. The underlying RabbitMQ server is written in the Erlang programming language which was originally designed for the telecoms industry by Ericsson. Erlang supports distributed, fault-tolerant applications and is, therefore, an ideal language to use to build a message queuing system.

Figure 2: RabbitMQ logo

While RabbitMQ is built with Erlang, it also supports many other development platforms via client libraries. Because of Erlang's heritage of working with telecoms networks, it is the ideal platform for handling messages for enterprise applications.

The RabbitMQ server is a message broker that acts as the message coordinator for the applications that you want to integrate together. This means that you can give your systems a common platform for sending and receiving messages.

Rabbit MQ contains many features to make your systems integration as painless as possible. These include:

- **Reliability**: With RabbitMQ being built on top of Erlang, the message broker is already built on top of solid, high-performance, reliable, and durable foundations. Messages can be persisted to disk to guard from lost messages in the event that a server is restarted, and you can send message delivery acknowledgements to a sender so they can be sure that the message has been received and stored.
- **Routing**: RabbitMQ works by passing your messages through exchanges before they are stored in a queue. There are different exchange types which let you perform routing, but you can also work with more complex routing scenarios by binding exchanges together.
- **Clustering and High Availability**: To increase the reliability and availability of RabbitMQ, you can cluster several servers together on a local network which forms a single logical message broker. Queues can also be mirrored across multiple servers in a cluster so that, in the advent of a server failure, you will not lose any messages.
- **Management Web User Interface (UI)**: RabbitMQ comes with a browser-based UI that lets you manage users and their permissions, exchanges, queues, and more. This tool is an excellent window into your RabbitMQ servers.

- **Command-Line Interface**: In addition to the Management UI, there is a command-line tool called "***rabbitmqctrl***" and "***rabbitmqadmin***." These command-line tools offer the same level of administration as the web UI, with the added advantage that you can incorporate RabbitMQ administration in scripts.

Note: Clustering is out of the scope of this book. More information on it can be found on the RabbitMQ website.

A major benefit of using RabbitMQ is the fact that it is a cross-platform system. You can run Erlang and the RabbitMQ server on various platforms including Windows Servers, Linux and Unix systems such as (Debian/Ubuntu, Fedora, and Solaris), Mac OS X, and on various cloud platforms such as Amazon EC2 and Microsoft Azure. Rabbit MQ also has client libraries that support many different programming environments such as Microsoft .NET, Java, Erlang, Ruby, Python, PHP, Perl, C/C++, Node.JS, Lisp, and more.

This cross-platform nature of RabbitMQ means you could have clients written in different programming languages easily sending and receiving messages from a RabbitMQ server hosted in any of the supported environments.

RabbitMQ is an open-source messaging solution but, as a company, you have the option of taking out a paid support plan (which is a requirement for many large organizations).

RabbitMQ is built to the AMQP Version 0-9-1. AMQP is a networking protocol that enables client applications to communicate with messaging middleware broker servers. In our case, the client will be our .NET application and the server is the RabbitMQ server or cluster of servers. Let's take a closer look at the AMQP protocol.

Chapter 3 AMQP Messaging Standard

RabbitMQ is built on top of the AMQP. This is a network protocol that enables client applications to communicate with a compatible messaging system.

A message broker works by receiving messages from a client (publisher) and that broker routes the message to a receiving application (consumer).

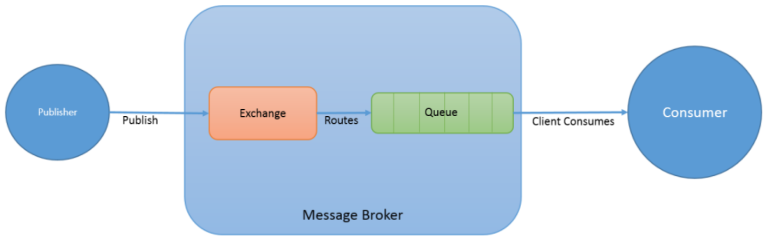


Figure 3: High-level overview of the AMQP

RabbitMQ currently supports version 0-9-1 of the AMQP protocol. With the AMQP protocol, a message is published by an application to an exchange. You can think of the exchange as a mailbox. The exchange then sends the message to a queue by using different rules called bindings. This is all within the message broker.

The message broker will then deliver the message from the queue to a consumer (the consumer pulls the message) that is subscribed to the queue. When a message is published to a queue, a publisher can specify various different message attributes. Some of these attributes will be used by the message broker but the rest is completely opaque to the broker and is only used by any applications that receive the message.

Due to network unreliability, applications could fail to correctly process messages, therefore the AMQP protocol has a mechanism for message acknowledgements. This means when a message is delivered to a consuming application, the consumer notifies the broker (either automatically or as soon as the application developer chooses to do so). When message acknowledgements are used, the message broker will only remove the message from the queue when it receives a notification for that message.

If you are using messages that are routed with a routingKey and the message cannot be routed anywhere, it can either be returned to the sender, dropped or, if configured, can be placed into a dead letter queue which is monitored. A publishing application will choose how to handle this situation by publishing messages by using certain parameters.

Exchanges

Exchanges are AMQP entities where messages are sent to the message broker. Exchanges take a message and then route it to one or more queues. The type of routing depends upon the exchange type used and different exchange rules called bindings. RabbitMQ supports AMQP 0-9-1 brokers which provides four exchange types. These are direct exchanges, fanout exchanges, topic exchanges, and headers exchanges.

Each exchange is also declared with a number of different attributes. The most important attributes to us are:

- **Name**: The name of the exchange.
- **Durability**: This flag determines whether or not messages sent to the exchange survive a broker/server restart by persisting the messages to disk.
- **Auto-delete**: This flag determines if the exchange is deleted when all of the queues have finished using it.
- **Arguments**: These are message broker-dependent.

AMQP message brokers contain a default exchange that is a direct exchange with no name (i.e., empty string) that has been predeclared. It has one special property that makes it useful for simple applications: every queue that is created is automatically bound to it with a routing key which is the same as the queue name.

For example, when you declare a queue with the name of "payment-requests," the message broker will bind it to the default exchange by using "payment-requests" as the routing key. In other words, the default exchange makes it seem as if it is possible to directly deliver messages to queues even though that is not technically what is happening.

Direct Exchange

A direct exchange delivers messages to queues that are based on a message routing key. A direct exchange is ideal if you need to publish a message onto just one queue (like with a more traditional MSMQ setup) but you can also route messages onto multiple queues as well.

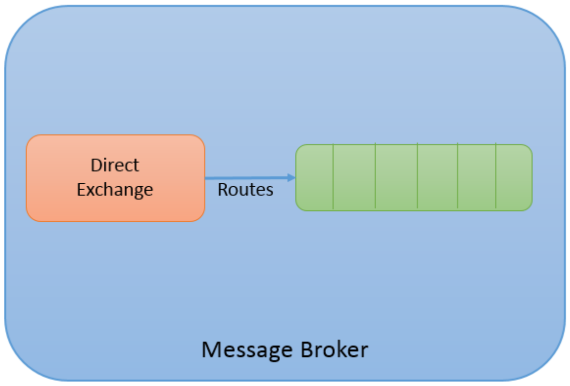


Figure 4: AMQP direct exhange routing with a routing key

A direct exchange works as follows: a queue binds to the exchange with a routing key (payment-message, for example). Then, when a new message with a routing key of payment-message arrives at the direct exchange, the exchange routes the message to the queue if both routing keys match.

Direct queues are commonly used to distribute messages between multiple worker processes in a round robin manner. Later in the book, we'll develop a sample that does exactly this.

Fanout Exchange

A fanout exchange routes messages to all of the queues that are bound to it and the routing key is ignored. If 10 queues are bound to a fanout exchange, when a new message is published to that exchange, a copy of the message is delivered to all 10 queues. Fanout exchanges are ideal for the broadcast routing of messages.

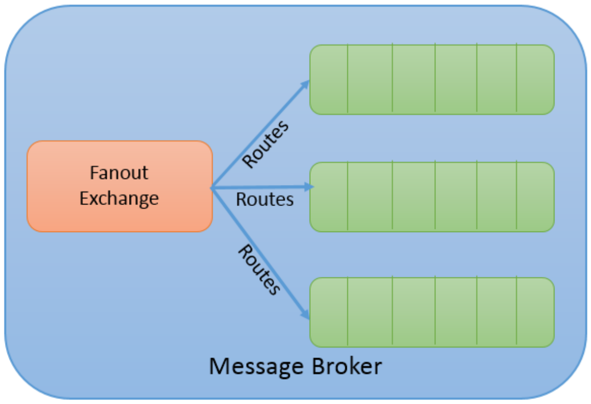


Figure 5: AMQP fanout exchange

This is in contrast to the direct exchange where the message is targeted to the relevant queue by using the routing key. So, if you want to broadcast a message to all of the queue consumers, the fanout exchange is what you want to use.

Some example uses of this might be:

- Sending online game scores to all players.
- Sending weather updates to all interested systems.
- Chat sessions between groups of people.

Topic Exchange

Topic exchanges route messages to one or many queues based upon matching between a message routing key and the pattern that was used to bind a queue to an exchange. The topic exchange type is often used to implement various publish/subscribe pattern variations. Topic exchanges are commonly used for the multicast routing of messages to different queues.

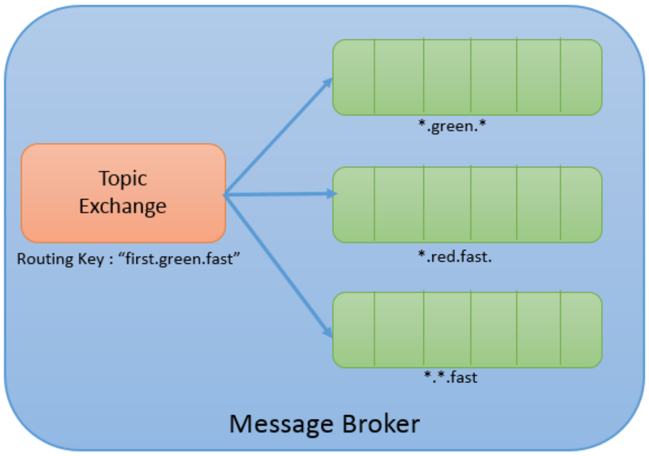


Figure 6: Topic exchange

Topic exchanges have a broad set of use cases. Whenever a problem involves multiple consumers/applications that selectively choose which type of messages they want to receive, the use of topic exchanges should be considered.

A topic exchange works by using a wildcarded routing key. For example, in the context of a company departmental hierarchy, if you send a message to "all.*.*", the message will be sent to all of the departments. If you sent a message to "all.payroll.*", the message will only be sent to consumers who are interested in the payroll department.

The topic exchange is powerful and can behave like other exchanges already discussed.

When a queue is bound with the "#" (hash) binding key, it will receive all of the messages regardless of the routing key, in exactly the same manner as a fanout exchange.

When special characters "*" (star) and "#" (hash) aren't used in bindings, the topic exchange will behave just like a regular direct exchange.

Some typical examples of this exchange might be:

- Sending messages to relevant departments in an organization.
- Stock prices for certain types of companies.
- Categorized news updates (e.g, business, technology, entertainment).

Headers Exchange

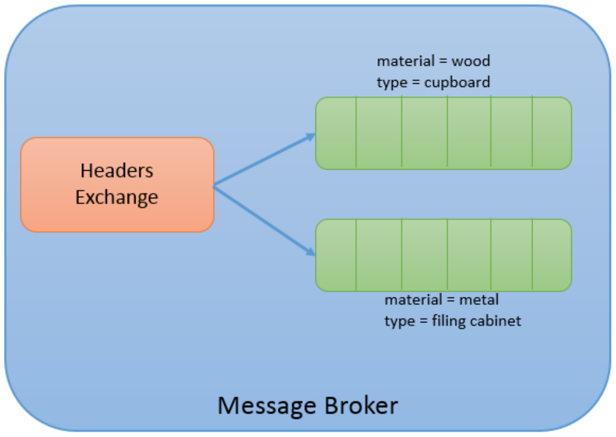


Figure 7: Headers exchange

The headers exchange is for routing on multiple attributes that are expressed in headers. This means that the routing key is ignored for this type of exchange due to the fact that it can only express one piece of information. You can assign multiple header values that pass into the exchange and are routed to the queues.

In Figure 7, the first queue takes messages that are routed with the header values of 'material = wood' and 'type = cupboard.' The second queue takes messages that are routed with 'material = metal' and 'type = filing cabinet.'

Header exchanges can be looked at as supercharged direct exchanges as they route based on header values, and they can be used as direct exchanges where the routing key does not have to be a string.

Queues

Queues in AMQP are similar to queues in any other messaging system. Messages are placed onto a queue and they work on a first in, first out (FIFO) basis. Queues have the following additional properties over exchanges:

- **Name**: The name of the queue.
- **Durable**: The queue and messages will survive a broker or server restart.
- **Exclusive**: The queue is used by only one connection and the queue will be deleted when that collection closes.
- **Auto Delete**: The queue is deleted when the consumer or subscriber unsubscribes.

Before you can use a queue, it must be declared. If the queue doesn't already exist, it will be created. If the queue already exists, then redeclaring the queue will have no additional effect on the queue that already exists.

Queue names can be picked by the application or they can be automatically named by the broker that generates them. We will see examples of this later on in the book. Queue names can be up to 255 characters in length. If you want the broker to pick the queue name for you, then you don't pass a name at the point where you declare the queue.

Queues can be made durable which means the queue is persisted to the disk. This only makes the queue persistent and not the messages. Queue durability means the queue will be redeclared once the broker is restarted. If you want the messages to be also persisted, then you have to post persistent messages. This is useful if you need the messages to remain intact if the broker or server is restarted. Making queues durable does come with additional overhead so you need to decide if you need this enabling but, if your application can't be in a position where it can lose messages, then you need queue durability.

Bindings

When you want to define rules that specify how messages are routed from exchanges to queues, you need to define bindings. Bindings may have an optional *routing key* attribute that is used by some exchange types to route messages from the exchange to the queue. The purpose of the routing key is to select certain messages published to an exchange to be routed to the bound queue. This means that the routing key acts like a filter.

If an AMQP message cannot be routed to any queue because it does not have a valid binding from the exchange to the queue, then it is either dropped or returned to the publisher depending upon message attributes the publisher has set.

Consumers

Storing messages in queues is all well and good but you need applications on the other side of the queues to consume those messages. Typically, applications will register as a consumer or subscribe to a queue. You can have more than one application registered as a subscriber to the queue, and this is a common usage scenario when you want to balance the load of applications feeding from the queue in high-volume scenarios.

Message Acknowledgements

When a consuming application acts on a message from the queue, it is possible that a problem could occur in that application which means that message is lost. Generally, when an application acts on a message, that message is removed from the queue but you may not want this to happen until you have successfully processed the message. The AMQP protocol gives you a couple of ways of defining when a message is removed from the queue.

- The message is removed once the broker has sent the message to the application.

- The message is removed once the application has sent an acknowledgement message back to the broker.

The first example uses an automatic acknowledgement from the application to remove the message whereas the second example uses an explicit acknowledgement. With the explicit acknowledgement, it is up to the application to decide when to remove the message from the queue. This could be when you have just received the message or after you have processed it.

If the consuming application crashes before the acknowledgement has been sent, then the message broker will try to redeliver the message to another consumer.

Rejecting Messages

When an application processes a message, that processing may or may not succeed. If the processing fails for any reason (for example, there is a database time-out), then the consuming application can reject the message. When this happens, the application can ask the broker to discard the message or requeue it. If there is only one consumer application subscribed to a queue, you need to make sure you do not create an infinite message delivery loop by rejecting and requeuing a message from the same consumer over and over again.

This has been a whistle stop tour of the AMQP protocol but it has not been exhaustive. If you wish to read up on the full specification, then you can do so on the AMQP Working Group website.

Chapter 4 Installing and Configuring RabbitMQ

Basic Installation

Now that we have covered the basics of message queuing, RabbitMQ, and the AMQP model, let's get RabbitMQ installed and configured. When you set up RabbitMQ on a server, you need to install two components. First, you need the Erlang run time and then RabbitMQ itself. First, go to the RabbitMQ website to download it.

Once you are on this page, select "Install: Windows" from the grey panel to the right of the screen:

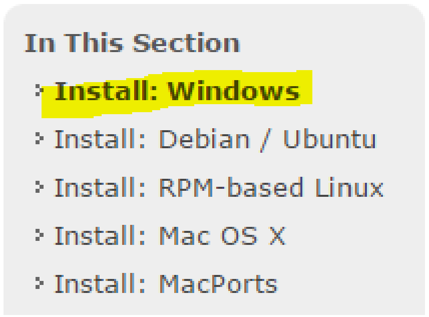
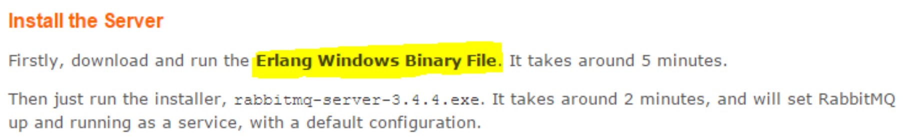


Figure 8: Link to install the Windows version of RabbitMQ

From this page, click "Erlang Windows Binary File" as shown in the following screenshot. This will take you to the Erlang website's downloads page:



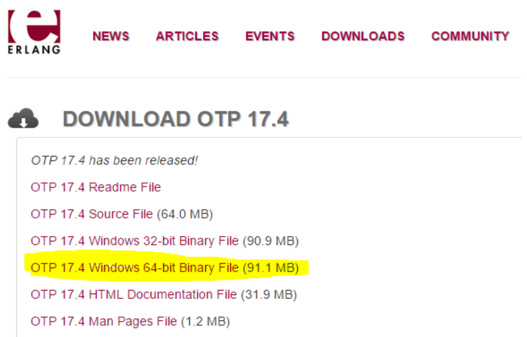
Install the Server

Firstly, download and run the Erlang Windows Binary File. It takes around 5 minutes.

Then just run the installer, `rabbitmq-server-3.4.4.exe`. It takes around 2 minutes, and will set RabbitMQ up and running as a service, with a default configuration.

Figure 9: Elang run time download link

When you are on the Erlang website, pick the latest version of the run time that matches your OS. If you are running a 64-bit OS, then pick the 64-bit version and vice versa with the 32-bit version.

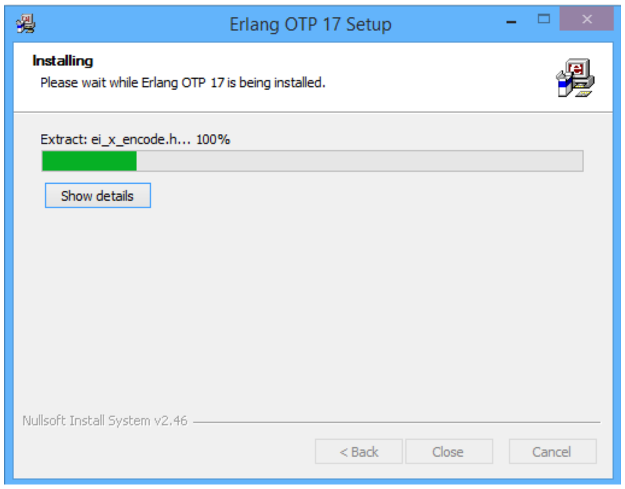


DOWNLOAD OTP 17.4

OTP 17.4 has been released!
OTP 17.4 Readme File
OTP 17.4 Source File (64.0 MB)
OTP 17.4 Windows 32-bit Binary File (90.9 MB)
OTP 17.4 Windows 64-bit Binary File (91.1 MB)
OTP 17.4 HTML Documentation File (31.9 MB)
OTP 17.4 Man Pages File (1.2 MB)

Figure 10: Download the latest version of the Erlang run time

When the installer file has downloaded, start it running. With this installer, you can stick with the defaults as they are all sensible defaults. Click through the menus so that all of the files get installed:

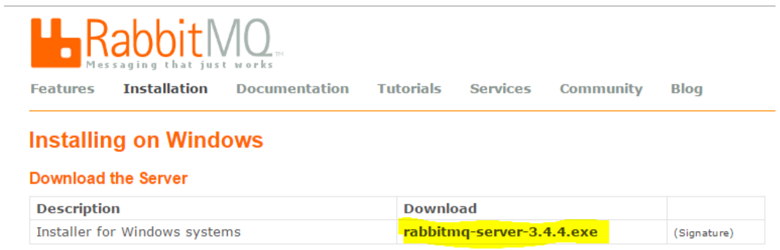

Erlang OTP 17 Setup

Installing
Please wait while Erlang OTP 17 is being installed.

Extract: ei_x_encode.h... 100%

Figure 11: Accept all of the defaults in the installer and let it run through

Once the Erlang run time has installed, go back to the RabbitMQ website and download the latest copy of the RabbtMQ server. This installer is small so it will not take long to download:



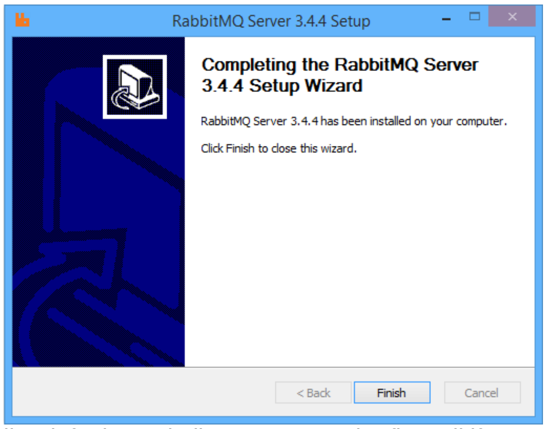
Installing on Windows

Download the Server

Description	Download	
Installer for Windows systems	rabbitmq-server-3.4.4.exe	(Signature)

Figure 12: Download the latest version of the Windows RabbitMQ server installation files

Once this is downloaded, run the installer. Again, as with the Erlang run time, you can stay with the defaults. During the installation, a firewall popup menu may appear and may ask you to set some firewall rules. If this happens, click “Allow Access”:



Completing the RabbitMQ Server 3.4.4 Setup Wizard

RabbitMQ Server 3.4.4 has been installed on your computer.

Click Finish to close this wizard.

Figure 13: Use the installer defaults and allow access to the firewall if you get any pop-up menus

RabbitMQ installs with a usable default message broker environment but you can change and customize the environment if you wish. This is beyond the scope of this book but further instructions for doing so are on the RabbitMQ website.

Once RabbitMQ has installed, the RabbitMQ service will automatically start up. This means you have a running message broker ready to go.

Setting Up the Management Portal

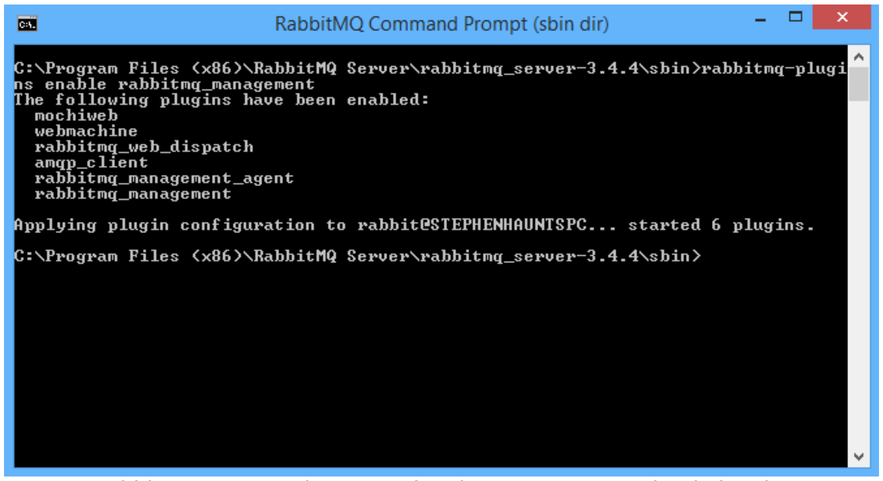
RabbitMQ comes with a web-based management portal. This portal serves as a place to configure RabbitMQ and also as a useful dashboard to monitor what is happening with the message broker. The management portal is not enabled by default; you need to do this separately.

Open up a command prompt and navigate to: “C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.4.4\sbin

From the command prompt, type:

```
rabbitmq-plugins enable rabbitmq_management
```

This will configure the web management portal. You should see the following in your command prompt once the portal is configured:



```
C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.4.4\sbin>rabbitmq-plugi
ns enable rabbitmq_management
The following plugins have been enabled:
  mochiweb
  webmachine
  rabbitmq_web_dispatch
  amqp_client
  rabbitmq_management_agent
  rabbitmq_management

Applying plugin configuration to rabbit@STEPHENHAUNTSPC... started 6 plugins.

C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.4.4\sbin>
```

Figure 14: RabbitMQ command prompt after the management plug-in has been set up

If you are following the examples in this book, then you should keep this command-line window open as you will need to use it in the next chapter.

To go to the portal, use the following web address where “server-name” is substituted with the name of the server containing the instance of RabbitMQ:

> http://server-name:15672

If you are running this on your local machine, then the address will be http://localhost:15672.

Once you go to this address, you will see a RabbitMQ logon screen like in the following screenshot:

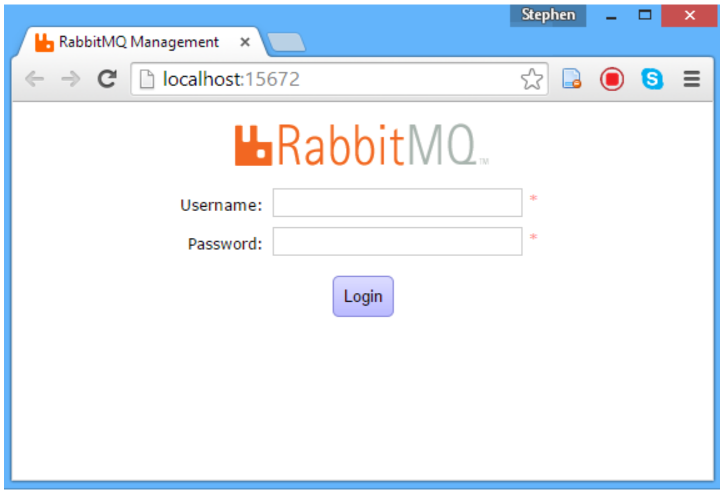


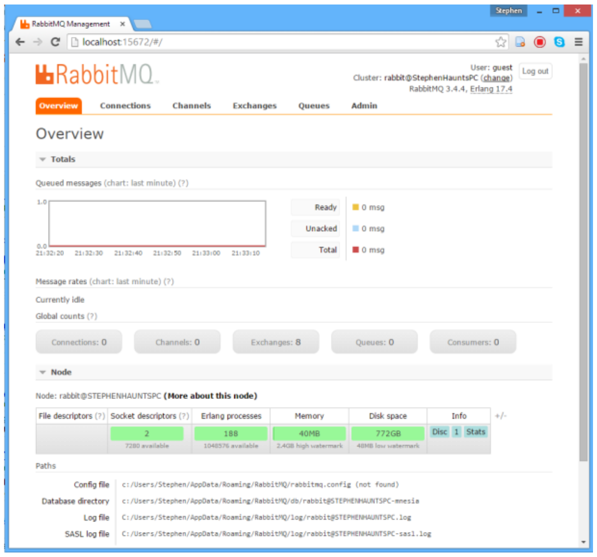
Figure 15: RabbitMQ mangement portal logon screen

To log onto the management portal, the default credentials are:

Username: guest

Password: guest

Once you log on, you will see the following overview screen. This page gives you a dashboard view of RabbitMQ running on that server where you can see how many message are flowing through the system, the message throughput rates, and the numbers of exchanges, queues, and consumers:



User: guest
Cluster: rabbit@StephenHauntsPC (change)
RabbitMQ 3.4.4, Erlang 17.4

Overview

Totals

Queued messages (chart: last minute) (?)

Ready	🟨 0 msg
Unacked	🟦 0 msg
Total	🟥 0 msg

Message rates (chart: last minute) (?)

Currently idle

Global counts (?)

Connections: 0 | Channels: 0 | Exchanges: 8 | Queues: 0 | Consumers: 0

Node

Node: rabbit@STEPHENHAUNTSPC **(More about this node)**

File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Info
	2 7280 available	188 1048576 available	40MB 2.4GB high watermark	772GB 48MB low watermark	Disc 1 Stats

Paths

Config file	c:/Users/Stephen/AppData/Roaming/RabbitMQ/rabbitmq.config (not found)
Database directory	c:/Users/Stephen/AppData/Roaming/RabbitMQ/db/rabbit@STEPHENHAUNTSPC-mnesia
Log file	C:/Users/Stephen/AppData/Roaming/RabbitMQ/log/rabbit@STEPHENHAUNTSPC.log
SASL log file	C:/Users/Stephen/AppData/Roaming/RabbitMQ/log/rabbit@STEPHENHAUNTSPC-sasl.log

Figure 16: RabbitMQ overview dashboard

Chapter 5 Overview of the Management Plug-in

The RabbitMQ management plug-in provides a browser-based UI to administer the message broker as well as a HTTP-based API for the management and monitoring of your RabbitMQ server.

The management plug-in features include:

- Declare, list, and delete exchanges, queues, bindings, users, virtual hosts, and permissions.
- Monitor queue length, message rates (globally and per channel), and data rates per connection, etc.
- Send and receive messages.
- Monitor Erlang processes, file descriptors, and memory use.
- Export/import object definitions to JSON.
- Force close connections, and purge queues.

The management portal is split into different screens that are selectable, with the menu bar at the top of the screen. The first screen you will see is the overview page. This acts as a dashboard view showing you how many messages are in the broker and the message throughput rate.

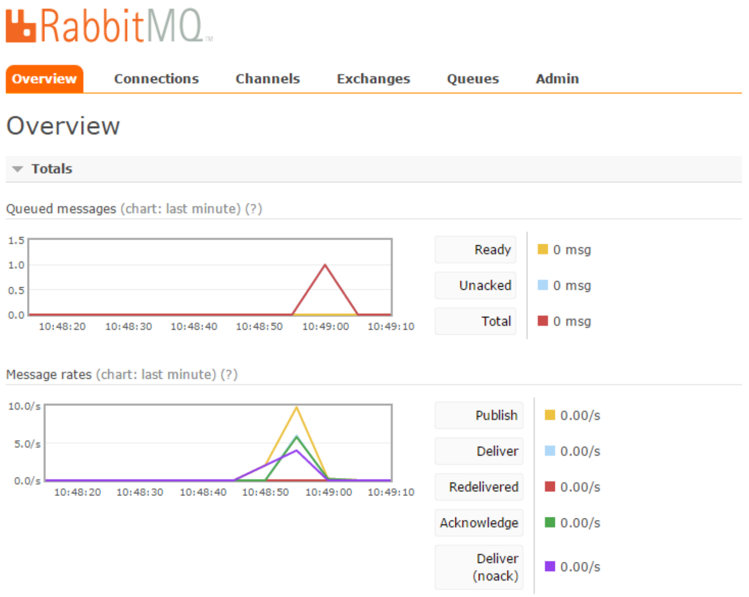
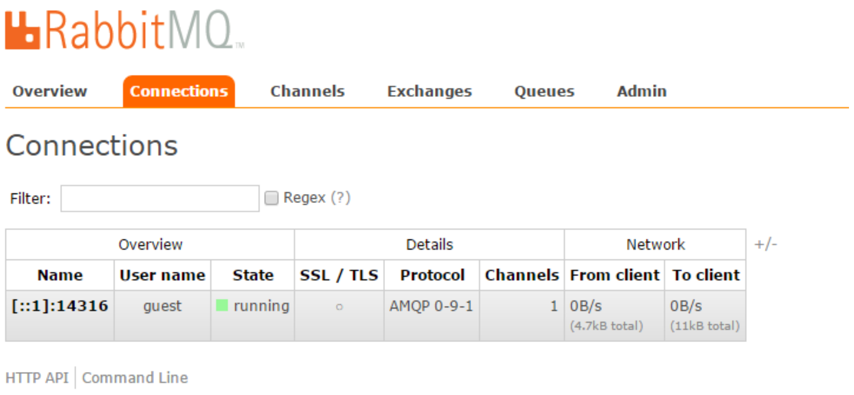


Figure 17: RabbitMQ management portal overview screen

This screen is useful to have up on a large display so you can see at a glance how RabbitMQ is performing. If, for example, any of your message consumer applications go down, you will start to see a large buildup of undelivered messages which is a good indication that something has gone wrong.

In Figure 17, you can see on the overview screen that one message was placed onto the queue at 10:49.00. In the chart below, you can see the message rates; they are split into Published messages, Delivered messages, Redelivered messagaes, Acknowledged messages, and Delivered messagaes where no acknowledgement was required.

The next tab in the management portal is the connections page. When you have consumer applications connecting to RabbitMQ to process messages, they will show up here as connections. A connection represents a real Transmission Control Protocol (TCP) connection to the message broker whereas a channel, which we will see in a moment, is virtual connection inside the connection. This means you can use as many virtual connections (i.e., channels) as you want inside your application without overloading the message broker with TCP connections:



Connections

Filter: Regex (?)

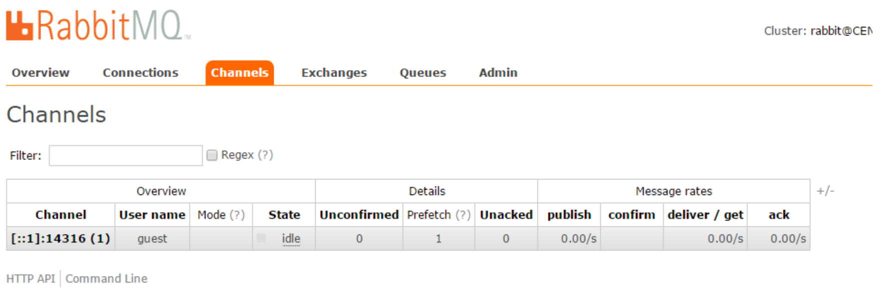
Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
[::1]:14316	guest	🟩 running	○	AMQP 0-9-1	1	0B/s (4.7kB total)	0B/s (11kB total)

+/-

HTTP API | Command Line

Figure 18: RabbitMQ management portal connections page

The next screen in the management portal is a screen to view the active message broker channels. Some applications need multiple connections to a RabbitMQ broker server. However, it is undesirable to keep many TCP connections open at the same time because doing so consumes system resources and makes it more difficult to configure firewalls. RabbitMQ connections are multiplexed with channels that can be thought of as "lightweight connections that share a single TCP connection."



Channels

Filter: Regex (?)

Overview				Details			Message rates			
Channel	User name	Mode (?)	State	Unconfirmed	Prefetch (?)	Unacked	publish	confirm	deliver / get	ack
[::1]:14316 (1)	guest		idle	0	1	0	0.00/s		0.00/s	0.00/s

+/-

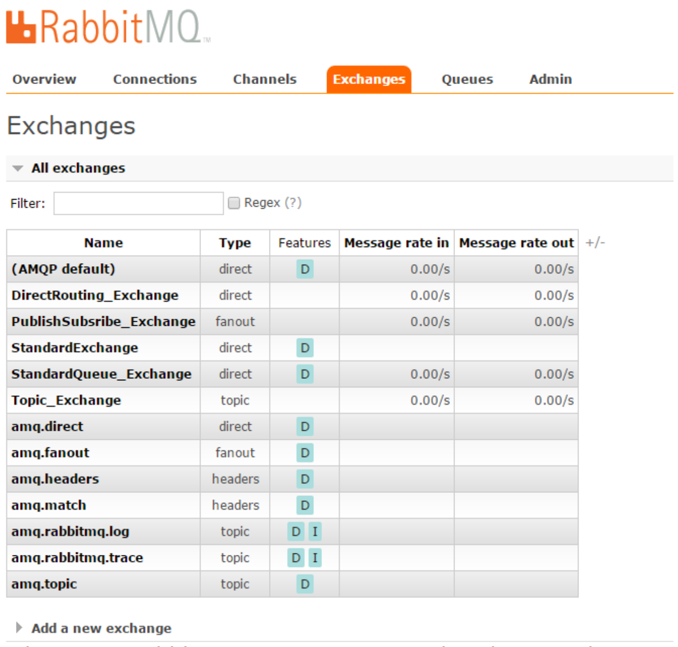
HTTP API | Command Line

Figure 19: RabbitMQ management portal channel view

For applications that use multiple threads/processes for processing, it is common to open a new channel per thread/process and not share channels between them.

Communication on a particular channel is completely separate from communication on another channel; therefore, every RabbitMQ method also carries a channel number that clients use to figure out which channel the method is for (and thus, which event handler needs to be invoked, for example).

The next screen in the management portal is the exchanges view. This gives you a list of all of the currently defined exchanges on the server, as well as the type of exchange and details such as indicating whether or not they are durable:



Name	Type	Features	Message rate in	Message rate out
(AMQP default)	direct	D	0.00/s	0.00/s
DirectRouting_Exchange	direct		0.00/s	0.00/s
PublishSubsribe_Exchange	fanout		0.00/s	0.00/s
StandardExchange	direct	D		
StandardQueue_Exchange	direct	D	0.00/s	0.00/s
Topic_Exchange	topic		0.00/s	0.00/s
amq.direct	direct	D		
amq.fanout	fanout	D		
amq.headers	headers	D		
amq.match	headers	D		
amq.rabbitmq.log	topic	D I		
amq.rabbitmq.trace	topic	D I		
amq.topic	topic	D		

Figure 20: RabbitMQ management portal exchanges view

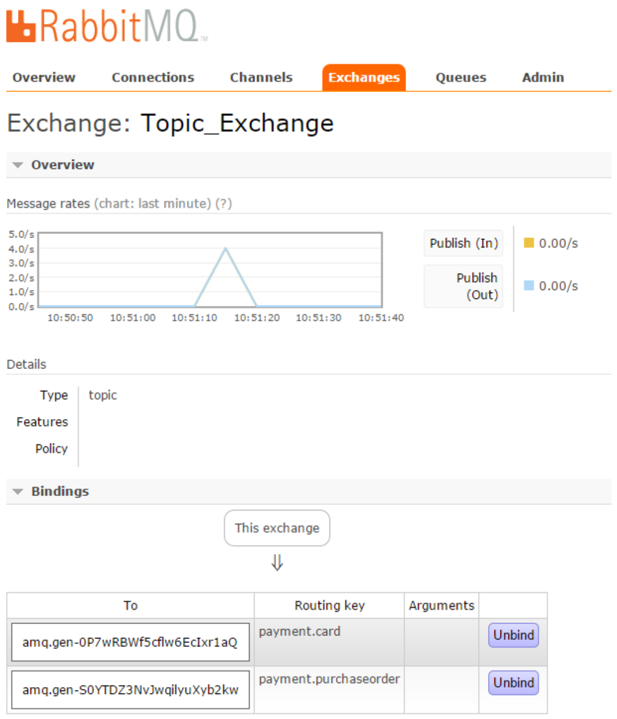
Exchanges are AMQP entities where messages are sent in the message broker. Exchanges take a message and then route it to one or more queues. The type of routing depends upon the exchange type used and different exchange rules called bindings.

Each exchange is also declared with a number of different attributes. The most important attributes to us are:

1. **Name**: The name of the exchange.
2. **Durability**: This flag determines whether or not messages sent to the exchange survive a broker/server restart by persisting the messages to disk.
3. **Auto-delete**: This flag determines if the exchange is deleted when all of the queues have finished using it.
4. **Arguments**: These are message broker-dependent.

The types of exchanges supported are Direct, Fanout, Topic, and Headers. These were previously discussed in the chapter on the AMQP messaging standard.

If you click any of the exchanges in the list, you will be taken to a screen that shows more information about that particular exchange. This includes information about the messaging rate for the exchange and the binding information.



Exchange: Topic_Exchange

Overview

Message rates (chart: last minute) (?)

Publish (In)	🟨 0.00/s
Publish (Out)	🟦 0.00/s

Details

Type	topic
Features	
Policy	

Bindings

This exchange

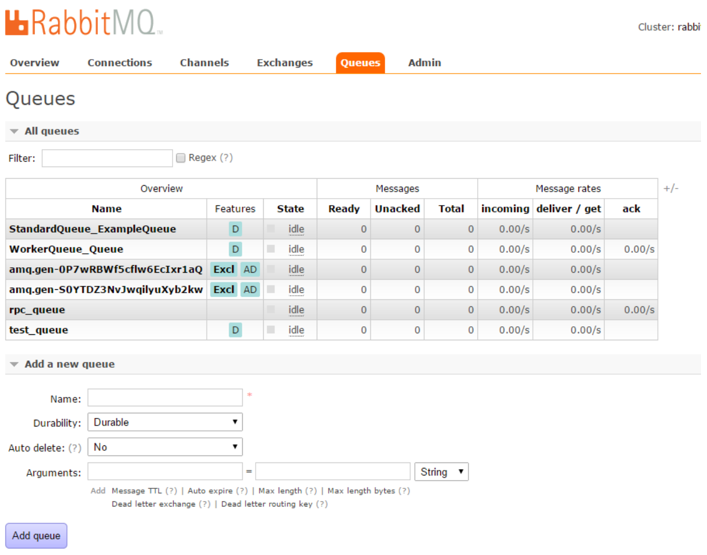
⇓

To	Routing key	Arguments	
amq.gen-0P7wRBWf5cflw6EcIxr1aQ	payment.card		Unbind
amq.gen-S0YTDZ3NvJwqilyuXyb2kw	payment.purchaseorder		Unbind

Figure 21: RabbitMQ messaging portal exchange view

In the preceding screenshot, you can see details of the bindings for this exchange. For the exchange named "Topic_Exchange," there are two queues that are bound to it. One of the queues has a routing key of "payment.card" and the other queue has a routing key of "payment.purchaseorder." From this screen, you can manually bind other queues to the exchange or unbind current queues.

The next screen in the management portal is the queues view. This screen lists all of the queues that are defined on the server. From this screen, you can also add new queues. This screen is useful as a dashboard to show you what is going on at the queue level. You can see when queues are idle and the status of messages in the queue (such as whether or not they are ready for consumption, and whether they have been acknowledged, etc.):



Queues

All queues

Filter: Regex (?)

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
StandardQueue_ExampleQueue	D	idle	0	0	0	0.00/s	0.00/s	
WorkerQueue_Queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
amq.gen-0P7wRBWf5cflw6EcIxr1aQ	Excl AD	idle	0	0	0	0.00/s	0.00/s	
amq.gen-S0YTDZ3NvJwqilyuXyb2kw	Excl AD	idle	0	0	0	0.00/s	0.00/s	
rpc_queue		idle	0	0	0	0.00/s	0.00/s	0.00/s
test_queue	D	idle	0	0	0	0.00/s	0.00/s	

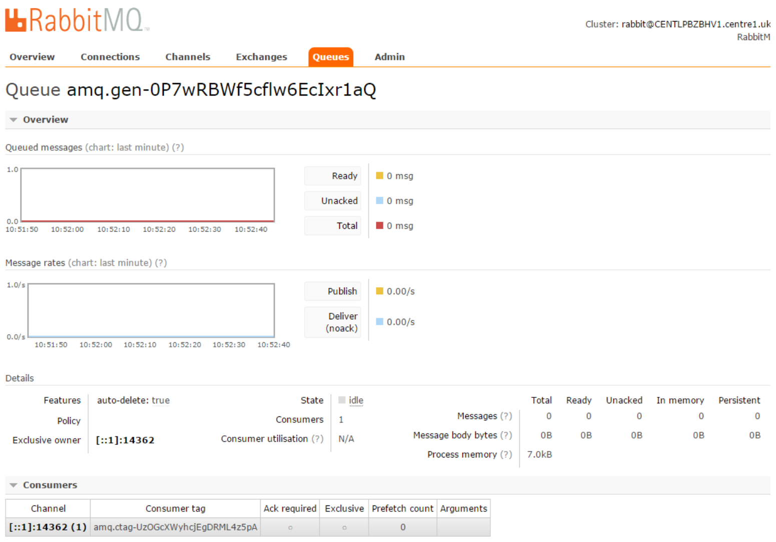
Add a new queue

Name: *
Durability: Durable
Auto delete: (?) No
Arguments: = String
Add Message TTL (?) | Auto expire (?) | Max length (?) | Max length bytes (?)
Dead letter exchange (?) | Dead letter routing key (?)

Add queue

Figure 22: RabbitMQ management portal queues view

Just as with the exchanges screen, you can also click any individual queue to see that status of that individual queue. This will give you a view similar to that seen in Figure 23 (in the following screenshot) where you can look at the message rate and what exchanges that queue is bound to:



Queue amq.gen-0P7wRBWf5cflw6EcIxr1aQ

Overview

Queued messages (chart: last minute) (?)

Ready 0 msg
Unacked 0 msg
Total 0 msg

Message rates (chart: last minute) (?)

Publish 0.00/s
Deliver (noack) 0.00/s

Details

Features auto-delete: true
Policy
Exclusive owner [::1]:14362

State idle
Consumers 1
Consumer utilisation (?) N/A

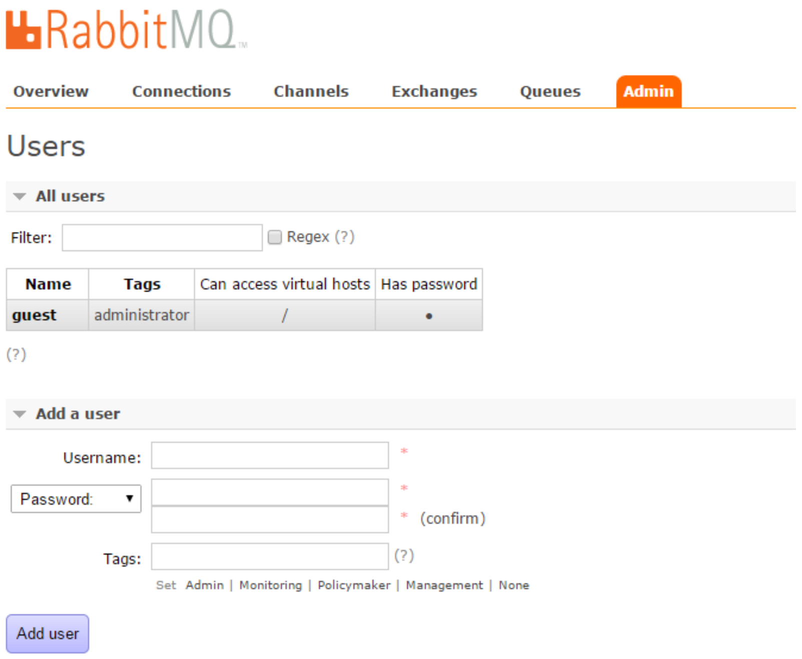
	Total	Ready	Unacked	In memory	Persistent
Messages (?)	0	0	0	0	0
Message body bytes (?)	0B	0B	0B	0B	0B
Process memory (?)	7.0kB				

Consumers

Channel	Consumer tag	Ack required	Exclusive	Prefetch count	Arguments
[::1]:14362 (1)	amq.ctag-UzOGcXWyhcjEgDRML4z5pA	○	○	0	

Figure 23: RabbitMQ management portal queue view

The final view we will look at is the Admin view. This is where users and their permissions are administered. By default, when RabbitMQ is installed, there is a guest user defined which we used to log onto the management portal. It is not a good idea to carry on using this user. In fact, by default, you can only access RabbitMQ via localhost with this user. It is advisable to delete this user and set up additional users:



Users

All users

Filter: Regex (?)

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/	•

(?)

Add a user

Username: *

Password: *

* (confirm)

Tags: (?)

Set Admin | Monitoring | Policymaker | Management | None

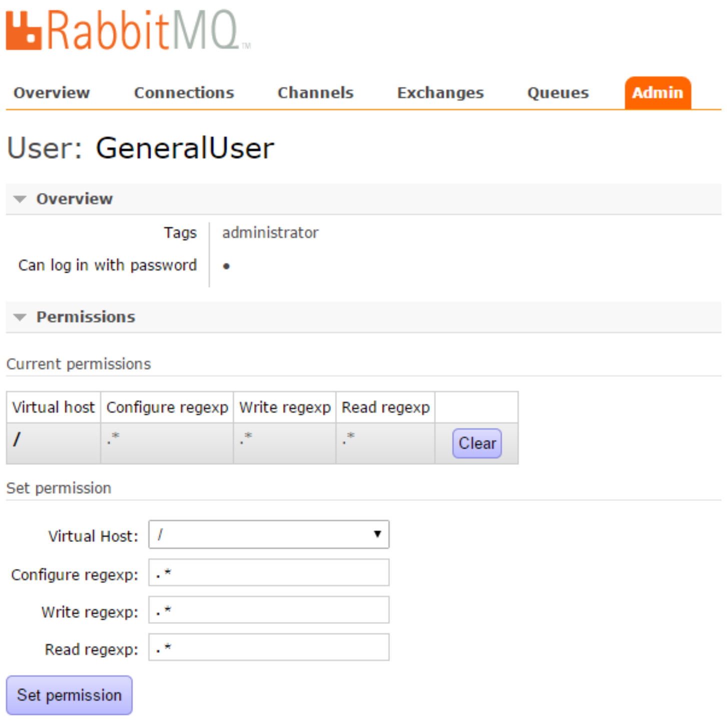
Add user

Figure 24: RabbitMQ management portal user administration

To create a new user, go to the “Add a user” section on the screen as shown in the preceding screenshot, and add a new username and password. You can then set a series of comma-separated lists of tags to apply to the user. The tags that are currently supported by the management plug-in are:

- **Management**: User can access the management plug-in.
- **Policymaker**: User can access the management plug-in, and manage policies and parameters for the vhosts to which they have access.
- **Monitoring**: User can access the management plug-in and see all of the connections and channels as well as node-related information.
- **Administrator**: User can do everything monitoring can do, and also manage users, vhosts, and permissions; close other users’ connections, and manane policies and parameters for all vhosts.

You can set any tag here but the above four tags are just for convenience. Once a user has been created, you have to then set permissions for that user. When a RabbitMQ client connects to a RabbitMQ server, it specifies a virtual host within which it intends to operate. A first level of access control is enforced at this point, with the server checking whether or not the user has any permission to access the virtual hosts, and rejecting the connection attempt otherwise:



User: GeneralUser

Overview

Tags	administrator
Can log in with password	●

Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
/	.*	.*	.*	Clear

Set permission

Virtual Host:	/
Configure regexp:	.*
Write regexp:	.*
Read regexp:	.*

Figure 25: RabbitMQ management portal user permissions management

For the purposes of the examples in this book, we will be using the default virtual host. For more information about virtual hosts and their permissions, you can read more on Access Control on the RabbitMQ website.

Chapter 6 Administration via the Command Line

In addition to using the web-based management portal to administer RabbitMQ, you can also use the command-line interface (**rabbitmqctrl.bat)**. In this chapter, we will demonstrate some of the basic features that you may need to most frequently use, but for a more exhaustive list of commands, you can read the RabbitMQ manual page for the **rabbitmqctrl.bat** tool.

At a high level, **rabbitmqctrl** lets you manage the run state of the message broker, manage your RabbitMQ clusters, administer users and permissions, and manage policies and list exchanges, bindings, and queues.

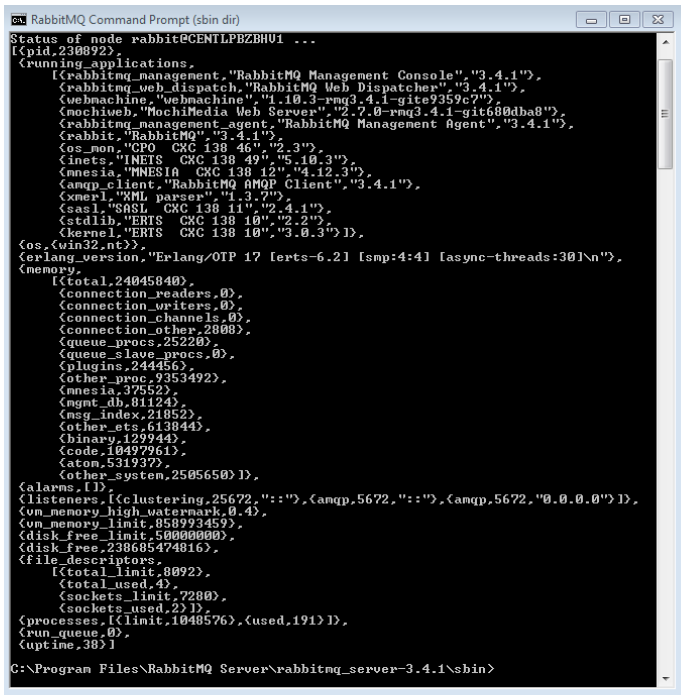
Let's work through a simple example of stopping and starting the RabbitMQ broker and checking the broker status.

Open up a command prompt and navigate to "C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.4.4\sbin

From the command prompt, type:

```
Rabbitmqctl status
```

You will see the following output in the command-line window:



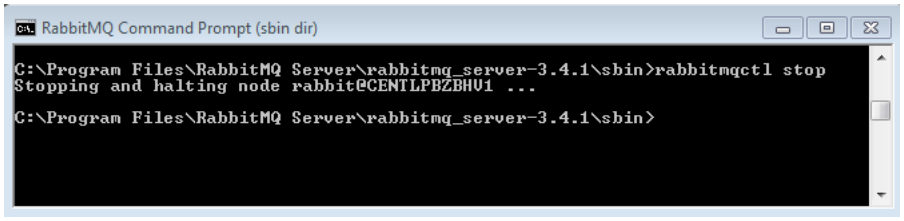
```
Status of node rabbit@CENTLPBZBHV1 ...
[{pid,230892},
 {running_applications,
     [{rabbitmq_management,"RabbitMQ Management Console","3.4.1"},
      {rabbitmq_web_dispatch,"RabbitMQ Web Dispatcher","3.4.1"},
      {webmachine,"webmachine","1.10.3-rmq3.4.1-gite9359c7"},
      {mochiweb,"MochiMedia Web Server","2.7.0-rmq3.4.1-git680dba8"},
      {rabbitmq_management_agent,"RabbitMQ Management Agent","3.4.1"},
      {rabbit,"RabbitMQ","3.4.1"},
      {os_mon,"CPO  CXC 138 46","2.3"},
      {inets,"INETS  CXC 138 49","5.10.3"},
      {mnesia,"MNESIA  CXC 138 12","4.12.3"},
      {amqp_client,"RabbitMQ AMQP Client","3.4.1"},
      {xmerl,"XML parser","1.3.7"},
      {sasl,"SASL  CXC 138 11","2.4.1"},
      {stdlib,"ERTS  CXC 138 10","2.2"},
      {kernel,"ERTS  CXC 138 10","3.0.3"}]},
 {os,{win32,nt}},
 {erlang_version,"Erlang/OTP 17 [erts-6.2] [smp:4:4] [async-threads:30]\n"},
 {memory,
     [{total,24045840},
      {connection_readers,0},
      {connection_writers,0},
      {connection_channels,0},
      {connection_other,2808},
      {queue_procs,25220},
      {queue_slave_procs,0},
      {plugins,244456},
      {other_proc,9353492},
      {mnesia,37552},
      {mgmt_db,81124},
      {msg_index,21852},
      {other_ets,613844},
      {binary,129944},
      {code,10497961},
      {atom,531937},
      {other_system,2505650}]},
 {alarms,[]},
 {listeners,[{clustering,25672,"::"},{amqp,5672,"::"},{amqp,5672,"0.0.0.0"}]},
 {vm_memory_high_watermark,0.4},
 {vm_memory_limit,858993459},
 {disk_free_limit,50000000},
 {disk_free,238685474816},
 {file_descriptors,
     [{total_limit,8092},
      {total_used,4},
      {sockets_limit,7280},
      {sockets_used,2}]},
 {processes,[{limit,1048576},{used,191}]},
 {run_queue,0},
 {uptime,38}]

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>
```

Figure 26: RabbitMQ status

To stop the RabbitMQ broker from running, you type the following into the command line:

```
Rabbitmqctl stop
```

This will give you console output that looks as follows:



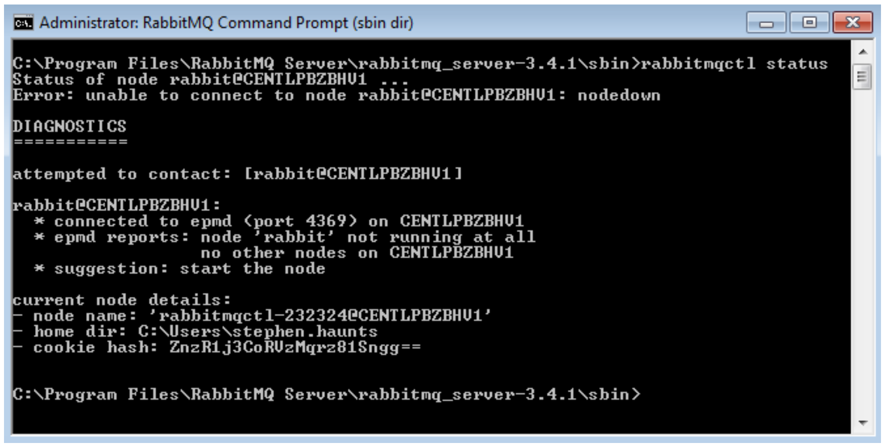
```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>rabbitmqctl stop
Stopping and halting node rabbit@CENTLPBZBHV1 ...

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>
```

Figure 27: RabbitMQ stopped service

If you run the status command line again by typing the following, you will see that the RabbitMQ service has stopped:

```
Rabbitmqctl status
```

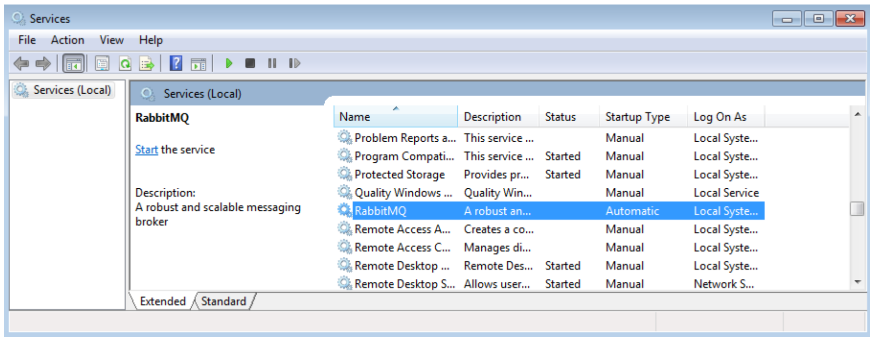
This will mean RabbitMQ will stop receiving and processing messages. If you have not set up durable queues and messages, you will lose any messages already in the system:



```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>rabbitmqctl status
Status of node rabbit@CENTLPBZBHV1 ...
Error: unable to connect to node rabbit@CENTLPBZBHV1: nodedown

DIAGNOSTICS
===========

attempted to contact: [rabbit@CENTLPBZBHV1]

rabbit@CENTLPBZBHV1:
  * connected to epmd (port 4369) on CENTLPBZBHV1
  * epmd reports: node 'rabbit' not running at all
                  no other nodes on CENTLPBZBHV1
  * suggestion: start the node

current node details:
- node name: 'rabbitmqctl-232324@CENTLPBZBHV1'
- home dir: C:\Users\stephen.haunts
- cookie hash: ZnzR1j3CoRVzMqrz81Sngg==

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>
```

Figure 28: The RabbitMQ service has been stopped

To restart the service, you can either go to the Windows services management window in Microsoft Windows (as shown in the following screenshot) or you can use the **rabbitmqctl** service command-line tool:



RabbitMQ

Start the service

Description:
A robust and scalable messaging broker

Name	Description	Status	Startup Type	Log On As
Problem Reports a...	This service ...		Manual	Local Syste...
Program Compati...	This service ...	Started	Manual	Local Syste...
Protected Storage	Provides pr...	Started	Manual	Local Syste...
Quality Windows ...	Quality Win...		Manual	Local Service
RabbitMQ	A robust an...		Automatic	Local Syste...
Remote Access A...	Creates a co...		Manual	Local Syste...
Remote Access C...	Manages di...		Manual	Local Syste...
Remote Desktop ...	Remote Des...	Started	Manual	Local Syste...
Remote Desktop S...	Allows user...	Started	Manual	Network S...

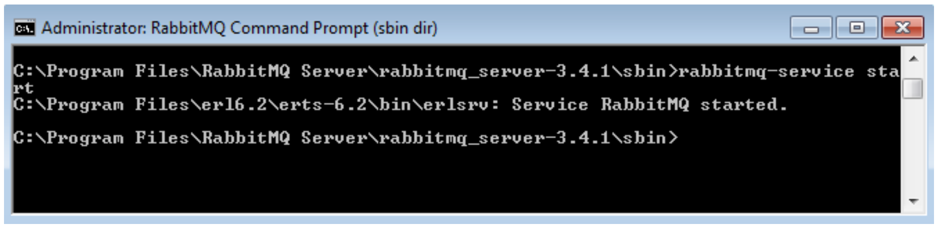
Figure 29: Start and stop the RabbitMQ services from the services window in Windows

To start the service from the command line, you need to type:

```
rabbitmq-service start
```

Note: Please note that the command to start the RabbitMQ Service is different from the command to stop the service (rabbitmqctl).

This will restart the service as shown in the following screenshot:



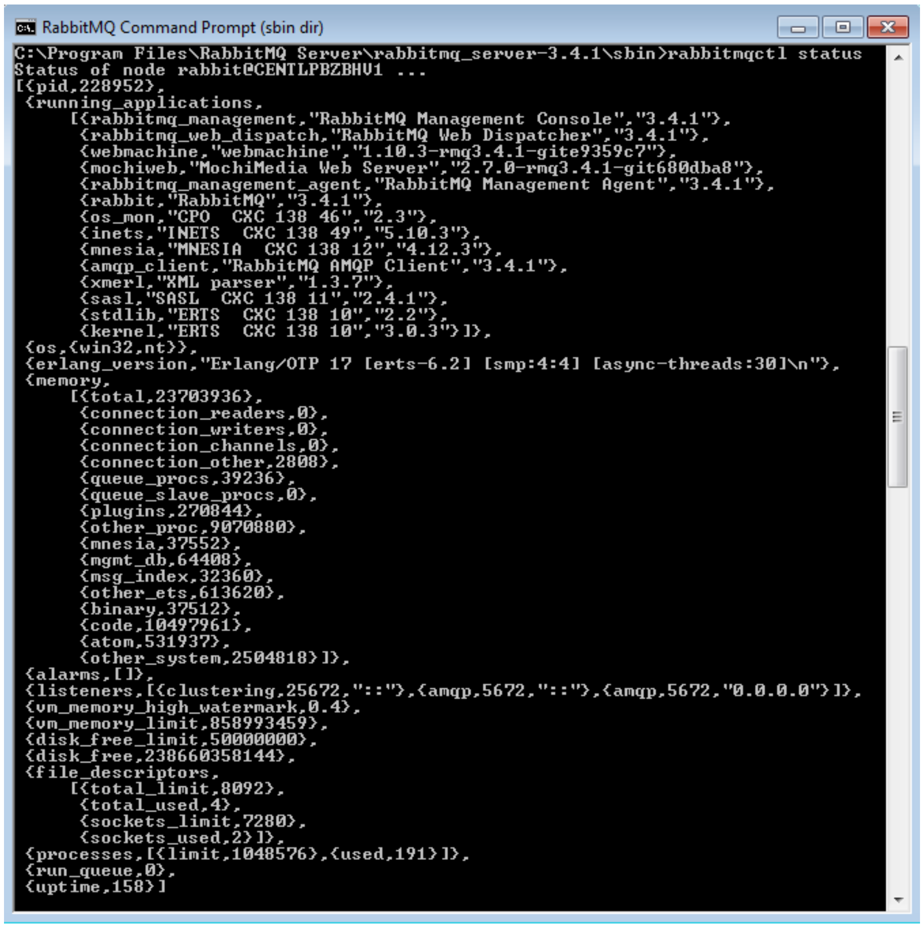
```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>rabbitmq-service sta
rt
C:\Program Files\erl6.2\erts-6.2\bin\erlsrv: Service RabbitMQ started.

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>
```

Figure 30: The RabbitMQ service has been restarted

If you get a “permission denied” error, then you will need to make sure you open the command-line window as an administrator. You can now query the status of the service again by using:

```
Rabbitmqctl status
```

You will now see that the service is up and running again.



```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.4.1\sbin>rabbitmqctl status
Status of node rabbit@CENTLPBZBHV1 ...
[{pid,228952},
 {running_applications,
     [{rabbitmq_management,"RabbitMQ Management Console","3.4.1"},
      {rabbitmq_web_dispatch,"RabbitMQ Web Dispatcher","3.4.1"},
      {webmachine,"webmachine","1.10.3-rmq3.4.1-gite9359c7"},
      {mochiweb,"MochiMedia Web Server","2.7.0-rmq3.4.1-git680dba8"},
      {rabbitmq_management_agent,"RabbitMQ Management Agent","3.4.1"},
      {rabbit,"RabbitMQ","3.4.1"},
      {os_mon,"CPO  CXC 138 46","2.3"},
      {inets,"INETS  CXC 138 49","5.10.3"},
      {mnesia,"MNESIA  CXC 138 12","4.12.3"},
      {amqp_client,"RabbitMQ AMQP Client","3.4.1"},
      {xmerl,"XML parser","1.3.7"},
      {sasl,"SASL  CXC 138 11","2.4.1"},
      {stdlib,"ERTS  CXC 138 10","2.2"},
      {kernel,"ERTS  CXC 138 10","3.0.3"}]},
 {os,{win32,nt}},
 {erlang_version,"Erlang/OTP 17 [erts-6.2] [smp:4:4] [async-threads:30]\n"},
 {memory,
     [{total,23703936},
      {connection_readers,0},
      {connection_writers,0},
      {connection_channels,0},
      {connection_other,2808},
      {queue_procs,39236},
      {queue_slave_procs,0},
      {plugins,270844},
      {other_proc,9070880},
      {mnesia,37552},
      {mgmt_db,64408},
      {msg_index,32360},
      {other_ets,613620},
      {binary,37512},
      {code,10497961},
      {atom,531937},
      {other_system,2504818}]},
 {alarms,[]},
 {listeners,[{clustering,25672,"::"},{amqp,5672,"::"},{amqp,5672,"0.0.0.0"}]},
 {vm_memory_high_watermark,0.4},
 {vm_memory_limit,858993459},
 {disk_free_limit,50000000},
 {disk_free,238660358144},
 {file_descriptors,
     [{total_limit,8092},
      {total_used,4},
      {sockets_limit,7280},
      {sockets_used,2}]},
 {processes,[{limit,1048576},{used,191}]},
 {run_queue,0},
 {uptime,158}]
```

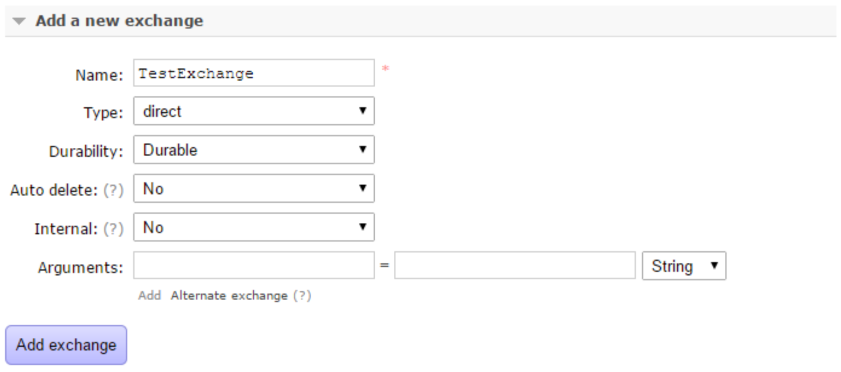
Figure 31: The RabbitMQ service is running again

This should have given you a flavor of administering RabbitMQ from the command line. For more information on the commands available, you can read the manual pages for abbitmq-service here and rabbitmqctl here.

Chapter 7 Basic Queue and Message Example

Before we dive into some code and look at our samples, let's work through a simple scenario in which we create an exchange and a queue, and bind them together via the management portal. We will then send a message to the exchange and pull it from the queue. It is a simple example but it serves as a good introduction before we tackle some real-world scenarios.

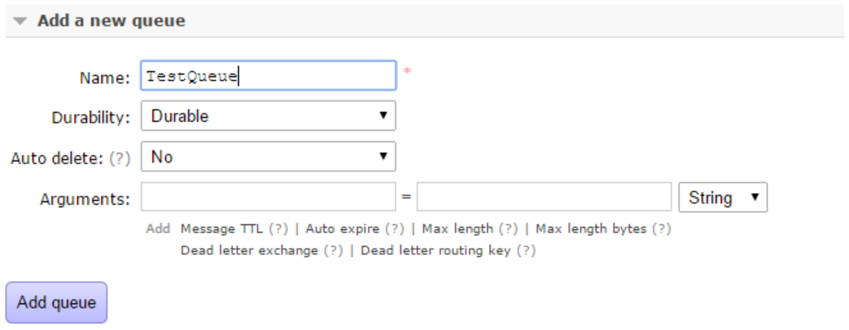
First, go to the management portal and click the exchanges tab. Once you are on the exchanges page, open "Add a new exchange" and fill it in as shown in the following screenshot. You will then need to click "Add exchange" to add the exchange:


Add a new exchange

Name:	TestExchange
Type:	direct
Durability:	Durable
Auto delete: (?)	No
Internal: (?)	No
Arguments:	= String

Figure 32: Create a new direct exchange

This will add a new "direct" exchange to the list of exchanges. Now, click the Queues tab at the top of the page to go to the queue list. Open "Add a new queue" and fill it in as per the following screenshot. Now, click the "Add queue" button:

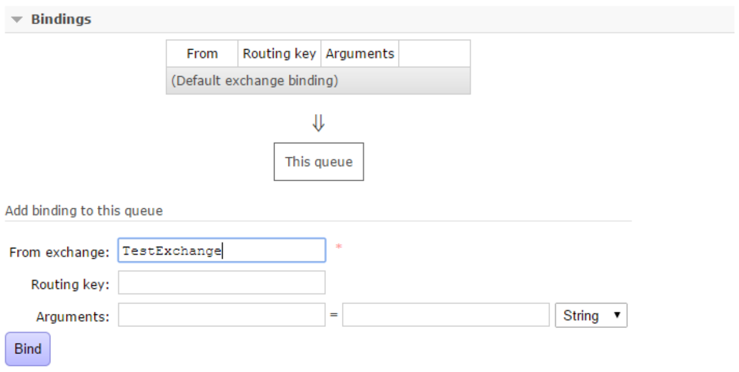

Add a new queue

Name:	TestQueue
Durability:	Durable
Auto delete: (?)	No
Arguments:	= String

Add Message TTL (?) | Auto expire (?) | Max length (?) | Max length bytes (?)
Dead letter exchange (?) | Dead letter routing key (?)
Add queue

Figure 33: Create a new test queue

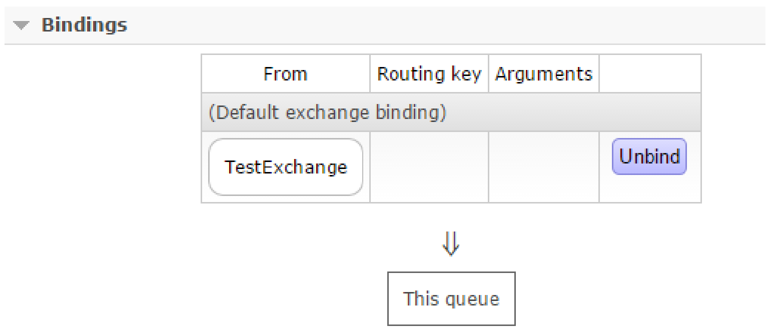
You will see this TestQueue appear in the queues list. Now, click "TestQueue" in that list. This will take you to the page for that specific queue. If you scroll down to the bindings section and open it, you will see (as per the following screenshot) that this queue is not bound any exchanges:



From exchange: TestExchange

Figure 34: Binding a queue to an exchange

To bind the queue to our TestExchange, type the name of the exchange into the "From exchange" text box and then hit "Bind." We will not put in a routing key for this example. You will now notice that this queue is bound to the TestExchange exchange:

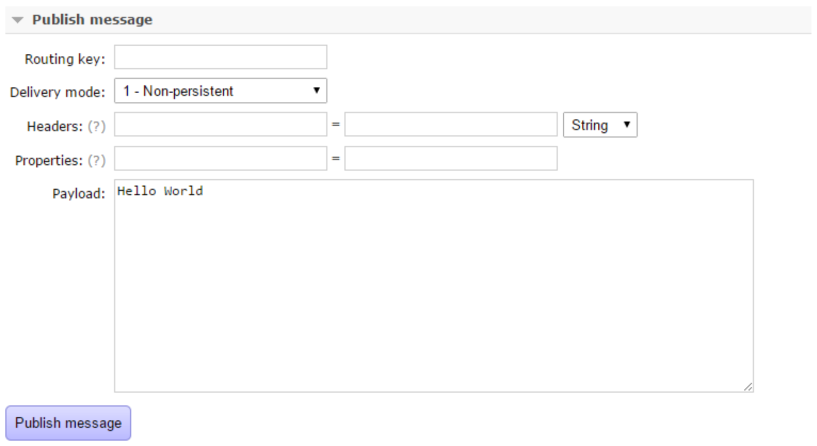


From	Routing key	Arguments	
(Default exchange binding)			
TestExchange			Unbind

Figure 35: The queue is now bound to the exchange

Now, go back to the exchanges tab in the management portal and scroll down to "Publish message." This is in a collapsible section so you may need to open it up. You should see the publish message options as shown in Figure 36.

We are not using a routing key in this example so leave that blank. In the payload section, write a message. In the screenshot, we are using our traditional Hello World sample:



Publish message

Routing key:

Delivery mode: 1 - Non-persistent

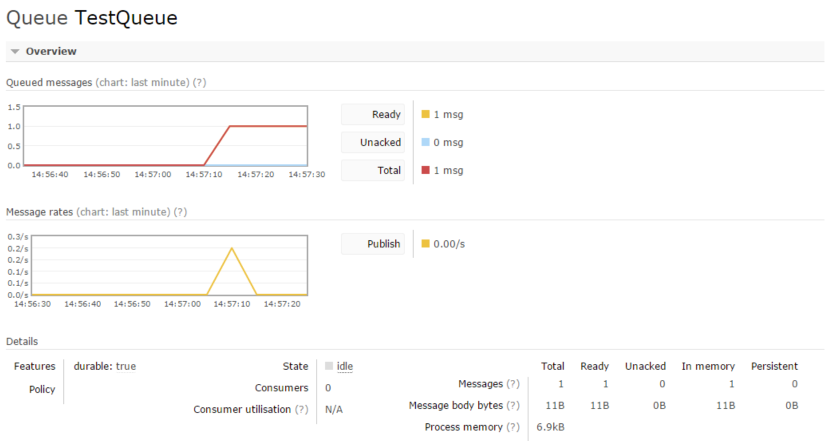
Headers: (?) = String

Properties: (?) =

Payload: Hello World

Figure 36: Publishing a simple test message to the exchange

Now, go back to “Queues” and select our TestQueue. You will notice that the chart is showing one queued message:



Queue TestQueue

Overview

Queued messages (chart: last minute) (?)

Ready	🟨 1 msg
Unacked	🟦 0 msg
Total	🟥 1 msg

Message rates (chart: last minute) (?)

Publish	🟨 0.00/s

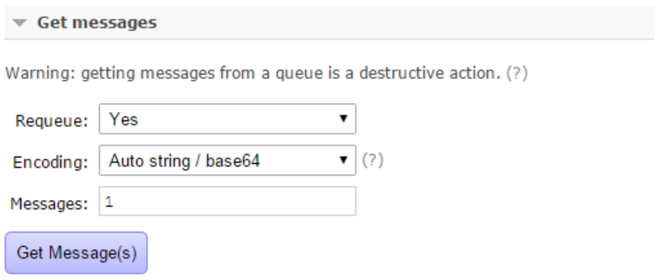
Details

Features	durable: true
Policy	
State	idle
Consumers	0
Consumer utilisation (?)	N/A

	Total	Ready	Unacked	In memory	Persistent
Messages (?)	1	1	0	1	0
Message body bytes (?)	11B	11B	0B	11B	0B
Process memory (?)	6.9kB				

Figure 37: The queue shows that a message has been queued

Now, scroll down to the bottom of that screen and open “Get messages.”



Get messages

Warning: getting messages from a queue is a destructive action. (?)

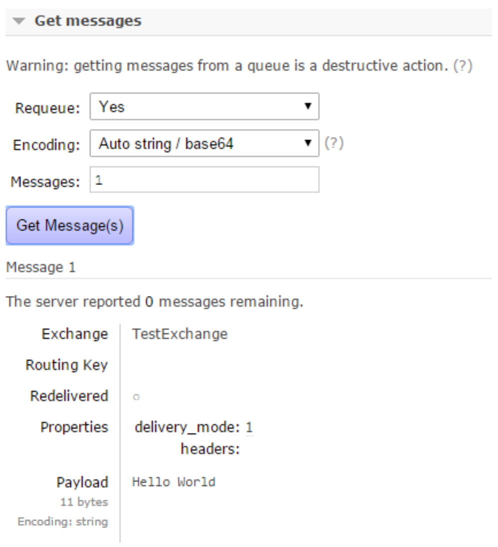
Requeue: Yes

Encoding: Auto string / base64 (?)

Messages: 1

Figure 38: Getting a message from the queue

If you click “Get Message(s),” you will now see how the message payload has now been removed from the queue:



Warning: getting messages from a queue is a destructive action. (?)

Requeue: Yes

Encoding: Auto string / base64 (?)

Messages: 1

Message 1

The server reported 0 messages remaining.

Exchange	TestExchange
Routing Key	
Redelivered	○
Properties	delivery_mode: 1 headers:
Payload 11 bytes Encoding: string	Hello World

Figure 39: The message has been retrieved but then requeued

If you look at the message chart at the top of the page, you will see that there is still a message in the queue. This is because, when we got the message, the “Requeue” parameter was set to “Yes.” This means that, once we have extracted the message, it was replaced back on the queue.

This time, set the “Requeue” parameter to “No” and get the message again. You will see the same message come back but, if you scroll back to the queues messages chart at the top of the screen, you will see that the message is gone and there are no messages left in the queue.

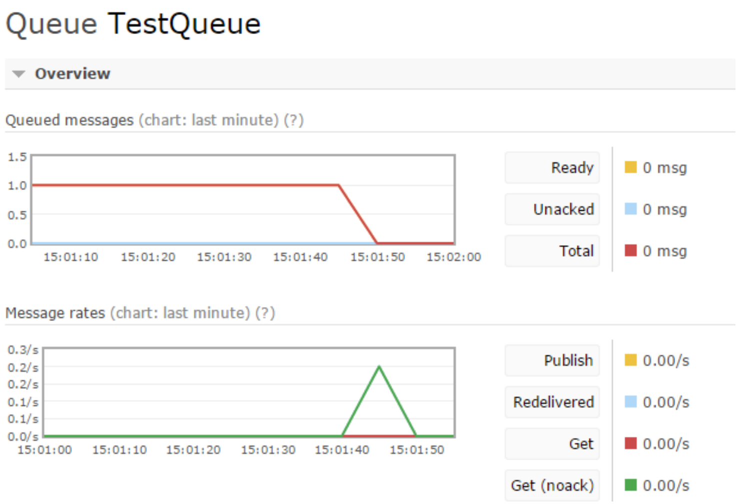


Figure 40: With requeuing turned off, the message disappears from the queue when it is retrieved

Although this is a simple example, you should now be at a point where you have a good understanding of what RabbitMQ is about, how to install and configure it, and how the basic process of setting up and binding exchanges and queues works. We are now ready to start looking at some real code.

Chapter 8 Working Examples

Now that we have covered a lot of the introductory material for RabbitMQ, this part of the book will look at developing software to interact with the message broker as both a producer and as a consumer. First, we will take a look at the RabbitMQ client library. Then, we will introduce the business scenario used for the sample applications. Before we start looking at the individual examples, we will take a quick look at the common code shared between them. Then, we will move onto the actual code examples themselves. These example will include:

- Basic queues
- Worker queues
- Publisher and subscribers
- Direct routing of queues
- Topic-based publisher and subscribers
- Remote procedure calls

RabbitMQ Client Library

To develop software against RabbitMQ, you will need to install the RabbitMQ client library for .NET. Before we look at how to install the client library, let's take a brief look at what it is. This book will not serve as an in-depth guide to the whole client library API. You can read a more in-depth document for the client library that explains the full library on the RabbitMQ website.

This section will serve as an introduction to the library and the examples in the rest of this book will help you further cement your understanding.

What is Contained in the Client Library?

The RabbitMQ .NET client is an implementation of an AMQP client library for C# and other .NET languages. The client library implements the AMQP specification 0-8 and 0-9. The API is closely modeled on the AMQP protocol specification with little additional abstraction so, if you have a good understanding of the AMQP protocol, then you will find the client library easy to follow.

The core API interfaces and classes are defined in the ***RabbitMQ.Client*** namespace. The main API interfaces and classes are:

- ***IModel***: This represents an AMQP data channel and provides most of the AMQP operations.
- ***IConnection***: This represents an AMQP connection.
- ***ConnectionFactory***: This constructs IConnection instances.

Some other useful interfaces and classes include:

- ***ConnectionParamters***: This configures a ConnectionFactory.
- ***QueueingBasicConsumer***: This receives messages delivered from the server.

Connection to a Message Broker

You can use the following code to connect to a RabbitMQ broker:

```
ConnectionFactory factory = new ConnectionFactory { HostName = "localhost", UserName
= "guest", Password = "guest" };

IConnection _connection = _factory.CreateConnection();
```

The parameters to connect to the broker are:

- ***HostName***: The RabbitMQ server host to which to connect.
- ***UserName***: The username with which to connect. This example uses the default guest account but, in reality, you should create your own account.
- ***Password***: The password with which to connect. This example uses the default guest account but, in reality, you should create your own account.

Once a connection has been made to RabbitMQ, you can then open a channel to RabbitMQ:

```
IModel _model = _connection.CreateModel();
```

The channel can now be used to send and receive messages.

Exchanges and Queues

Client applications work with exchanges and queues. These are high-level constructs in AMQP. Exchanges and queues must be declared before they can be used. Declaring either type of object ensures the one of that name exists. If it doesn't exist, it is created.

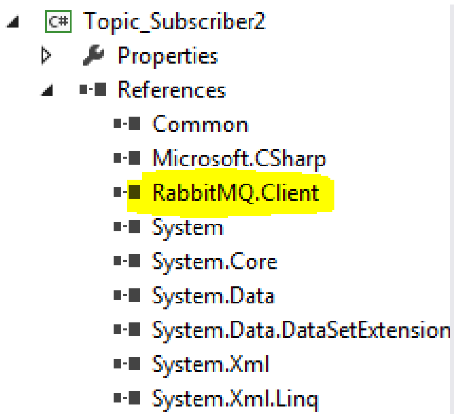
The following code snippet declares an exchange and a queue, then binds them together:

```
channel.ExchangeDeclare("MyExchange", "direct");
channel.QueueDeclare("MyQueue");

channel.QueueBind("MyQueue", ExchangeName, "");
```

Once the exchange and queues are set up and bound together, you are free to start sending and receiving messages. We will cover how to do this when we start with the first worked example.

Installing the .NET Client Library

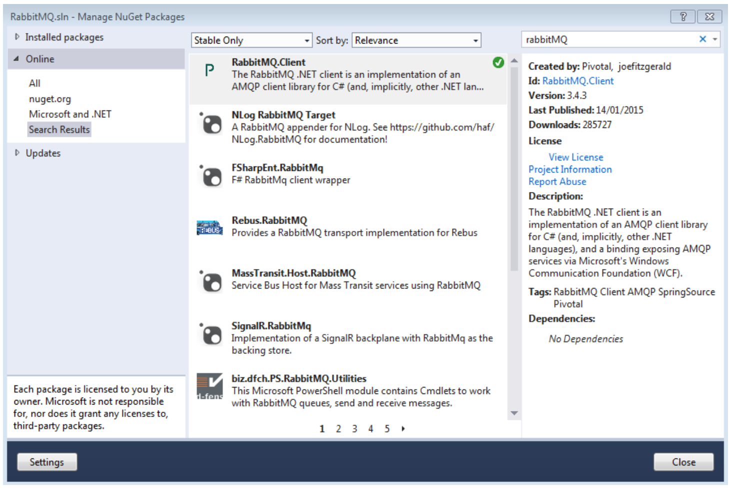
There are a few ways to get the client library set up in your application. The first way is to go to the RabbtMQ client library downloads page and directly download the library. Once you have downloaded the library, you can include a reference to “RabbitMQ.Client.dll” in your application as shown in the following screenshot:



Topic_Subscriber2
- Properties
- References
  - Common
  - Microsoft.CSharp
  - RabbitMQ.Client
  - System
  - System.Core
  - System.Data
  - System.Data.DataSetExtension
  - System.Xml
  - System.Xml.Linq

Figure 41: RabbitMQ.Client library reference

A better way would be to use the NuGet package manager to get the library. You can do this via the NuGet packages dialog box or via the Package Manager Console. To do this via the UI, open the Packages dialog box in Visual Studio from the Tools Menu -> NuGet Package Manager -> Manage NuGet Packages for Solution. This will show a window as seen in Figure 42:



RabbitMQ.Client
The RabbitMQ .NET client is an implementation of an AMQP client library for C# (and, implicitly, other .NET lan...

NLog RabbitMQ Target
A RabbitMQ appender for NLog. See https://github.com/haf/NLog.RabbitMQ for documentation!

FSharpEnt.RabbitMq
F# RabbitMq client wrapper

Rebus.RabbitMQ
Provides a RabbitMQ transport implementation for Rebus

MassTransit.Host.RabbitMQ
Service Bus Host for Mass Transit services using RabbitMQ

SignalR.RabbitMq
Implementation of a SignalR backplane with RabbitMq as the backing store.

biz.dfch.PS.RabbitMQ.Utilities
This Microsoft PowerShell module contains Cmdlets to work with RabbitMQ queues, send and receive messages.

Created by: Pivotal, joefitzgerald
Id: RabbitMQ.Client
Version: 3.4.3
Last Published: 14/01/2015
Downloads: 285727
License
View License
Project Information
Report Abuse
Description:
The RabbitMQ .NET client is an implementation of an AMQP client library for C# (and, implicitly, other .NET languages), and a binding exposing AMQP services via Microsoft's Windows Communication Foundation (WCF).
Tags: RabbitMQ Client AMQP SpringSource Pivotal
Dependencies:
No Dependencies

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

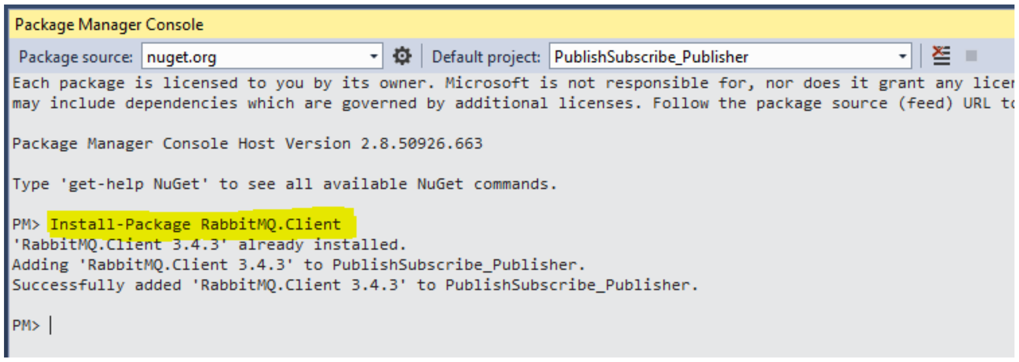
Figure 42: RabbitMQ package manager UI in Visual Studio

Select the nuget.org option in the “Online” tree to the left of the screen. Then, in the search box on the top right-hand corner of the screen, type “rabbitMQ”. This will show the RabbitMQ.Client library in the list of libraries in the middle column of the screen. Double-click this library to install it into your project.

You can also install the client library by using the package manager console. To open this, go to the Tools menu and select Tools -> NuGet Package Manager -> Package Manager Console. When the console window appears, type:

```
Install-Package RabbitMQ.Client
```

This will install the client library into your project. You should see output similar to the following screenshot:


Package Manager Console
Package source: nuget.org
Default project: PublishSubscribe_Publisher

```
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licer
may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to

Package Manager Console Host Version 2.8.50926.663

Type 'get-help NuGet' to see all available NuGet commands.

PM> Install-Package RabbitMQ.Client
'RabbitMQ.Client 3.4.3' already installed.
Adding 'RabbitMQ.Client 3.4.3' to PublishSubscribe_Publisher.
Successfully added 'RabbitMQ.Client 3.4.3' to PublishSubscribe_Publisher.

PM> |
```

Figure 43: Installing the RabbitMQ client library in the package manager console

You are now ready to start developing code against RabbitMQ.

Example Code Scenario

The code samples in the rest of this book all use a common theme of a payment provider. The theme has been kept consistent to show how the different queuing scenarios relate to each other. The samples are based around a system (e.g., the producer or publisher) making either a card payment or raising a purchase order. These card payments or purchase orders will be placed into a RabbitMQ queue setup for different scenarios.

Then there will be a consuming application or applications that will process these payments of purchase orders. Payment processors are a common use case for message queuing systems as the durable nature of the message queues means that you won't lose payment messages. It goes without saying that payment processors are not the only use case for message queues and you may have very different requirements. But payment processing made a good example for the purposes of this book.

Common Code Throughout the Examples

Each of the example projects in the rest of this book uses a common code project that contains code shared between all of the examples. Before we dive into the example project, let's take a look at the shared code.

Note: I recommend that you get the sample code for this book to refer to. The code can be downloaded from https://bitbucket.org/syncfusiontech/message-queuing-with-rabbitmq-succinctly.

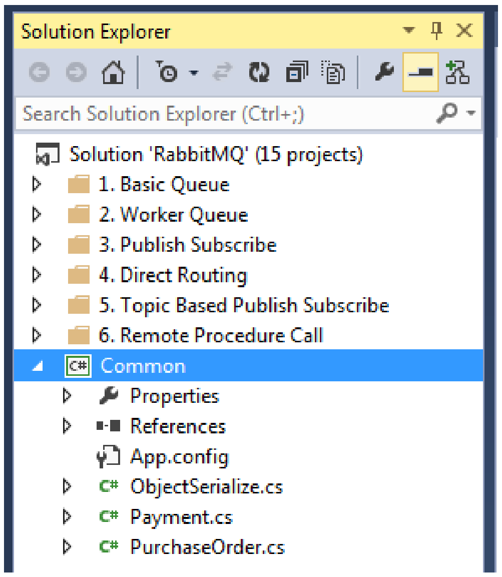


Figure 44: Common code for the example project

Payment.cs

All of the example projects have the concept of placing card payments onto a queue. The examples use the ***Payment.cs*** class:

```
using System;

namespace RabbitMQ.Examples
{
    [Serializable]
    public class Payment
    {
        public decimal AmountToPay;
        public string CardNumber;
        public string Name;
    }
}
```

The class is marked serializable as it will need to be serialized into a byte array before it is placed onto a queue. The class contains three public properties. The first is ***AmountToPay*** which is a decimal that represents a payment amount. The second property, ***CardNumber***, is a string that represents a debit/credit card number. For the examples in this book, it is just treated as an arbitrary string. The final property, ***Name***, is the name of the person making the payment.

PurchaseOrder.cs

Some of the example projects post different types of payments onto a queue. These are card payments (as shown previously) and purchase orders. Again, this class is marked as serializable as it will be turned into a byte array before posting onto the queue:

```
using System;

namespace RabbitMQ.Examples
{
    [Serializable]
    public class PurchaseOrder
    {
        public decimal AmountToPay;
        public string PoNumber;
        public string CompanyName;
        public int PaymentDayTerms;
    }
}
```

The first property is a decimal representing an **AmountToPay**. The second property, ***PoNumber***, represents a purchase order number. This is typically used when companies raise purchase orders instead of paying with cards. A prearranged purchase order would be set up with a vendor and the purchase order number would be logged along with the payment. The difference between this and a card payment is that the money may not show up for a number of months after the vendor has raised an invoice.

The third property is the ***CompanyName*** and this is the name of the company against which the purchase order has been raised. The fourth and final property, ***PaymentDayTerms***, is an integer representing the number of days after an invoice is raised that the purchase order will be paid. This may typically be 30, 45, or 75 days.

ObjectSerialize

The final class in the common code project is an object serializer. This static class exposes two extension methods. One of the extension methods is on the ***Object*** base class to serialize that object into a byte array. We do this with a binary formatter for the purposes of the examples in this book. A binary formatter is a good serialization method to use if the systems you are integrating with are also implemented in .NET. If you are integrating with a Java system, for example, you may want to use an XmlFormatter but that is out the scope of this book. The key principle here is, you need to turn an object into a byte array to place it onto a message queue.

The ***DeSerialize*** method is an extension method for a byte array which lets you turn a serialized byte array back into its original form (i.e., a ***Payment*** message):

```
using System;
using System.IO;
using System.IO.Compression;
using System.Runtime.Serialization.Formatters.Binary;

namespace RabbitMQ.Examples
{
    public static class ObjectSerialize
    {
        public static byte[] Serialize(this Object obj)
        {
            if (obj == null)
            {
                return null;
            }

            using (var memoryStream = new MemoryStream())
            {
                var binaryFormatter = new BinaryFormatter();

                binaryFormatter.Serialize(memoryStream, obj);

                var compressed = Compress(memoryStream.ToArray());
                return compressed;
            }
        }

        public static Object DeSerialize(this byte[] arrBytes)
        {
            using (var memoryStream = new MemoryStream())
            {
                var binaryFormatter = new BinaryFormatter();
                var decompressed = Decompress(arrBytes);

                memoryStream.Write(decompressed, 0, decompressed.Length);
                memoryStream.Seek(0, SeekOrigin.Begin);

                return binaryFormatter.Deserialize(memoryStream);
            }
        }

        private static byte[] Compress(byte[] input)
        {
            byte[] compressesData;

            using (var outputStream = new MemoryStream())
            {
                using (var zip = new GZipStream(outputStream,
CompressionMode.Compress))
                {
                    zip.Write(input, 0, input.Length);
                }

                compressesData = outputStream.ToArray();
            }

            return compressesData;
        }

        private static byte[] Decompress(byte[] input)
        {
            byte[] decompressedData;

            using (var outputStream = new MemoryStream())
            {
                using (var inputStream = new MemoryStream(input))
                {
                    using (var zip = new GZipStream(inputStream,
CompressionMode.Decompress))
                    {
                        zip.CopyTo(outputStream);
                    }
                }

                decompressedData = outputStream.ToArray();
            }

            return decompressedData;
        }
    }
}
```

The ObjectSerialize class also contains two private static methods that compress and decompress the byte array by using the ***GZipStream*** class in .NET. This helps ensure the serialized message is as small as it can be:

```
Payment payment1 = new Payment { AmountToPay = 25.0m, CardNumber = "1234123412341234"
};

byte [] serialized = payment1.Serialize();

Payment payment_deserialized =  serialized.DeSerialize();
```

The preceding code example shows a payment message that has been constructed and is then serialized into a byte array. The serialized byte array is then immediately deserialized back into a payment message.

Example 1: Basic Queue

In this first example, we will show a basic queue scenario that is probably familiar to most enterprise software developers—especially if you are familiar with the Microsoft MSMQ platform. In this example, we will have one producer posting a payment message onto the queue and one consumer reading that message from the queue. In this example, the producer and consumer application are the same.

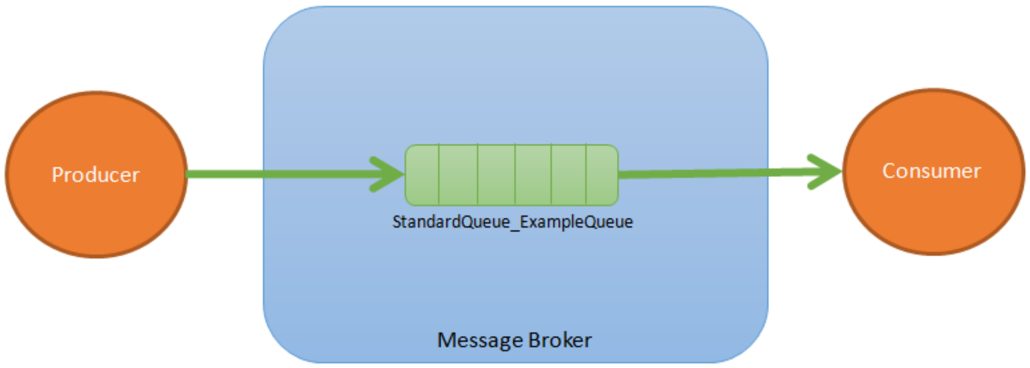


Figure 45: Basic queue example

In the ***Main*** method, we start off by creating 10 payment messages. Once they have been created, the queue is created, the messages are sent to the queue, and then the messages are read directly off the queue:

```
public static void Main()
{
    var payment1 = new Payment { AmountToPay = 25.0m, CardNumber = "1234123412341234"
};
    var payment2 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234"
};
    var payment3 = new Payment { AmountToPay = 2.0m, CardNumber = "1234123412341234"
};
    var payment4 = new Payment { AmountToPay = 17.0m, CardNumber = "1234123412341234"
};
    var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
    var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
    var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
    var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
    var payment9 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234"
};
    var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

    CreateQueue();
    SendMessage(payment1);
    SendMessage(payment2);
    SendMessage(payment3);
    SendMessage(payment4);
    SendMessage(payment5);
    SendMessage(payment6);
    SendMessage(payment7);
    SendMessage(payment8);
    SendMessage(payment9);
    SendMessage(payment10);

    Recieve();

    Console.ReadLine();
}
```

You may have already noticed from the preceding diagram that the producer directly posts onto the queue instead of to an exchange. You can post directly to a queue from the client API but, under the covers, you are posting to a default exchange.

For example, when you declare a queue with the name of "StandardQueue_ExampleQueue," the RabbitMQ broker will bind it to the default exchange by using "StandardQueue_ExampleQueue" as the routing key. Therefore, a message published to the default exchange with the routing key "StandardQueue_ExampleQueue" will be routed to the queue "StandardQueue_ExampleQueue." This means the default exchange makes it seem like it is possible to deliver messages directly to queues even though that is not what is actually happening behind the scenes.

Continuing on with this example, next we have the ***CreateQueue*** method. Notice that the queue name is placed in a constant called ***QueueName***:

```
private static ConnectionFactory _factory;
private static IConnection _connection;
private static IModel _model;

private const string QueueName = "StandardQueue_ExampleQueue";

private static void CreateQueue()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest"};
    _connection = _factory.CreateConnection();
    _model = _connection.CreateModel();
    _model.QueueDeclare(QueueName, true, false, false, null);
    _connection = _factory.CreateConnection();
}
```

The ***CreateQueue*** method is where we create a connection with the ***ConnectionFactory*** and then create the model by using ***CreateModel***. Once the model has been opened, the queue called "StandardQueue_ExampleQueue" is declared by passing in the queue name. The second "true" parameter tells the broker that we want the queue to be durable (i.e. the messages to be persisted to disk).

Next, we have a static method called ***SendMessage*** to post a payment message to the queue:

```
private static void SendMessage(Payment message)
{
    _model.BasicPublish("", QueueName, null, message.Serialize());
    Console.WriteLine(" [x] Payment Message Sent : {0} : {1}", message.CardNumber,
message.AmountToPay);
}
```

Here we have a call to ***BasicPublish*** on the channel which is the "_model" global property in our program. The first empty parameter is an exchange name but, because we are not defining our own customer exchange, we leave it blank to use the default exchange. The next parameter is the name of the queue we want to publish to; in this case it is "StandardQueue_ExampleQueue."

The next null parameter is for a set of Basic Parameters that can be passed into the queue like correlationIDs, ReplyTo addresses, etc., but we don't need them for this example. Then, finally, we call ***Serialize*** on our payment message. This calls the extension method that we defined earlier which converts our payment message instance into a compressed byte array:

```
public static void Recieve()
{
    var consumer = new QueueingBasicConsumer(_model);

    var msgCount = GetMessageCount(_model, QueueName);
    _model.BasicConsume(QueueName, true, consumer);

    var count = 0;

    while (count < msgCount)
    {
        var message = (Payment)consumer.Queue.Dequeue().Body.DeSerialize();

        Console.WriteLine("----- Received {0} : {1}", message.CardNumber,
message.AmountToPay);
        count++;
    }
}

private static uint GetMessageCount(IModel channel, string queueName)
{
    var results = channel.QueueDeclare(queueName, true, false, false, null);
    return results.MessageCount;
}
```

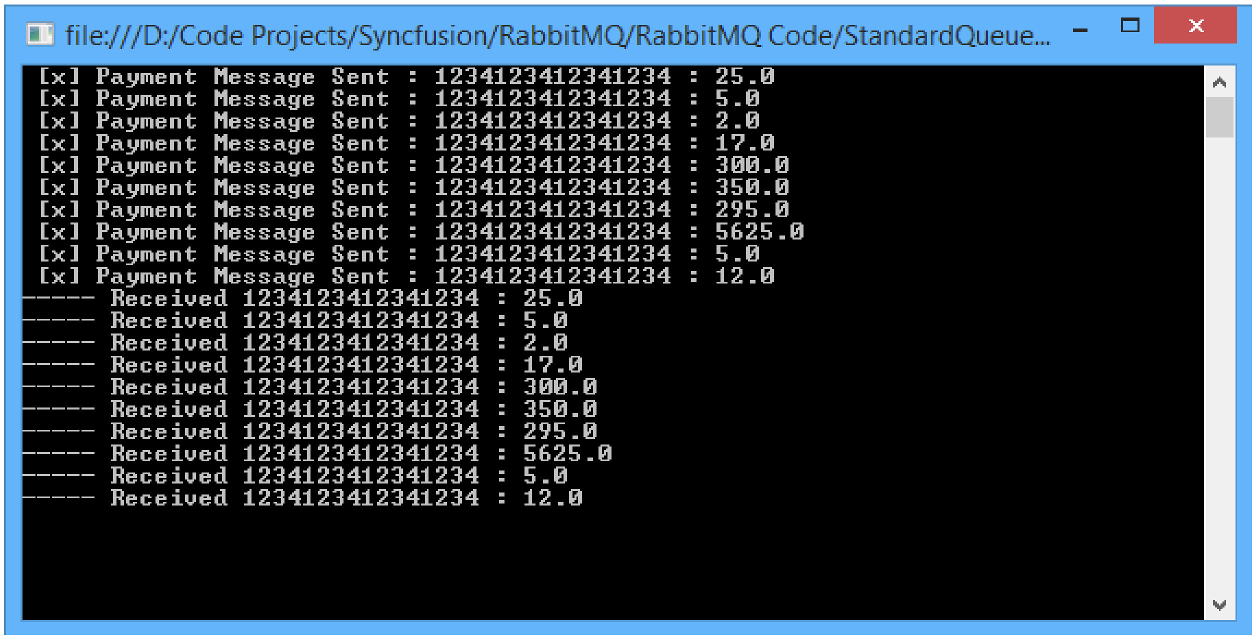
Next, we have a method to receive messages that have been posted onto the queue. First, a ***QueueingBasicConsumer*** is created which takes an instance of the model as a parameter. Then we get a count of the messages on the queue. To do this, you need to call ***QueueDeclare*** again to redeclare the queue. From doing this, you will get a ***QueueDeclareOK*** object back. In this object there will be a message count.

Note: Declaring a queue in RabbitMQ is an idempotent operation. This means it will only be created if it doesn't already exist. By redeclaring the queue again, we get details returned about the queue that already exists. An idempotent operation is one that has no additional effect if it is called more than once with the same input parameters. In our example, if you call QueueDeclare twice with the same queue name, you wouldn't get two queues created but, rather, just the one queue as the second call would have no effect.

Next, a call is made to BasicConsume on the model. This method asks the server to start a "consumer" which is a transient request for messages from a specific queue. Consumers last as long as the channel on which they were declared or until the client cancels them.

Next, we iterate through the messages by calling ***Dequeue*** on the queue and then calling ***DeSerialize*** on the results of the DeQueued message. This will give us back our original queues payment message.

The following screenshot of this program running shows how the original 10 payment messages were sent to the queue and then consumed straight away from the queue in the same order:



file:///D:/Code Projects/Syncfusion/RabbitMQ/RabbitMQ Code/StandardQueue...

```
 [x] Payment Message Sent : 1234123412341234 : 25.0
 [x] Payment Message Sent : 1234123412341234 : 5.0
 [x] Payment Message Sent : 1234123412341234 : 2.0
 [x] Payment Message Sent : 1234123412341234 : 17.0
 [x] Payment Message Sent : 1234123412341234 : 300.0
 [x] Payment Message Sent : 1234123412341234 : 350.0
 [x] Payment Message Sent : 1234123412341234 : 295.0
 [x] Payment Message Sent : 1234123412341234 : 5625.0
 [x] Payment Message Sent : 1234123412341234 : 5.0
 [x] Payment Message Sent : 1234123412341234 : 12.0
----- Received 1234123412341234 : 25.0
----- Received 1234123412341234 : 5.0
----- Received 1234123412341234 : 2.0
----- Received 1234123412341234 : 17.0
----- Received 1234123412341234 : 300.0
----- Received 1234123412341234 : 350.0
----- Received 1234123412341234 : 295.0
----- Received 1234123412341234 : 5625.0
----- Received 1234123412341234 : 5.0
----- Received 1234123412341234 : 12.0
```

Figure 46: Messages played onto the queue and then directly received

That is our first example completed. Next, we will expand on this sample to create a worker queue. Here is the complete code for this example if you didn't type in any of the previous code:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static IModel _model;

        private const string QueueName = "StandardQueue_ExampleQueue";

        public static void Main()
        {
            var payment1 = new Payment { AmountToPay = 25.0m, CardNumber =
"1234123412341234" };
            var payment2 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment3 = new Payment { AmountToPay = 2.0m, CardNumber =
"1234123412341234" };
            var payment4 = new Payment { AmountToPay = 17.0m, CardNumber =
"1234123412341234" };
            var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
            var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
            var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
            var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
            var payment9 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

            CreateQueue();

            SendMessage(payment1);
            SendMessage(payment2);
            SendMessage(payment3);
            SendMessage(payment4);
            SendMessage(payment5);
            SendMessage(payment6);
            SendMessage(payment7);
            SendMessage(payment8);
            SendMessage(payment9);
            SendMessage(payment10);

            Recieve();

            Console.ReadLine();
        }

        private static void CreateQueue()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest"};
            _connection = _factory.CreateConnection();
            _model = _connection.CreateModel();
            _model.QueueDeclare(QueueName, true, false, false, null);

        }

        private static void SendMessage(Payment message)
        {
            _model.BasicPublish("", QueueName, null, message.Serialize());
            Console.WriteLine(" [x] Payment Message Sent : {0} : {1}",
message.CardNumber, message.AmountToPay);
        }


        public static void Recieve()
        {
            var consumer = new QueueingBasicConsumer(_model);

            var msgCount = GetMessageCount(_model, QueueName);
            _model.BasicConsume(QueueName, true, consumer);

            var count = 0;

            while (count < msgCount)
            {
                var message = (Payment)consumer.Queue.Dequeue().Body.DeSerialize();

                Console.WriteLine("----- Received {0} : {1}", message.CardNumber,
message.AmountToPay);
                count++;
            }
        }

        private static uint GetMessageCount(IModel channel, string queueName)
        {
            var results = channel.QueueDeclare(queueName, true, false, false, null);
            return results.MessageCount;
        }
    }
}
```

Example 2: Worker Queue

In this next example, we will build on the basic queue by introducing multiple consumers. This creates what is called a worker queue in which the messages from the queue are shared between two consumers. This is commonly used where you need to share the load between consumers when processing a high volume of messages:

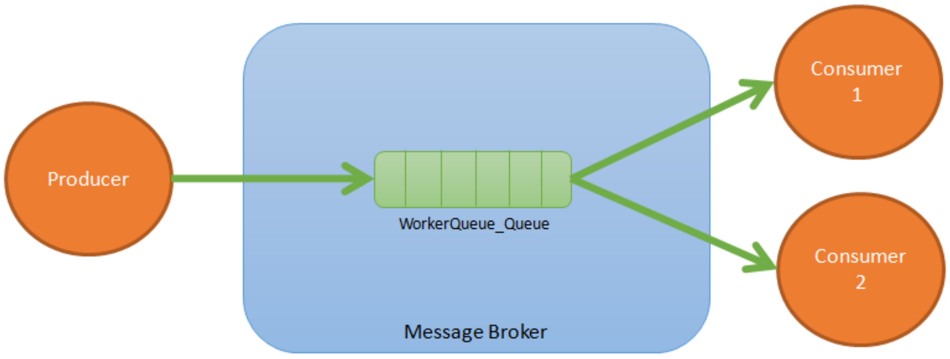


Figure 47: Worker queue example

As with the previous example, we start off in the ***Main*** method by create 10 payment messages:

```
static void Main()
{
    var payment1 = new Payment { AmountToPay = 25.0m, CardNumber = "1234123412341234" 
};
    var payment2 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234" 
};
    var payment3 = new Payment { AmountToPay = 2.0m, CardNumber = "1234123412341234" 
};
    var payment4 = new Payment { AmountToPay = 17.0m, CardNumber = "1234123412341234" 
};
    var payment5 = new Payment { AmountToPay = 300.0m, CardNumber = 
"1234123412341234" };
    var payment6 = new Payment { AmountToPay = 350.0m, CardNumber = 
"1234123412341234" };
    var payment7 = new Payment { AmountToPay = 295.0m, CardNumber = 
"1234123412341234" };
    var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber = 
"1234123412341234" };
    var payment9 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234" 
};
    var payment10 = new Payment { AmountToPay = 12.0m, CardNumber = 
"1234123412341234" };

    CreateConnection();

    SendMessage(payment1);
    SendMessage(payment2);
    SendMessage(payment3);
    SendMessage(payment4);
    SendMessage(payment5);
    SendMessage(payment6);
    SendMessage(payment7);
    SendMessage(payment8);
    SendMessage(payment9);
    SendMessage(payment10);
    Console.ReadLine();
}
```

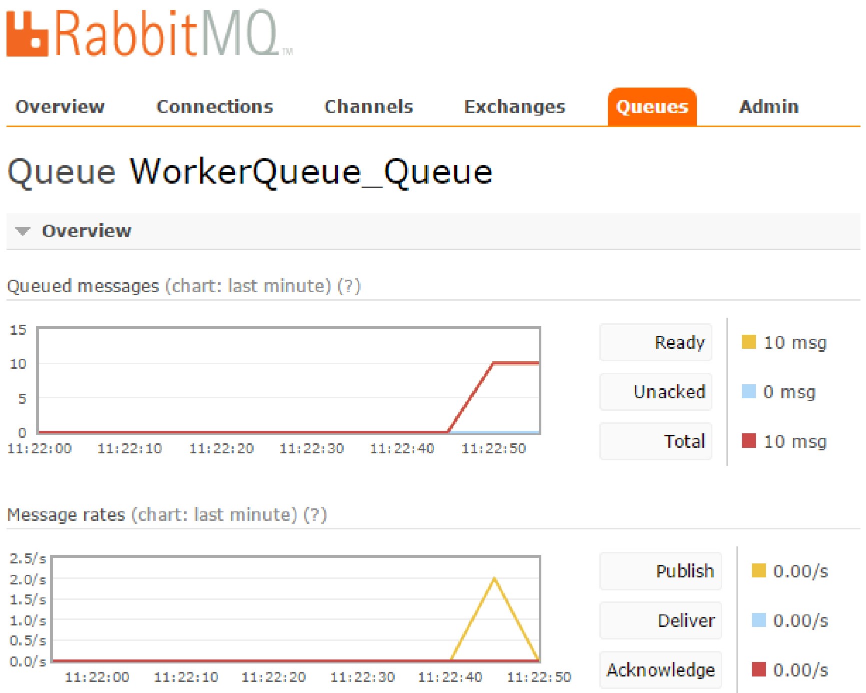
Then, we create the connection RabbitMQ along with the mode and declare a queue called “WorkerQueue_Queue.” As with the Basic Queue example, we are not directly creating an exchange but the default exchange will be used with a routing key that equates to the queue name “WorkerQueue_Queue.”

```
private static ConnectionFactory _factory;
private static IConnection _connection;
private static IModel _model;

private const string QueueName = "WorkerQueue_Queue";

private static void CreateConnection()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    _connection = _factory.CreateConnection();
    _model = _connection.CreateModel();
    _model.QueueDeclare(QueueName, true, false, false, null);
}
```

This queue has also been set up as a durable queue, meaning that the messages will be stored to the local disk on the RabbitMQ server. Once this has been done, the 10 payment messages are sent to the queue by serializing the payment to a compressed byte array:

```
private static void SendMessage(Payment message)
{
    _model.BasicPublish("", QueueName, null, message.Serialize());
    Console.WriteLine(" Payment Sent {0}, £{1}", message.CardNumber,
message.AmountToPay);
}
```

If you run the producer application first and not the consumer, and then go to the RabbitMQ management portal and browse the “WorkerQueue_Queue,” you will see the 10 messages waiting on the queue (as shown in the following screenshot):



Queue WorkerQueue_Queue

Overview

Queued messages (chart: last minute) (?)

Ready	🟨 10 msg
Unacked	🟦 0 msg
Total	🟥 10 msg

Message rates (chart: last minute) (?)

Publish	🟨 0.00/s
Deliver	🟦 0.00/s
Acknowledge	🟥 0.00/s

Figure 48: Messages are played onto the queue from the producer application

Now that the producer application is running, let's take a look at the consumer. For this example, we will have one consumer application but we will be able to run as many instances as we want to share the load of the messages placed onto the queue. In the WorkerQueue_Consumer project in this book's sample code, you will see a method called from ***Main*** called ***Receive***. There is a few things going on here so let's work down the method from top to bottom.

```
public static void Receive()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    using (_connection = _factory.CreateConnection())
    {
        using (var channel = _connection.CreateModel())
        {
            channel.QueueDeclare(QueueName, true, false, false, null);
            channel.BasicQos(0, 1, false);

            var consumer = new QueueingBasicConsumer(channel);
            channel.BasicConsume(QueueName, false, consumer);

            while (true)
            {
                var ea = consumer.Queue.Dequeue();
                var message = (Payment)ea.Body.DeSerialize();
                channel.BasicAck(ea.DeliveryTag, false);
                Console.WriteLine("----- Payment Processed {0} : {1}", 
message.CardNumber, message.AmountToPay);
            }
        }
    }
}
```

First of all, a new ***ConnectionFactory*** and connection is created to connect us to RabbitMQ. Then, a channel is created by calling ***CreateModel***. Next, the queue is redeclared. As we discussed in the previous example, the call to QueueDeclare is idempotent which means that, if you call ***QueueDeclare*** for a queue that already exists, then you will get the original queue's details returned instead of the queue actually being redeclared.

Next, we have a call to ***BasicQos***. The second parameter, which is set to 1, is defining a prefetch count. What this means is, RabbitMQ won't dispatch a new message to a consumer until that consumer has finished processing and acknowledging the message (we will look at message acknowledgement in a moment). RabbitMQ will instead dispatch the message to the next worker that is not busy:

```
channel.BasicQos(0, 1, false);
```

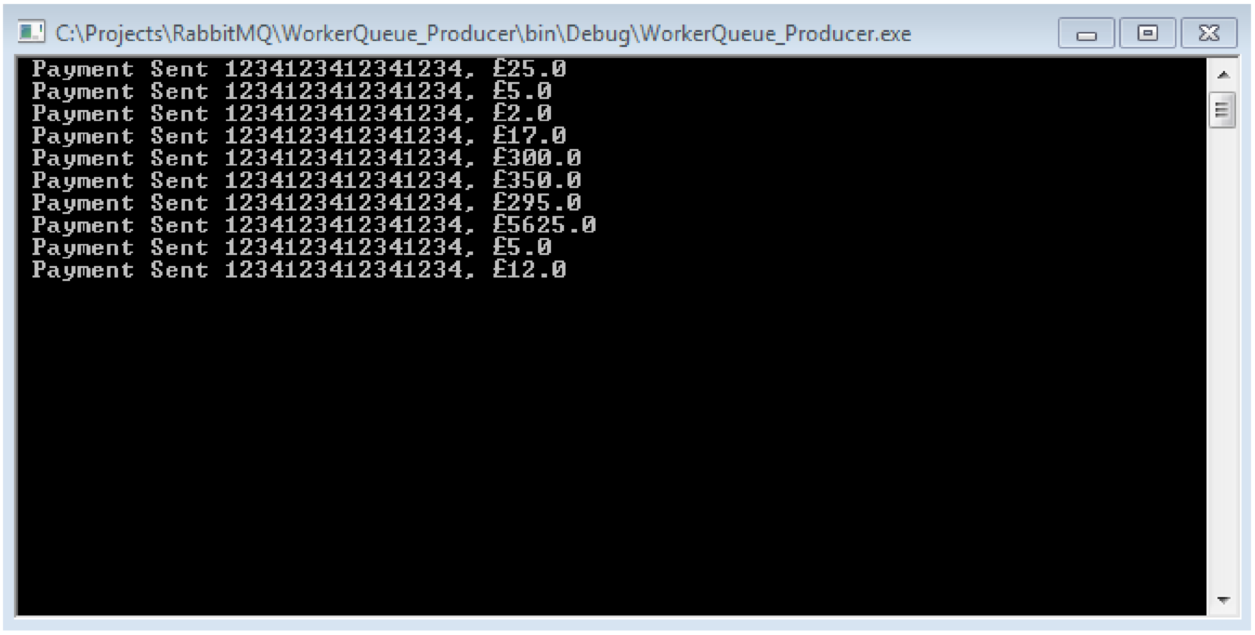
This helps to prevent situations in which you might have one consumer that is constantly busy and another that is lighter on work. By setting this prefetch count, you are ensuring a more equal and fair dispatch of messages.

```
var consumer = new QueueingBasicConsumer(channel);
channel.BasicConsume(QueueName, false, consumer);

while (true)
{
    var ea = consumer.Queue.Dequeue();
    var message = (Payment)ea.Body.DeSerialize();
    channel.BasicAck(ea.DeliveryTag, false);

    Console.WriteLine("----- Payment Processed {0} : {1}", message.CardNumber, 
message.AmountToPay);
}
```

Next, the ***BasicQueueingConsumer*** is created and ***BasicConsume*** called for the queue "WorkerQueue_Consumer." The second parameter to ***BasicConsume*** is a Boolean which represents whether or not we want to expect an acknowledgement message. In this example, we want to send message acknowledgements so this parameter is set to false.

We then have a while loop that is started where the message is dequeued and deserialized back into a payment message. Once we have the message and have acted upon it, we send a delivery acknowledgement with the ***BasicAck*** call. This tells the message broker that we have finished processing the message and are ready for the next message when it is ready.

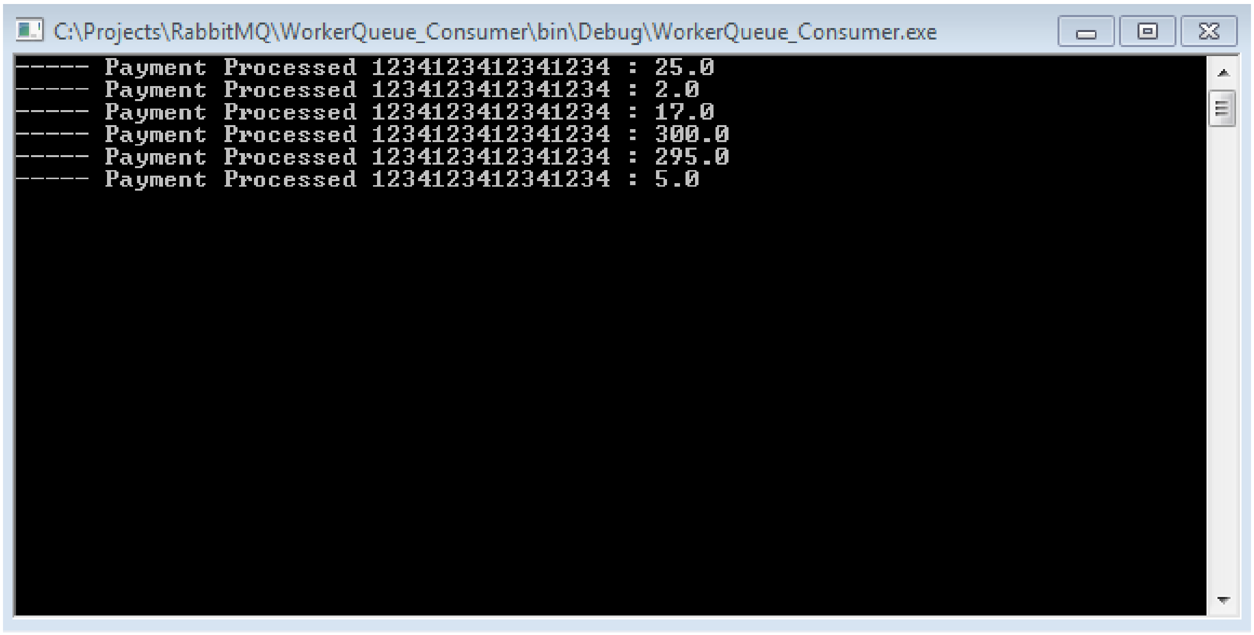
To demonstrate this application, run two copies of the consumer application and then run the producer application. The producer will show that 10 messages have been sent as in the following screenshot:


C:\Projects\RabbitMQ\WorkerQueue_Producer\bin\Debug\WorkerQueue_Producer.exe

```
Payment Sent 1234123412341234, £25.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £2.0
Payment Sent 1234123412341234, £17.0
Payment Sent 1234123412341234, £300.0
Payment Sent 1234123412341234, £350.0
Payment Sent 1234123412341234, £295.0
Payment Sent 1234123412341234, £5625.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £12.0
```

Figure 49: Worker queue producer has put 10 payment messages onto the queue

You will then see the two consumer applications processing the messages as shown in the following screenshot for one of the consumers:



C:\Projects\RabbitMQ\WorkerQueue_Consumer\bin\Debug\WorkerQueue_Consumer.exe

```
----- Payment Processed 1234123412341234 : 25.0
----- Payment Processed 1234123412341234 : 2.0
----- Payment Processed 1234123412341234 : 17.0
----- Payment Processed 1234123412341234 : 300.0
----- Payment Processed 1234123412341234 : 295.0
----- Payment Processed 1234123412341234 : 5.0
```

Figure 50: Messages are distributed to the different consumer queues

As each consumer processes a message, the next message will not be received by that consumer until it sends the delivery acknowledgement. You can easily demonstrate this by commenting out the following line from the ***Receive*** method:

```
channel.BasicAck(ea.DeliveryTag, false);
```

If you then re-run both of the consumers and then execute the producer application, the consumers will only process one message each and not receive any more messages (as they have not sent an acknowledgement back to the RabbitMQ server).

In the code following code, you will find the complete implementation for the producer application:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    public class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static IModel _model;

        private const string QueueName = "WorkerQueue_Queue";

        static void Main()
        {
            var payment1 = new Payment { AmountToPay = 25.0m, CardNumber =
"1234123412341234" };
            var payment2 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment3 = new Payment { AmountToPay = 2.0m, CardNumber =
"1234123412341234" };
            var payment4 = new Payment { AmountToPay = 17.0m, CardNumber =
"1234123412341234" };
            var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
            var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
            var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
            var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
            var payment9 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

            CreateConnection();

            SendMessage(payment1);
            SendMessage(payment2);
            SendMessage(payment3);
            SendMessage(payment4);
            SendMessage(payment5);
            SendMessage(payment6);
            SendMessage(payment7);
            SendMessage(payment8);
            SendMessage(payment9);
            SendMessage(payment10);

            Console.ReadLine();
        }

        private static void CreateConnection()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            _connection = _factory.CreateConnection();
            _model = _connection.CreateModel();
            _model.QueueDeclare(QueueName, true, false, false, null);
        }

        private static void SendMessage(Payment message)
        {
            _model.BasicPublish("", QueueName, null, message.Serialize());
            Console.WriteLine(" Payment Sent {0}, £{1}", message.CardNumber,
message.AmountToPay);
        }
    }
}
```

In the code below, you will find the complete code for the consumer application. Please note that you need the RabbitMQ Client installed on any machine where the consumer and producer applications are executed. This example also uses the shared code from the code solution provided with this book:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    public class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private const string QueueName = "WorkerQueue_Queue";

        static void Main()
        {
            Receive();

            Console.ReadLine();
        }

        public static void Receive()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    channel.QueueDeclare(QueueName, true, false, false, null);
                    channel.BasicQos(0, 1, false);

                    var consumer = new QueueingBasicConsumer(channel);
                    channel.BasicConsume(QueueName, false, consumer);

                    while (true)
                    {
                        var ea = consumer.Queue.Dequeue();
                        var message = (Payment)ea.Body.DeSerialize();
                        channel.BasicAck(ea.DeliveryTag, false);

                        Console.WriteLine("----- Payment Processed {0} : {1}",
message.CardNumber, message.AmountToPay);
                    }
                }
            }
        }
    }
}
```

Example 3: Publisher and Subscriber

In this example, we are going to expand on the previous worker queue concept. This example is called a publisher and subscriber. In the worker queue example, the consumers could share the load of messages from the queue. This means that messages are distributed between multiple consumers:

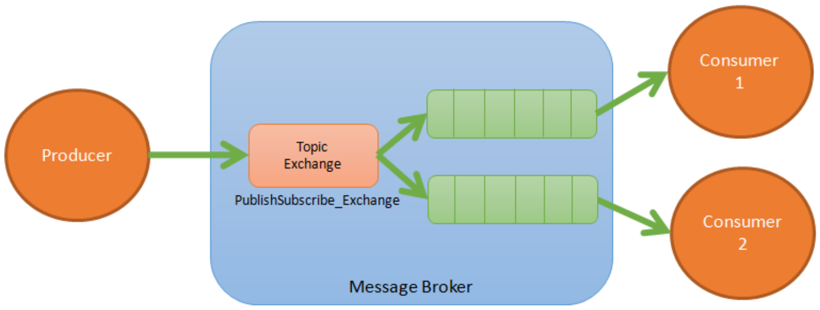


Figure 51: Publisher and subscriber example

With the publisher and subscriber example, the messages are sent from the exchange to all of the consumers that are bound to the exchange. This means the messages are not picked up by multiple consumers to distribute load but, instead, all consumers are interested in receiving the messages.

Unlike the previous example, we are going to set up an explicit exchange. In the worker queue example we directly defined a queue and this used a default direct exchange behind the scenes. This time, we will set up an exchange and set its type to "fanout." A fanout exchange routes messages to all of the queues that are bound to it and the routing key is ignored. If 10 queues are bound to a fanout exchange, when a new message is published to that exchange, a copy of the message is delivered to all 10 queues.

Let's take a look at the code for the publisher application. First, as in the previous examples, we will set up some payment messages:

```
static void Main()
{
    var payment1 = new Payment { AmountToPay = 25.0m, CardNumber = "1234123412341234"
};
    var payment2 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234"
};
    var payment3 = new Payment { AmountToPay = 2.0m, CardNumber = "1234123412341234"
};
    var payment4 = new Payment { AmountToPay = 17.0m, CardNumber = "1234123412341234"
};
    var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
```

```
    var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
    var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
    var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
    var payment9 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234"
};
    var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

    CreateConnection();

    SendMessage(payment1);
    SendMessage(payment2);
    SendMessage(payment3);
    SendMessage(payment4);
    SendMessage(payment5);
    SendMessage(payment6);
    SendMessage(payment7);
    SendMessage(payment8);
    SendMessage(payment9);
    SendMessage(payment10);
}
```

When we create the connection, we instead declare an exchange instead of a queue. Notice when the exchange is declared, we specify the “fanout” exchange type:

```
private static ConnectionFactory _factory;
private static IConnection _connection;
private static IModel _model;

private const string ExchangeName = "PublishSubscribe_Exchange";

private static void CreateConnection()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    _connection = _factory.CreateConnection();
    _model = _connection.CreateModel();
    _model.ExchangeDeclare(ExchangeName, "fanout");
}
```

Once the connection and queue is declared, we then send the payment messages to the queue in a similar fashion as before but, instead of providing a queue name in the second parameter as the routing key, we instead provide the exchange name—which in this case is “PublishSubscribe_Exchange.”

```
private static void SendMessage(Payment message)
```

```
{
    _model.BasicPublish(ExchangeName, "", null, message.Serialize());
    Console.WriteLine(" Payment Sent {0}, £{1}", message.CardNumber,
message.AmountToPay);
}
```

Now that the messages are in the queue, we can now look at the subscriber application that will receive the messages. Remember, that in this example, if we have five subscribers all connected to the exchange, each subscriber will receive all of the messages that are sent:

```
private static ConnectionFactory _factory;
private static IConnection _connection;
private static QueueingBasicConsumer _consumer;

private const string ExchangeName = "PublishSubscribe_Exchange";

static void Main()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    using (_connection = _factory.CreateConnection())
    {
        using (var channel = _connection.CreateModel())
        {
            var queueName = DeclareAndBindQueueToExchange(channel);
            channel.BasicConsume(queueName, true, _consumer);

            while (true)
            {
                var ea = _consumer.Queue.Dequeue();
                var message = (Payment)ea.Body.DeSerialize();

                Console.WriteLine("----- Payment Processed {0} : {1}",
message.CardNumber, message.AmountToPay);
            }
        }
    }
}
```

As with previous examples, we create a connection to RabbitMQ and then a channel. Next, we need to set up the exchange and queues from which we want to read. This is done in the following code:

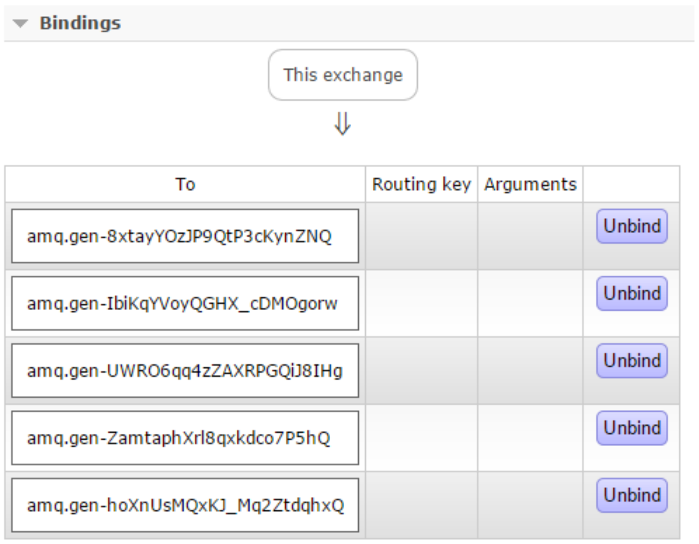
```
private static string DeclareAndBindQueueToExchange(IModel channel)
{
    channel.ExchangeDeclare(ExchangeName, "fanout");
    var queueName = channel.QueueDeclare().QueueName;
    channel.QueueBind(queueName, ExchangeName, "");
    _consumer = new QueueingBasicConsumer(channel);
    return queueName;
}
```

First, we declare the exchange by name: “PublishSubscribe_Exchange.” Similar to the last example, the ***ExchangeDeclare*** call is idempotent which means, that if you redeclare an exchange that is already there, then the already-defined exchange will be used.

Next, there is a call to declare queue. This creates a queue that is specific to this subscriber. Unlike in previous examples in which we have declared a queue by name, this time we use a system-generated queue name. This will have a name that looks something like:

amq.gen-TsdoX9qziswm9QCbdkp9Zw

Once the queue has been declared, it is bound to the exchange and a ***QueueingBasicConsumer*** is created. If we run five instances of the subscriber application, load up the RabbitMQ management portal website, and browse to our exchange “PublishSubscribe_Exchange,” you will see that there are five queues that are bound to the exchange. You can see this in the following screenshot:



To	Routing key	Arguments	
amq.gen-8xtayYOzJP9QtP3cKynZNQ			Unbind
amq.gen-IbiKqYVoyQGHX_cDMOgorw			Unbind
amq.gen-UWRO6qq4zZAXRPGQiJ8IHg			Unbind
amq.gen-ZamtaphXrl8qxkdco7P5hQ			Unbind
amq.gen-hoXnUsMQxKJ_Mq2ZtdqhxQ			Unbind

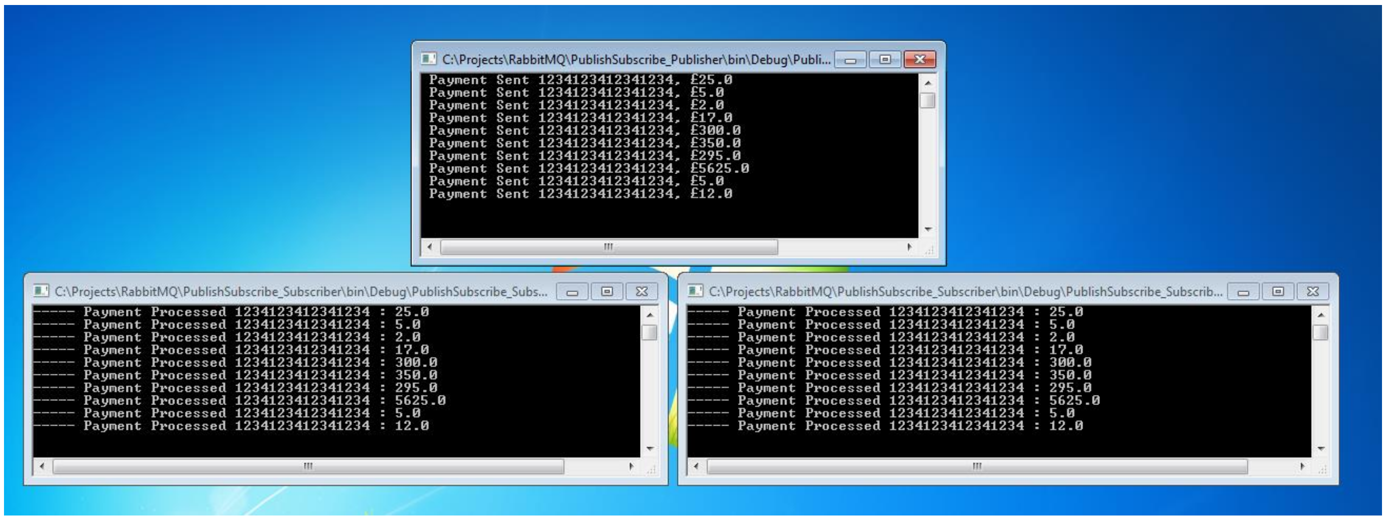
Figure 52: Subscriber queues are bound to the PublishSubscriber_Exchange

Now that the exchange and queues are declared and bound together, ***BasicConsume*** is called on the channel. The queue name has to be provided here as we are using a specific queue for this instance of the subscriber application. The second parameter is set to true. This indicates that we will not be waiting for a message acknowledgement before receiving the next message. We don’t need to in this case as our subscriber application is reading from its own queue so it will take messages at a rate at which it can deal.

Now we are ready to pull the messages from the queue. This is done within the while loop in the subscriber application as follows:

```
while (true)
{
    var ea = _consumer.Queue.Dequeue();
    var message = (Payment)ea.Body.DeSerialize();

    Console.WriteLine("----- Payment Processed {0} : {1}", message.CardNumber,
message.AmountToPay);
}
```

As with the previous examples, this example dequeues the message and deserializes it so it can be acted upon. To demonstrate this example working, open a couple of instances of the subscriber application, then open an instance of the publisher application. You will see that the publisher pushes 10 messages to the exchange, and each of the subscribers receives all of the messages. This is demonstrated in the following screenshot:


C:\Projects\RabbitMQ\PublishSubscribe_Publisher\bin\Debug\Publi...
C:\Projects\RabbitMQ\PublishSubscribe_Subscriber\bin\Debug\PublishSubscribe_Subs...
C:\Projects\RabbitMQ\PublishSubscribe_Subscriber\bin\Debug\PublishSubscribe_Subscrib...

```
Payment Sent 1234123412341234, £25.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £2.0
Payment Sent 1234123412341234, £17.0
Payment Sent 1234123412341234, £300.0
Payment Sent 1234123412341234, £350.0
Payment Sent 1234123412341234, £295.0
Payment Sent 1234123412341234, £5625.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £12.0
```

```
----- Payment Processed 1234123412341234 : 25.0
----- Payment Processed 1234123412341234 : 5.0
----- Payment Processed 1234123412341234 : 2.0
----- Payment Processed 1234123412341234 : 17.0
----- Payment Processed 1234123412341234 : 300.0
----- Payment Processed 1234123412341234 : 350.0
----- Payment Processed 1234123412341234 : 295.0
----- Payment Processed 1234123412341234 : 5625.0
----- Payment Processed 1234123412341234 : 5.0
----- Payment Processed 1234123412341234 : 12.0
```

```
----- Payment Processed 1234123412341234 : 25.0
----- Payment Processed 1234123412341234 : 5.0
----- Payment Processed 1234123412341234 : 2.0
----- Payment Processed 1234123412341234 : 17.0
----- Payment Processed 1234123412341234 : 300.0
----- Payment Processed 1234123412341234 : 350.0
----- Payment Processed 1234123412341234 : 295.0
----- Payment Processed 1234123412341234 : 5625.0
----- Payment Processed 1234123412341234 : 5.0
----- Payment Processed 1234123412341234 : 12.0
```

Figure 53: One publisher and two subscribers

Here is the complete code for the publisher application:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static IModel _model;

        private const string ExchangeName = "PublishSubscribe_Exchange";
        static void Main()
        {
            var payment1 = new Payment { AmountToPay = 25.0m, CardNumber =
"1234123412341234" };
            var payment2 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment3 = new Payment { AmountToPay = 2.0m, CardNumber =
"1234123412341234" };
            var payment4 = new Payment { AmountToPay = 17.0m, CardNumber =
"1234123412341234" };
            var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
            var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
            var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
            var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
            var payment9 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

            CreateConnection();

            SendMessage(payment1);
            SendMessage(payment2);
            SendMessage(payment3);
            SendMessage(payment4);
            SendMessage(payment5);
            SendMessage(payment6);
            SendMessage(payment7);
            SendMessage(payment8);
            SendMessage(payment9);
            SendMessage(payment10);
        }

        private static void CreateConnection()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            _connection = _factory.CreateConnection();
            _model = _connection.CreateModel();
            _model.ExchangeDeclare(ExchangeName, "fanout");
        }

        private static void SendMessage(Payment message)
        {
            _model.BasicPublish(ExchangeName, "", null, message.Serialize());
            Console.WriteLine(" Payment Sent {0}, £{1}", message.CardNumber,
message.AmountToPay);
        }
    }
```

```
}
```

And here is the code for the subscriber application:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static QueueingBasicConsumer _consumer;

        private const string ExchangeName = "PublishSubscribe_Exchange";

        static void Main()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    var queueName = DeclareAndBindQueueToExchange(channel);
                    channel.BasicConsume(queueName, true, _consumer);

                    while (true)
                    {
                        var ea = _consumer.Queue.Dequeue();
                        var message = (Payment)ea.Body.DeSerialize();

                        Console.WriteLine("----- Payment Processed {0} : {1}",
message.CardNumber, message.AmountToPay);
                    }
                }
            }
        }

        private static string DeclareAndBindQueueToExchange(IModel channel)
        {
            channel.ExchangeDeclare(ExchangeName, "fanout");
            var queueName = channel.QueueDeclare().QueueName;
            channel.QueueBind(queueName, ExchangeName, "");
            _consumer = new QueueingBasicConsumer(channel);
            return queueName;
        }
    }
}
```

Example 4: Direct Routing

In this next example, we are going to produce a system that looks similar to the publish and subscribe demo we have just looked at. However, it is fundamentally different. As before, we will have a series of consuming applications that will connect to a queue that is bound to an exchange:

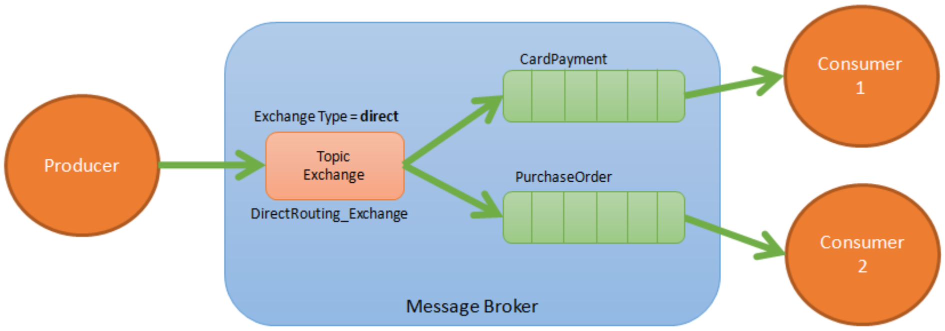


Figure 54. A Series of Consuming Applications

In the previous example, the exchange was set to use the “fanout” exchange type which meant all of the messages were routed to each of the subscribers. This meant that, if you provided a routing key when send a message to an exchange, that routing key would be ignored.

In this example, we will use a “direct” exchange type and the routing key will be used to direct messages to a specific consumer. The producer application in this example will post two different types of message. First is the CardPayment message with which we are already familiar, and the second type of message is a PurchaseOrder. As these are posted to the exchange, they will be sent with a specific routing key identifying what type of message they are (i.e. CardPayment or PurchaseOrder).

The example has two different consumer applications. One is specifically looking out for CardPayments and the other is only interested in PurchaseOrders. They pick up their messages based on the routing key. Let’s take a look at the code. First, as before, we set up our different payment messages. We have 10 card payments and 10 purchase orders:

```
static void Main()
{
    var payment1 = new Payment { AmountToPay = 25.0m, CardNumber = "1234123412341234"
};
    var payment2 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234"
};
    var payment3 = new Payment { AmountToPay = 2.0m, CardNumber = "1234123412341234"
};
    var payment4 = new Payment { AmountToPay = 17.0m, CardNumber = "1234123412341234"
};
    var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
    var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
```

```
    var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
    var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
    var payment9 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234"
};
    var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

    var purchaseOrder1 = new PurchaseOrder{AmountToPay = 50.0m, CompanyName =
"Company A", PaymentDayTerms = 75, PoNumber = "123434A"};
    var purchaseOrder2 = new PurchaseOrder { AmountToPay = 150.0m, CompanyName =
"Company B", PaymentDayTerms = 75, PoNumber = "193434B" };
    var purchaseOrder3 = new PurchaseOrder { AmountToPay = 12.0m, CompanyName =
"Company C", PaymentDayTerms = 75, PoNumber = "196544A" };
    var purchaseOrder4 = new PurchaseOrder { AmountToPay = 2150.0m, CompanyName =
"Company D", PaymentDayTerms = 75, PoNumber = "234434H" };
    var purchaseOrder5 = new PurchaseOrder { AmountToPay = 2150.0m, CompanyName =
"Company E", PaymentDayTerms = 75, PoNumber = "876434W" };
    var purchaseOrder6 = new PurchaseOrder { AmountToPay = 7150.0m, CompanyName =
"Company F", PaymentDayTerms = 75, PoNumber = "1423474U" };
    var purchaseOrder7 = new PurchaseOrder { AmountToPay = 3150.0m, CompanyName =
"Company G", PaymentDayTerms = 75, PoNumber = "1932344O" };
    var purchaseOrder8 = new PurchaseOrder { AmountToPay = 3190.0m, CompanyName =
"Company H", PaymentDayTerms = 75, PoNumber = "1123457Q" };
    var purchaseOrder9 = new PurchaseOrder { AmountToPay = 50.0m, CompanyName =
"Company I", PaymentDayTerms = 75, PoNumber =   "1595344R" };
    var purchaseOrder10 = new PurchaseOrder { AmountToPay = 2150.0m, CompanyName =
"Company J", PaymentDayTerms = 75, PoNumber = "656734L" };

    CreateConnection();

    SendPayment(payment1);
    SendPayment(payment2);
    SendPayment(payment3);
    SendPayment(payment4);
    SendPayment(payment5);
    SendPayment(payment6);
    SendPayment(payment7);
    SendPayment(payment8);
    SendPayment(payment9);
    SendPayment(payment10);

    SendPurchaseOrder(purchaseOrder1);
    SendPurchaseOrder(purchaseOrder2);
    SendPurchaseOrder(purchaseOrder3);
    SendPurchaseOrder(purchaseOrder4);
    SendPurchaseOrder(purchaseOrder5);
    SendPurchaseOrder(purchaseOrder6);
    SendPurchaseOrder(purchaseOrder7);
    SendPurchaseOrder(purchaseOrder8);
    SendPurchaseOrder(purchaseOrder9);
    SendPurchaseOrder(purchaseOrder10);
}
```

Next, we create the connection, channel, and then the exchange. This time, the exchange is named “DirectRouting_Exchange” and the exchange type is set to “direct” instead of “fanout.”

```
private static void CreateConnection()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    _connection = _factory.CreateConnection();
    _model = _connection.CreateModel();
    _model.ExchangeDeclare(ExchangeName, "direct");
}
```

Now that the connection, channel, and exchange are ready to go, we can send the card payments and purchase orders to the exchange:

```
private static void SendPayment(Payment payment)
{
    SendMessage(payment.Serialize(), "CardPayment");
    Console.WriteLine(" Payment Sent {0}, £{1}", payment.CardNumber,
payment.AmountToPay);
}

private static void SendPurchaseOrder(PurchaseOrder purchaseOrder)
{
    SendMessage(purchaseOrder.Serialize(), "PurchaseOrder");
    Console.WriteLine(" Purchase Order Sent {0}, £{1}, {2}, {3}",
purchaseOrder.CompanyName, purchaseOrder.AmountToPay, purchaseOrder.PaymentDayTerms,
purchaseOrder.PoNumber);
}

private static void SendMessage(byte[] message, string routingKey)
{
    _model.BasicPublish(ExchangeName, routingKey, null, message);
}
```

The ***SendPayment*** and ***SendPurchaseOrder*** methods are similar—except for the type of object that is serialized to send to the queue, and the routing key that is specified to tell the exchange to which consumer the message should be routed.

Now let’s take a look at one of the consumers. Both of the consumers are similar in how they work. We will discuss the consumer for the card payment processor and highlight the differences for the purchase order version.

```
static void Main()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
```

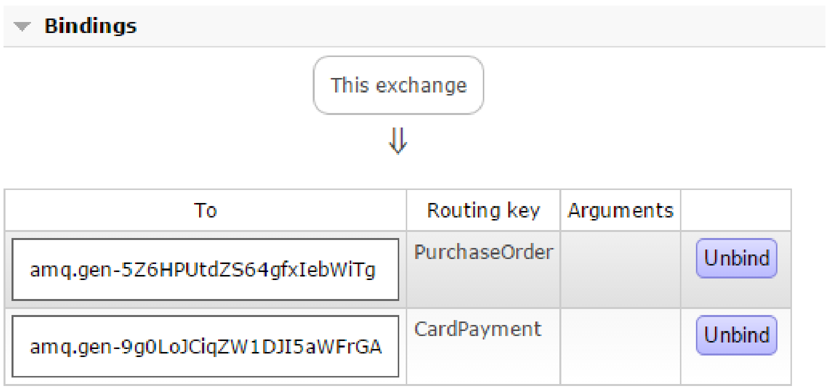
```
    using (_connection = _factory.CreateConnection())
    {
        using (var channel = _connection.CreateModel())
        {
            channel.ExchangeDeclare(ExchangeName, "direct");
            var queueName = channel.QueueDeclare().QueueName;

            channel.QueueBind(queueName, ExchangeName, "CardPayment");

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

            while (true)
            {
                var ea = consumer.Queue.Dequeue();
                var message = (Payment)ea.Body.DeSerialize();
                var routingKey = ea.RoutingKey;
                Console.WriteLine("--- Payment - Key <{0}> : {1} : {2}", routingKey,
message.CardNumber, message.AmountToPay);
            }
        }
    }
}
```

As with the other examples, the code creates a connection and channel, and then redeclares the exchange as a direct exchange. Next, a queue is created by using a RabbitMQ-generated queue name. This queue is then bound to the exchange. The third parameter to the ***QueueDeclare*** method is the routing key. For the first consumer, this is set to “***CardPayment***” as it is just interested in the ***CardPayment*** messages.

For the second consumer, the queue is bound to the exchange by using the “***PurchaseOrder***” routing key. Next, the ***QueueingBasicConsumer*** is created and ***BasicConsume*** is called to start, reading from the queue. Again, for this example, we are not sending a message acknowledgement so the second parameter to ***BasicConsume*** is set to ***true***.

Then, we enter a while loop, and the message is dequeued and deserialized back into a ***Payment*** object for the ***CardPayment*** consumer and a ***PurchaseOrder*** consumer.

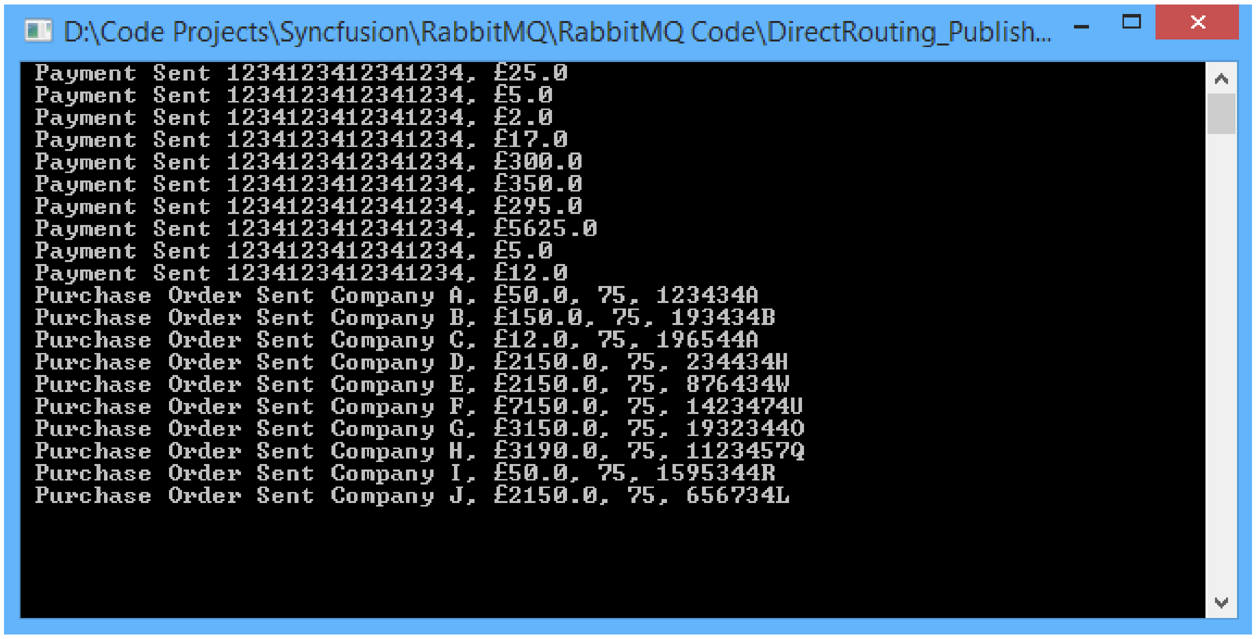
Next, run both of the consumer applications on their own, and then go to the RabbitMQ management portal and browse to the “DirectRouting_Exchange.” You will see that there are two queues bound to the exchange. The first queue has a routing key of ***PurchaseOrder*** set against it and the second queue has a routing key of ***CardPayment*** set. If you see this, you are ready to post messages to the exchange.



To	Routing key	Arguments	
amq.gen-5Z6HPUtdZS64gfxIebWiTg	PurchaseOrder		Unbind
amq.gen-9g0LoJCiqZW1DJI5aWFrGA	CardPayment		Unbind

Figure 55: Direct routing queues bound to the exchange with different routing keys

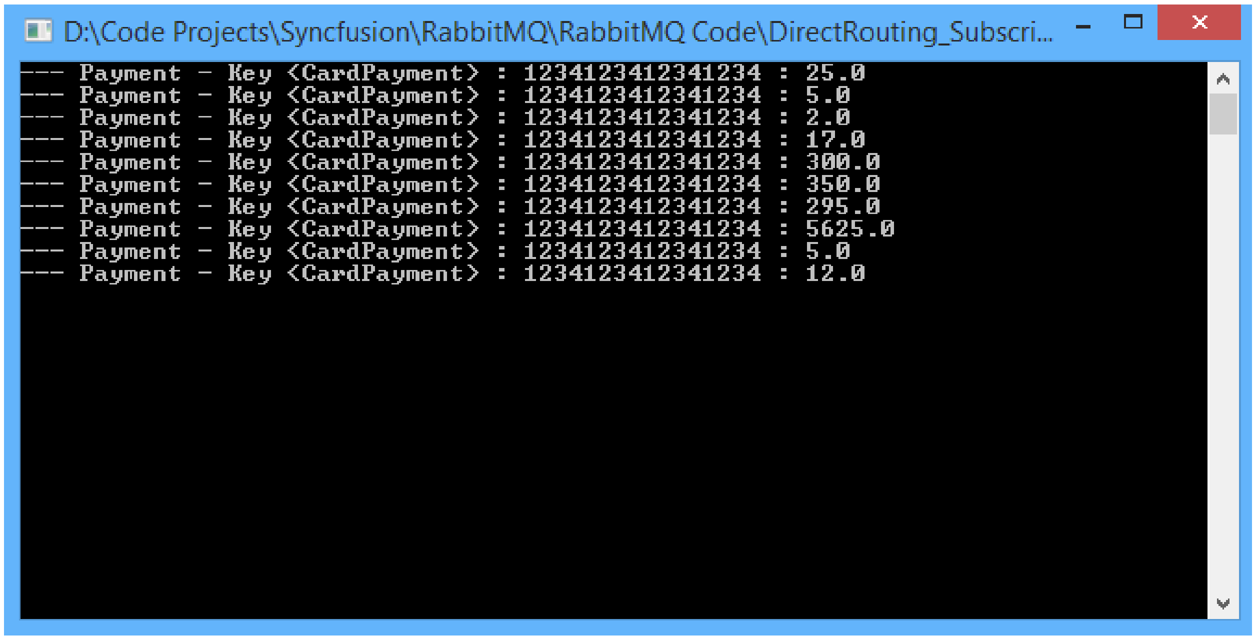
Now, if you run the producer application, you will see that the 10 payments and 10 purchase orders have been sent to the exchange as shown in the following screenshot:


D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\DirectRouting_Publish...

```
Payment Sent 1234123412341234, £25.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £2.0
Payment Sent 1234123412341234, £17.0
Payment Sent 1234123412341234, £300.0
Payment Sent 1234123412341234, £350.0
Payment Sent 1234123412341234, £295.0
Payment Sent 1234123412341234, £5625.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £12.0
Purchase Order Sent Company A, £50.0, 75, 123434A
Purchase Order Sent Company B, £150.0, 75, 193434B
Purchase Order Sent Company C, £12.0, 75, 196544A
Purchase Order Sent Company D, £2150.0, 75, 234434H
Purchase Order Sent Company E, £2150.0, 75, 876434W
Purchase Order Sent Company F, £7150.0, 75, 1423474U
Purchase Order Sent Company G, £3150.0, 75, 19323440
Purchase Order Sent Company H, £3190.0, 75, 11234570
Purchase Order Sent Company I, £50.0, 75, 1595344R
Purchase Order Sent Company J, £2150.0, 75, 656734L
```

Figure 56: Payments and purchase orders have been sent to the exchange and routed via a routing key

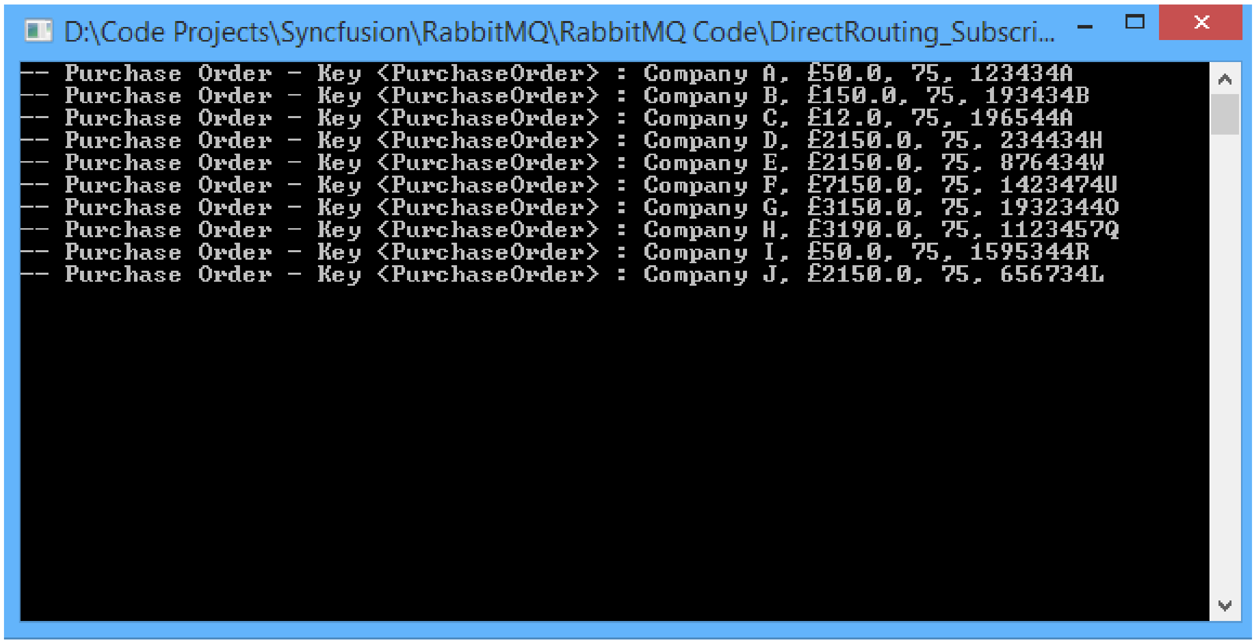
Then, in the first consumer application, you will see that all of the payment messages have been routed to it as shown in the following screenshot:



```
D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\DirectRouting_Subscri...
--- Payment - Key <CardPayment> : 1234123412341234 : 25.0
--- Payment - Key <CardPayment> : 1234123412341234 : 5.0
--- Payment - Key <CardPayment> : 1234123412341234 : 2.0
--- Payment - Key <CardPayment> : 1234123412341234 : 17.0
--- Payment - Key <CardPayment> : 1234123412341234 : 300.0
--- Payment - Key <CardPayment> : 1234123412341234 : 350.0
--- Payment - Key <CardPayment> : 1234123412341234 : 295.0
--- Payment - Key <CardPayment> : 1234123412341234 : 5625.0
--- Payment - Key <CardPayment> : 1234123412341234 : 5.0
--- Payment - Key <CardPayment> : 1234123412341234 : 12.0
```

Figure 57: The card payment consumer has only picked up the payment messages

Then, in the other consumer window, you will see all of the purchase order messages:



```
D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\DirectRouting_Subscri...
-- Purchase Order - Key <PurchaseOrder> : Company A, £50.0, 75, 123434A
-- Purchase Order - Key <PurchaseOrder> : Company B, £150.0, 75, 193434B
-- Purchase Order - Key <PurchaseOrder> : Company C, £12.0, 75, 196544A
-- Purchase Order - Key <PurchaseOrder> : Company D, £2150.0, 75, 234434H
-- Purchase Order - Key <PurchaseOrder> : Company E, £2150.0, 75, 876434W
-- Purchase Order - Key <PurchaseOrder> : Company F, £7150.0, 75, 1423474U
-- Purchase Order - Key <PurchaseOrder> : Company G, £3150.0, 75, 19323440
-- Purchase Order - Key <PurchaseOrder> : Company H, £3190.0, 75, 1123457Q
-- Purchase Order - Key <PurchaseOrder> : Company I, £50.0, 75, 1595344R
-- Purchase Order - Key <PurchaseOrder> : Company J, £2150.0, 75, 656734L
```

Figure 58: The purchase consumer has only picked up the purchase order messages

Here is the complete code for the publisher application:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static IModel _model;

        private const string ExchangeName = "DirectRouting_Exchange";

        static void Main()
        {
            var payment1 = new Payment { AmountToPay = 25.0m, CardNumber =
"1234123412341234" };
            var payment2 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment3 = new Payment { AmountToPay = 2.0m, CardNumber =
"1234123412341234" };
            var payment4 = new Payment { AmountToPay = 17.0m, CardNumber =
"1234123412341234" };
            var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234" };
            var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234" };
            var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234" };
            var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234" };
            var payment9 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234" };
            var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234" };

            var purchaseOrder1 = new PurchaseOrder{AmountToPay = 50.0m, CompanyName =
"Company A", PaymentDayTerms = 75, PoNumber = "123434A"};
            var purchaseOrder2 = new PurchaseOrder { AmountToPay = 150.0m,
CompanyName = "Company B", PaymentDayTerms = 75, PoNumber = "193434B" };
            var purchaseOrder3 = new PurchaseOrder { AmountToPay = 12.0m, CompanyName
= "Company C", PaymentDayTerms = 75, PoNumber = "196544A" };
            var purchaseOrder4 = new PurchaseOrder { AmountToPay = 2150.0m,
CompanyName = "Company D", PaymentDayTerms = 75, PoNumber = "234434H" };
            var purchaseOrder5 = new PurchaseOrder { AmountToPay = 2150.0m,
CompanyName = "Company E", PaymentDayTerms = 75, PoNumber = "876434W" };
            var purchaseOrder6 = new PurchaseOrder { AmountToPay = 7150.0m,
CompanyName = "Company F", PaymentDayTerms = 75, PoNumber = "1423474U" };
            var purchaseOrder7 = new PurchaseOrder { AmountToPay = 3150.0m,
CompanyName = "Company G", PaymentDayTerms = 75, PoNumber = "1932344O" };
            var purchaseOrder8 = new PurchaseOrder { AmountToPay = 3190.0m,
CompanyName = "Company H", PaymentDayTerms = 75, PoNumber = "1123457Q" };
            var purchaseOrder9 = new PurchaseOrder { AmountToPay = 50.0m, CompanyName
= "Company I", PaymentDayTerms = 75, PoNumber =   "1595344R" };
            var purchaseOrder10 = new PurchaseOrder { AmountToPay = 2150.0m,
CompanyName = "Company J", PaymentDayTerms = 75, PoNumber = "656734L" };
```

```
            CreateConnection();

            SendPayment(payment1);
            SendPayment(payment2);
            SendPayment(payment3);
            SendPayment(payment4);
            SendPayment(payment5);
            SendPayment(payment6);
            SendPayment(payment7);
            SendPayment(payment8);
            SendPayment(payment9);
            SendPayment(payment10);

            SendPurchaseOrder(purchaseOrder1);
            SendPurchaseOrder(purchaseOrder2);
            SendPurchaseOrder(purchaseOrder3);
            SendPurchaseOrder(purchaseOrder4);
            SendPurchaseOrder(purchaseOrder5);
            SendPurchaseOrder(purchaseOrder6);
            SendPurchaseOrder(purchaseOrder7);
            SendPurchaseOrder(purchaseOrder8);
            SendPurchaseOrder(purchaseOrder9);
            SendPurchaseOrder(purchaseOrder10);
        }

        private static void SendPayment(Payment payment)
        {
            SendMessage(payment.Serialize(), "CardPayment");
            Console.WriteLine(" Payment Sent {0}, £{1}", payment.CardNumber, 
payment.AmountToPay);
        }

        private static void SendPurchaseOrder(PurchaseOrder purchaseOrder)
        {
            SendMessage(purchaseOrder.Serialize(), "PurchaseOrder");
            Console.WriteLine(" Purchase Order Sent {0}, £{1}, {2}, {3}", 
purchaseOrder.CompanyName, purchaseOrder.AmountToPay, purchaseOrder.PaymentDayTerms, 
purchaseOrder.PoNumber);
        }

        private static void CreateConnection()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName = 
"guest", Password = "guest" };
            _connection = _factory.CreateConnection();
            _model = _connection.CreateModel();
            _model.ExchangeDeclare(ExchangeName, "direct");
        }

        private static void SendMessage(byte[] message, string routingKey)
        {
            _model.BasicPublish(ExchangeName, routingKey, null, message);
        }
    }
```

```
}
```

Here is the complete code for the card payment consumer application:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;

        private const string ExchangeName = "DirectRouting_Exchange";

        static void Main()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    channel.ExchangeDeclare(ExchangeName, "direct");
                    var queueName = channel.QueueDeclare().QueueName;

                    channel.QueueBind(queueName, ExchangeName, "CardPayment");

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

                    while (true)
                    {
                        var ea = consumer.Queue.Dequeue();
                        var message = (Payment)ea.Body.DeSerialize();
                        var routingKey = ea.RoutingKey;
                        Console.WriteLine("--- Payment - Key <{0}> : {1} : {2}",
routingKey, message.CardNumber, message.AmountToPay);
                    }
                }
            }
        }
    }
}
```

Here is the complete code for the purchase order consumer application:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;

        private const string ExchangeName = "DirectRouting_Exchange";

        static void Main()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName = 
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    channel.ExchangeDeclare(ExchangeName, "direct");
                    var queueName = channel.QueueDeclare().QueueName;

                    channel.QueueBind(queueName, ExchangeName, "PurchaseOrder");

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

                    while (true)
                    {
                        var ea = consumer.Queue.Dequeue();
                        var message = (PurchaseOrder)ea.Body.DeSerialize();
                        var routingKey = ea.RoutingKey;
                        Console.WriteLine("-- Purchase Order - Key <{0}> : {1}, £{2}, 
{3}, {4}", routingKey, message.CompanyName, message.AmountToPay, 
message.PaymentDayTerms, message.PoNumber);
                    }
                }
            }
        }
    }
}
```

Example 5: Topic-based Publish and Subscribe

In the last example, we routed different types of messages to specific consumers through a direct exchange by using the routing key. In this next example, we are going to do something similar but we are going to use topics instead. When the exchange is defined, instead of setting its type to “direct,” we will set the exchange type to “topic.”

Messages sent to a topic exchange can't have an arbitrary routing key as we saw earlier. The routing key must be a list of words that are separated by dots. The words can be anything but usually they specify some features connected to the message. In the context of our payment processor scenario, we will have routing keys such as "payment.card" and "payment.purchaseorder.” There can be as many words in the routing key as you like, up to the limit of 255 bytes.

The logic behind the topic exchange is similar to a direct exchange as seen in the previous example. A message sent with a particular routing key will be delivered to all of the queues that are bound with a matching key. However, there are two important special cases for binding keys:

- A ‘*’ (star) can substitute for exactly one word.
- A ‘#’ (hash) can substitute for zero or more words.

Look at the following diagram as an example:

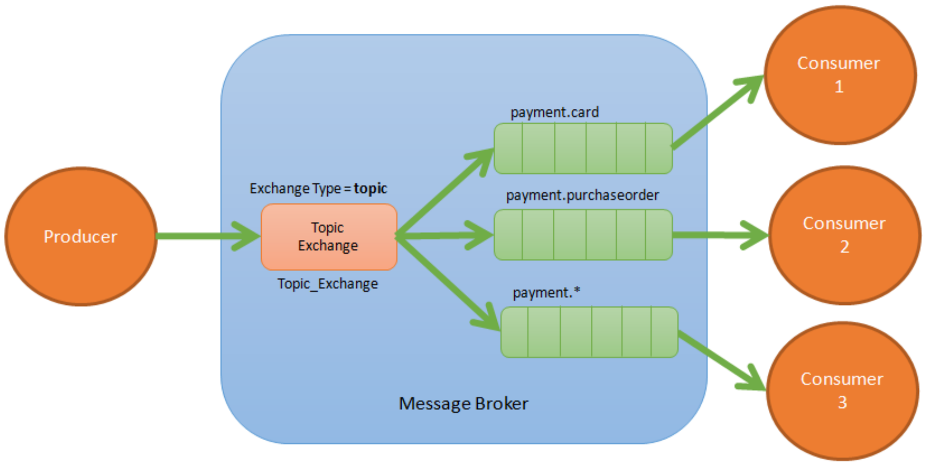


Figure 59: Topic-based publisher and subscriber example

Consumer 1 and Consumer 2 both receive card and purchase order messages as seen in the previous example; the difference here is that the routing key looks different. The real difference is when we look at Consumer 3. This has a routing key of “payment.*”

This means that Consumer 3 is interested in any message that starts with “payment.” So, for this example, we would expect both card payments and purchase orders to be picked up by this consumer.

If we apply this to a real-world scenario, Consumers 1 and 2 may be actual services processing payments whereas Consumer 3 could be a hook into an accounting system where someone is interested in getting visibility of all of the payments (regardless of whether or not they are card payments or purchase orders).

Let’s now take a look at the code for the publisher application:

```
static void Main()
{
    var payment1 = new Payment { AmountToPay = 25.0m, CardNumber =
"1234123412341234", Name = "Mr F Bloggs"};
    var payment2 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234",
Name = "Mr S Simpson" };
    var payment3 = new Payment { AmountToPay = 2.0m, CardNumber = "1234123412341234",
Name = "Mr G Washington" };
    var payment4 = new Payment { AmountToPay = 17.0m, CardNumber =
"1234123412341234", Name = "Mr B Gates" };
    var payment5 = new Payment { AmountToPay = 300.0m, CardNumber =
"1234123412341234", Name = "Mrs K Kardashian" };
    var payment6 = new Payment { AmountToPay = 350.0m, CardNumber =
"1234123412341234", Name = "Mrs M Whitehouse" };
    var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234", Name = "Mrs E Windsor" };
    var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234", Name = "Mr B Obama" };
    var payment9 = new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234",
Name = "Mr S Haunts" };
    var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234", Name = "Mr F Bloggs Jr" };

    var purchaseOrder1 = new PurchaseOrder { AmountToPay = 50.0m, CompanyName =
"Company A", PaymentDayTerms = 75, PoNumber = "123434A" };
    var purchaseOrder2 = new PurchaseOrder { AmountToPay = 150.0m, CompanyName =
"Company B", PaymentDayTerms = 75, PoNumber = "193434B" };
    var purchaseOrder3 = new PurchaseOrder { AmountToPay = 12.0m, CompanyName =
"Company C", PaymentDayTerms = 75, PoNumber = "196544A" };
    var purchaseOrder4 = new PurchaseOrder { AmountToPay = 2150.0m, CompanyName =
"Company D", PaymentDayTerms = 75, PoNumber = "234434H" };
    var purchaseOrder5 = new PurchaseOrder { AmountToPay = 2150.0m, CompanyName =
"Company E", PaymentDayTerms = 75, PoNumber = "876434W" };
    var purchaseOrder6 = new PurchaseOrder { AmountToPay = 7150.0m, CompanyName =
"Company F", PaymentDayTerms = 75, PoNumber = "1423474U" };
    var purchaseOrder7 = new PurchaseOrder { AmountToPay = 3150.0m, CompanyName =
"Company G", PaymentDayTerms = 75, PoNumber = "1932344O" };
    var purchaseOrder8 = new PurchaseOrder { AmountToPay = 3190.0m, CompanyName =
"Company H", PaymentDayTerms = 75, PoNumber = "1123457Q" };
    var purchaseOrder9 = new PurchaseOrder { AmountToPay = 50.0m, CompanyName =
"Company I", PaymentDayTerms = 75, PoNumber = "1595344R" };
    var purchaseOrder10 = new PurchaseOrder { AmountToPay = 2150.0m, CompanyName =
"Company J", PaymentDayTerms = 75, PoNumber = "656734L" };

    CreateConnection();
    SendPayment(payment1);
    SendPayment(payment2);
    SendPayment(payment3);
    SendPayment(payment4);
    SendPayment(payment5);
    SendPayment(payment6);
    SendPayment(payment7);
    SendPayment(payment8);
    SendPayment(payment9);
    SendPayment(payment10);

    SendPurchaseOrder(purchaseOrder1);
    SendPurchaseOrder(purchaseOrder2);
    SendPurchaseOrder(purchaseOrder3);
    SendPurchaseOrder(purchaseOrder4);
    SendPurchaseOrder(purchaseOrder5);
    SendPurchaseOrder(purchaseOrder6);
    SendPurchaseOrder(purchaseOrder7);
    SendPurchaseOrder(purchaseOrder8);
    SendPurchaseOrder(purchaseOrder9);
    SendPurchaseOrder(purchaseOrder10);
}
```

This is similar to the previous direct routing example. We set up 10 card payment messages and then 10 purchase order messages. Next, we create the connection and channel, and define an exchange called “Topic_Exchange”:

```
private static void CreateConnection()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    _connection = _factory.CreateConnection();
    _model = _connection.CreateModel();
    _model.ExchangeDeclare(ExchangeName, "topic");
}
```

When the exchange is declared, the exchange type of “topic” is used. Next, there is the code to send the payment and purchase order messages to the exchange. You will see in the following example that a payment object is serialized and sent to the exchange with the “payment.card” routing key. The purchase order object is serialized and sent with the “payment.purchaseorder” routing key:

```
private static void SendPayment(Payment payment)
{
    SendMessage(payment.Serialize(), "payment.card");
    Console.WriteLine(" Payment Sent {0}, £{1}", payment.CardNumber,
payment.AmountToPay);
}
```

```
private static void SendPurchaseOrder(PurchaseOrder purchaseOrder)
{
    SendMessage(purchaseOrder.Serialize(), "payment.purchaseorder");
    Console.WriteLine(" Purchase Order Sent {0}, £{1}, {2}, {3}",
purchaseOrder.CompanyName, purchaseOrder.AmountToPay, purchaseOrder.PaymentDayTerms,
purchaseOrder.PoNumber);
}

private static void SendMessage(byte[] message, string routingKey)
{
    _model.BasicPublish(ExchangeName, routingKey, null, message);
}
```

For this example, there are three consumer applications: one that reads messages from the “payment.card” queue, another that reads from “payment.purchaseorder,” and the third that reads everything from ‘payment.*’ which includes card payments and purchase orders.

In the following explaination, we will treat the first two consumer applications as the same but call out the changes between “payment.cards” and “payment.purchaseorder”:

```
static void Main()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    using (_connection = _factory.CreateConnection())
    {
        using (var channel = _connection.CreateModel())
        {
            Console.WriteLine("Publisher listening for Topic <payment.card>");
            Console.WriteLine("--------------------------------------------");
            Console.WriteLine();

            channel.ExchangeDeclare(ExchangeName, "topic");
            var queueName = channel.QueueDeclare().QueueName;

            channel.QueueBind(queueName, ExchangeName, "payment.card");

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

            while (true)
            {
                var ea = consumer.Queue.Dequeue();
                var message = (Payment)ea.Body.DeSerialize();
                var routingKey = ea.RoutingKey;
                Console.WriteLine("--- Payment - Routing Key <{0}> : {1} : {2}",
routingKey, message.CardNumber, message.AmountToPay);
            }
        }
    }
}
```

The preceding code is for the first consumer application. As in previous examples, the connection is made, the channel created, and the exchange "Topic_Exchange" (with a type of "topic") is redeclared. Then, a queue for this application is declared and bound to the exchange with a routing key of "payment.card."

This is what ensures that this consumer only listens to card payment messages from the exchange. If you want to receive purchase order messages, you bind the queue to the exchange with a routing key of "payment.purchaseorder."

Once the queue is bound to the exchange, a ***QueuingBasicConsumer*** is created for the channel and ***BasicConsume*** is called to start the read from the queue. Once this has happened, the application goes into a while loop where the messages are dequeued and deserialized. For the "payment.card" message, the message is deserialized to the ***Payment*** object. For the purchase order messages, the message is deserialized to the ***PurchaseOrder*** object type.

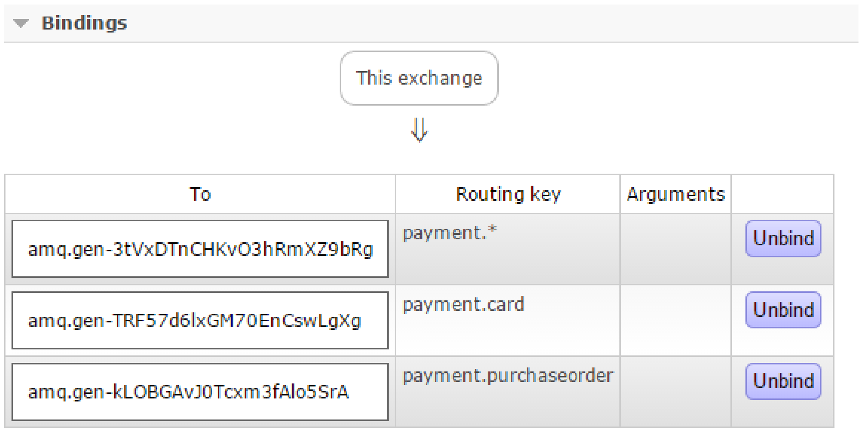
The third consumer is designed to read all of the payment messages regardless of whether they are card payments or purchase orders. The first change is where we bind the consumer queue to the exchange and specify a routing key of "payment.*":

```
channel.QueueBind(queueName, ExchangeName, "payment.*");
```

The second change is in the while loop when the messages are dequeued. Once the message has been deserialized, a type check is performed to see if the message is a ***PurchaseOrder*** or a ***Payment*** and if the relevant message is written to the console output:

```
while (true)
{
    var ea = consumer.Queue.Dequeue();
    var reference = ea.Body.DeSerialize();

    var order = reference as PurchaseOrder;
    if (order != null)
    {
        Console.WriteLine("Purchase Order Received from company '{0}'",
order.CompanyName);
    }

    var payment = reference as Payment;
    if (payment != null)
    {
        Console.WriteLine("Card Payment Received from person '{0}'", payment.Name);
    }
}
```

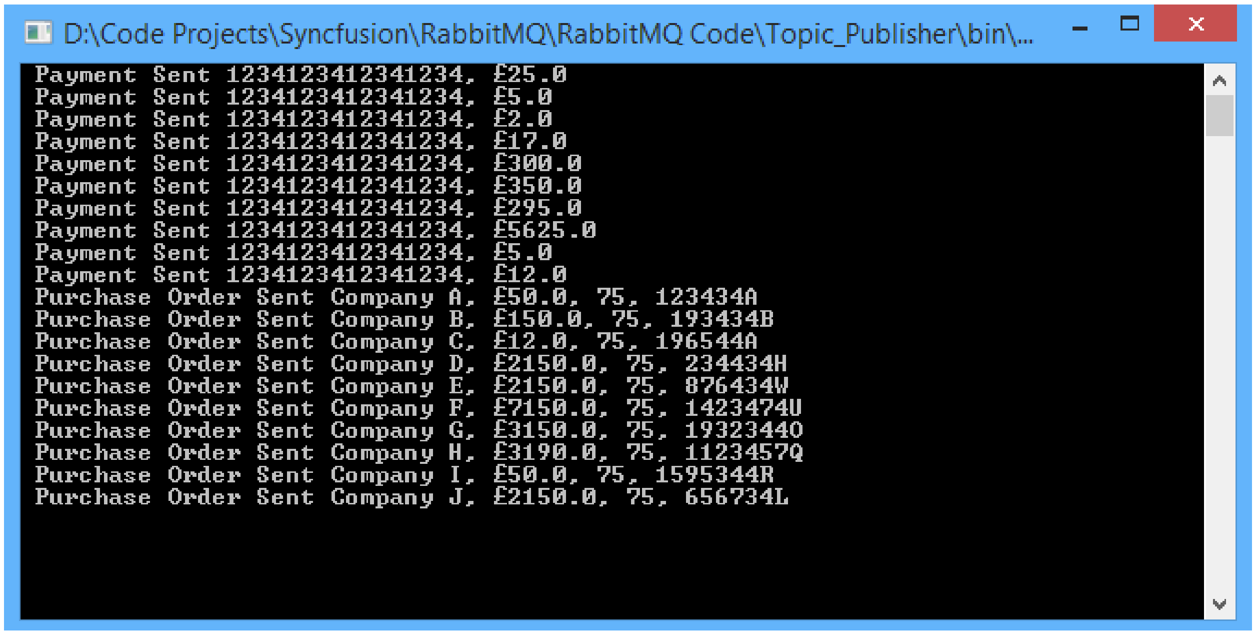
If you run all three of the consumer applications and then log onto the RabbitMQ management portal and browse to “Topic_Exchange,” you will see each of the queues bound to the exchange. Each queue will have its appropriate routing key assigned to the queue as shown in the following screenshot:



To	Routing key	Arguments	
amq.gen-3tVxDTnCHKvO3hRmXZ9bRg	payment.*		Unbind
amq.gen-TRF57d6lxGM70EnCswLgXg	payment.card		Unbind
amq.gen-kLOBGAvJ0Tcxm3fAlo5SrA	payment.purchaseorder		Unbind

Figure 60: Topic exchange with three consumer queues bound to it

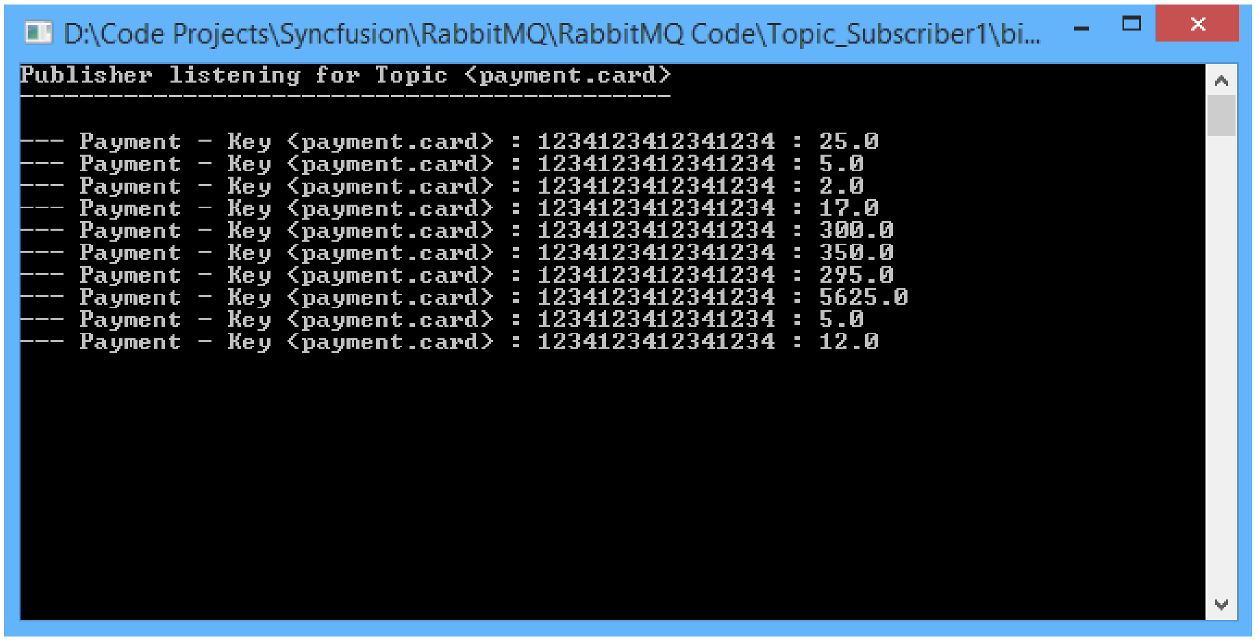
If you now run the producer application, you will see the 10 card payments and 10 purchase orders sent to the exchange:


D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\Topic_Publisher\bin\...

```
Payment Sent 1234123412341234, £25.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £2.0
Payment Sent 1234123412341234, £17.0
Payment Sent 1234123412341234, £300.0
Payment Sent 1234123412341234, £350.0
Payment Sent 1234123412341234, £295.0
Payment Sent 1234123412341234, £5625.0
Payment Sent 1234123412341234, £5.0
Payment Sent 1234123412341234, £12.0
Purchase Order Sent Company A, £50.0, 75, 123434A
Purchase Order Sent Company B, £150.0, 75, 193434B
Purchase Order Sent Company C, £12.0, 75, 196544A
Purchase Order Sent Company D, £2150.0, 75, 234434H
Purchase Order Sent Company E, £2150.0, 75, 876434W
Purchase Order Sent Company F, £7150.0, 75, 1423474U
Purchase Order Sent Company G, £3150.0, 75, 19323440
Purchase Order Sent Company H, £3190.0, 75, 1123457Q
Purchase Order Sent Company I, £50.0, 75, 1595344R
Purchase Order Sent Company J, £2150.0, 75, 656734L
```

Figure 61: Topic example, consumer application after posting payments and purchase orders

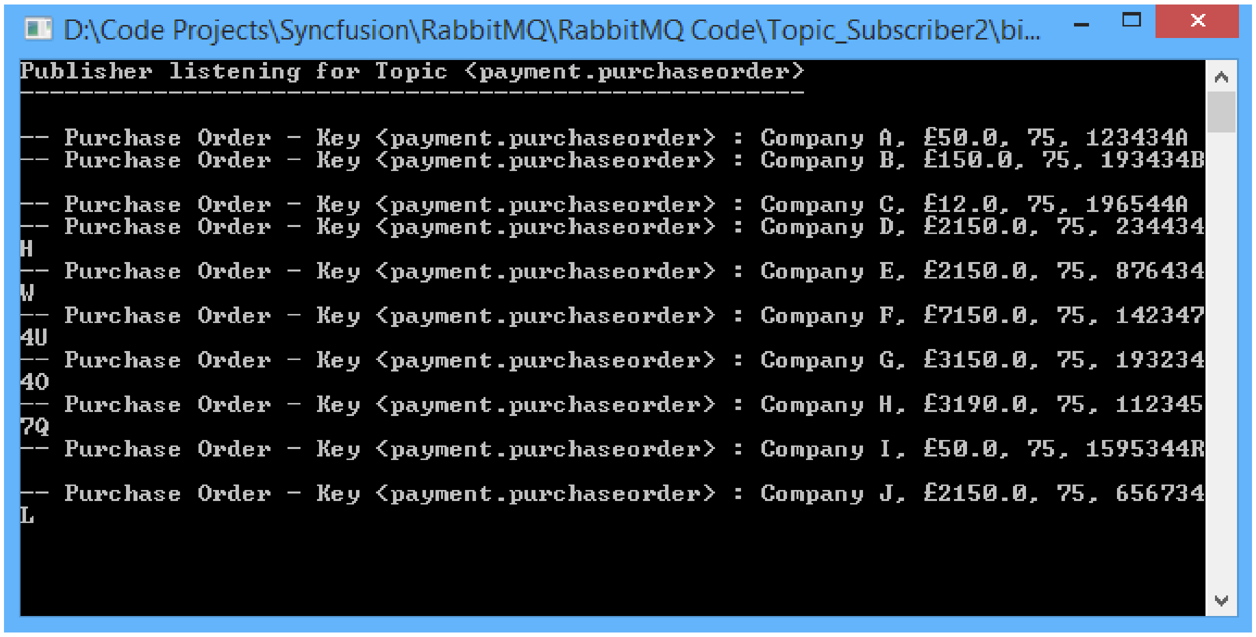
Once these messages have been sent to the exchange, they will be routed to the relevant queues based upon their routing key. In the following screenshot, the consumer application has picked up the messages for card payments:



```
D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\Topic_Subscriber1\bi...
Publisher listening for Topic <payment.card>
--------------------------------------------

--- Payment - Key <payment.card> : 1234123412341234 : 25.0
--- Payment - Key <payment.card> : 1234123412341234 : 5.0
--- Payment - Key <payment.card> : 1234123412341234 : 2.0
--- Payment - Key <payment.card> : 1234123412341234 : 17.0
--- Payment - Key <payment.card> : 1234123412341234 : 300.0
--- Payment - Key <payment.card> : 1234123412341234 : 350.0
--- Payment - Key <payment.card> : 1234123412341234 : 295.0
--- Payment - Key <payment.card> : 1234123412341234 : 5625.0
--- Payment - Key <payment.card> : 1234123412341234 : 5.0
--- Payment - Key <payment.card> : 1234123412341234 : 12.0
```

Figure 62: Topic example, consumer application receiving card payments

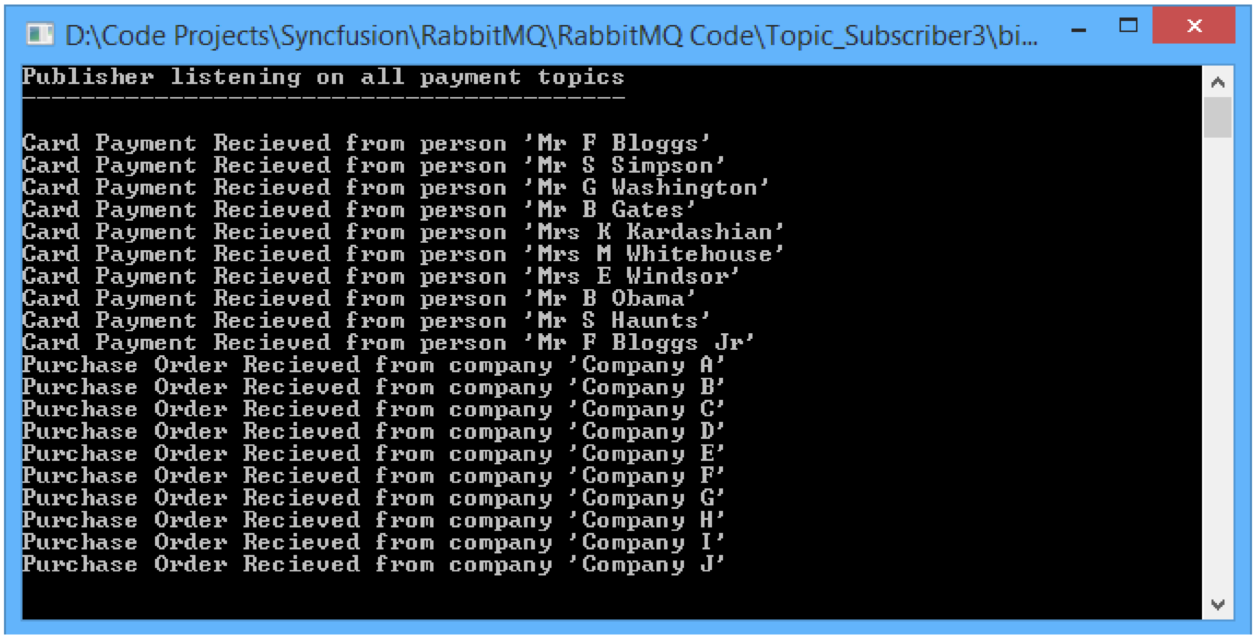
Then, in the following screenshot, we have the consumer application that receives all of the purchase orders from the exchange:



```
D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\Topic_Subscriber2\bi...
Publisher listening for Topic <payment.purchaseorder>
-----------------------------------------------------

-- Purchase Order - Key <payment.purchaseorder> : Company A, £50.0, 75, 123434A
-- Purchase Order - Key <payment.purchaseorder> : Company B, £150.0, 75, 193434B

-- Purchase Order - Key <payment.purchaseorder> : Company C, £12.0, 75, 196544A
-- Purchase Order - Key <payment.purchaseorder> : Company D, £2150.0, 75, 234434
H
-- Purchase Order - Key <payment.purchaseorder> : Company E, £2150.0, 75, 876434
W
-- Purchase Order - Key <payment.purchaseorder> : Company F, £7150.0, 75, 142347
4U
-- Purchase Order - Key <payment.purchaseorder> : Company G, £3150.0, 75, 193234
40
-- Purchase Order - Key <payment.purchaseorder> : Company H, £3190.0, 75, 112345
7Q
-- Purchase Order - Key <payment.purchaseorder> : Company I, £50.0, 75, 1595344R

-- Purchase Order - Key <payment.purchaseorder> : Company J, £2150.0, 75, 656734
L
```

Figure 63: Topic example, consumer application receiving purchase orders

Then, finally, in the following final screenshot, we have the consumer application that receives the entire collection card payments and purchase orders:


D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\Topic_Subscriber3\bi...

```
Publisher listening on all payment topics
-----------------------------------------

Card Payment Recieved from person 'Mr F Bloggs'
Card Payment Recieved from person 'Mr S Simpson'
Card Payment Recieved from person 'Mr G Washington'
Card Payment Recieved from person 'Mr B Gates'
Card Payment Recieved from person 'Mrs K Kardashian'
Card Payment Recieved from person 'Mrs M Whitehouse'
Card Payment Recieved from person 'Mrs E Windsor'
Card Payment Recieved from person 'Mr B Obama'
Card Payment Recieved from person 'Mr S Haunts'
Card Payment Recieved from person 'Mr F Bloggs Jr'
Purchase Order Recieved from company 'Company A'
Purchase Order Recieved from company 'Company B'
Purchase Order Recieved from company 'Company C'
Purchase Order Recieved from company 'Company D'
Purchase Order Recieved from company 'Company E'
Purchase Order Recieved from company 'Company F'
Purchase Order Recieved from company 'Company G'
Purchase Order Recieved from company 'Company H'
Purchase Order Recieved from company 'Company I'
Purchase Order Recieved from company 'Company J'
```

Figure 64: Topic example, consumer application receiving both card payments and purchase orders

The following code is for the producer application that places messages onto the topic-based exchange:

```
using RabbitMQ.Client;
using System;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static IModel _model;

        private const string ExchangeName = "Topic_Exchange";

        static void Main()
        {
            var payment1 = new Payment { AmountToPay = 25.0m, CardNumber = 
"1234123412341234", Name = "Mr F Bloggs"};
            var payment2 = new Payment { AmountToPay = 5.0m, CardNumber = 
"1234123412341234", Name = "Mr S Simpson" };
            var payment3 = new Payment { AmountToPay = 2.0m, CardNumber = 
"1234123412341234", Name = "Mr G Washington" };
            var payment4 = new Payment { AmountToPay = 17.0m, CardNumber = 
"1234123412341234", Name = "Mr B Gates" };
            var payment5 = new Payment { AmountToPay = 300.0m, CardNumber = 
"1234123412341234", Name = "Mrs K Kardashian" };
            var payment6 = new Payment { AmountToPay = 350.0m, CardNumber = 
"1234123412341234", Name = "Mrs M Whitehouse" };
            var payment7 = new Payment { AmountToPay = 295.0m, CardNumber =
"1234123412341234", Name = "Mrs E Windsor" };
            var payment8 = new Payment { AmountToPay = 5625.0m, CardNumber =
"1234123412341234", Name = "Mr B Obama" };
            var payment9 = new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234", Name = "Mr S Haunts" };
            var payment10 = new Payment { AmountToPay = 12.0m, CardNumber =
"1234123412341234", Name = "Mr F Bloggs Jr" };

            var purchaseOrder1 = new PurchaseOrder { AmountToPay = 50.0m, CompanyName
= "Company A", PaymentDayTerms = 75, PoNumber = "123434A" };
            var purchaseOrder2 = new PurchaseOrder { AmountToPay = 150.0m,
CompanyName = "Company B", PaymentDayTerms = 75, PoNumber = "193434B" };
            var purchaseOrder3 = new PurchaseOrder { AmountToPay = 12.0m, CompanyName
= "Company C", PaymentDayTerms = 75, PoNumber = "196544A" };
            var purchaseOrder4 = new PurchaseOrder { AmountToPay = 2150.0m,
CompanyName = "Company D", PaymentDayTerms = 75, PoNumber = "234434H" };
            var purchaseOrder5 = new PurchaseOrder { AmountToPay = 2150.0m,
CompanyName = "Company E", PaymentDayTerms = 75, PoNumber = "876434W" };
            var purchaseOrder6 = new PurchaseOrder { AmountToPay = 7150.0m,
CompanyName = "Company F", PaymentDayTerms = 75, PoNumber = "1423474U" };
            var purchaseOrder7 = new PurchaseOrder { AmountToPay = 3150.0m,
CompanyName = "Company G", PaymentDayTerms = 75, PoNumber = "1932344O" };
            var purchaseOrder8 = new PurchaseOrder { AmountToPay = 3190.0m,
CompanyName = "Company H", PaymentDayTerms = 75, PoNumber = "1123457Q" };
            var purchaseOrder9 = new PurchaseOrder { AmountToPay = 50.0m, CompanyName
= "Company I", PaymentDayTerms = 75, PoNumber = "1595344R" };
            var purchaseOrder10 = new PurchaseOrder { AmountToPay = 2150.0m,
CompanyName = "Company J", PaymentDayTerms = 75, PoNumber = "656734L" };

            CreateConnection();

            SendPayment(payment1);
            SendPayment(payment2);
            SendPayment(payment3);
            SendPayment(payment4);
            SendPayment(payment5);
            SendPayment(payment6);
            SendPayment(payment7);
            SendPayment(payment8);
            SendPayment(payment9);
            SendPayment(payment10);

            SendPurchaseOrder(purchaseOrder1);
            SendPurchaseOrder(purchaseOrder2);
            SendPurchaseOrder(purchaseOrder3);
            SendPurchaseOrder(purchaseOrder4);
            SendPurchaseOrder(purchaseOrder5);
            SendPurchaseOrder(purchaseOrder6);
            SendPurchaseOrder(purchaseOrder7);
            SendPurchaseOrder(purchaseOrder8);
            SendPurchaseOrder(purchaseOrder9);
            SendPurchaseOrder(purchaseOrder10);
        }
        private static void CreateConnection()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            _connection = _factory.CreateConnection();
            _model = _connection.CreateModel();
            _model.ExchangeDeclare(ExchangeName, "topic");
        }

        private static void SendPayment(Payment payment)
        {
            SendMessage(payment.Serialize(), "payment.card");
            Console.WriteLine(" Payment Sent {0}, £{1}", payment.CardNumber,
payment.AmountToPay);
        }

        private static void SendPurchaseOrder(PurchaseOrder purchaseOrder)
        {
            SendMessage(purchaseOrder.Serialize(), "payment.purchaseorder");
            Console.WriteLine(" Purchase Order Sent {0}, £{1}, {2}, {3}",
purchaseOrder.CompanyName, purchaseOrder.AmountToPay, purchaseOrder.PaymentDayTerms,
purchaseOrder.PoNumber);
        }

        private static void SendMessage(byte[] message, string routingKey)
        {
            _model.BasicPublish(ExchangeName, routingKey, null, message);
        }
    }
}
```

Next, we have the complete code for the consumer application that processes the “payment.card” messages:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;

        private const string ExchangeName = "Topic_Exchange";

        static void Main()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    Console.WriteLine("Publisher listening for Topic
<payment.card>");
                    Console.WriteLine("--------------------------------------------
");

                    Console.WriteLine();

                    channel.ExchangeDeclare(ExchangeName, "topic");
                    var queueName = channel.QueueDeclare().QueueName;

                    channel.QueueBind(queueName, ExchangeName, "payment.card");

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

                    while (true)
                    {
                        var ea = consumer.Queue.Dequeue();
                        var message = (Payment)ea.Body.DeSerialize();
                        var routingKey = ea.RoutingKey;
                        Console.WriteLine("--- Payment - Routing Key <{0}> : {1} :
{2}", routingKey, message.CardNumber, message.AmountToPay);
                    }
                }
            }
        }
    }
}
```

Next, we have the complete code for the consumer application that processes the “payment.purchaseorder” messages:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;

        private const string ExchangeName = "Topic_Exchange";
        static void Main()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    Console.WriteLine("Publisher listening for Topic
<payment.purchaseorder>");
                    Console.WriteLine("-----------------------------------------------
-------");
                    Console.WriteLine();

                    channel.ExchangeDeclare(ExchangeName, "topic");
                    var queueName = channel.QueueDeclare().QueueName;

                    channel.QueueBind(queueName, ExchangeName,
"payment.purchaseorder");

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

                    while (true)
                    {
                        var ea = consumer.Queue.Dequeue();
                        var message = (PurchaseOrder)ea.Body.DeSerialize();
                        var routingKey = ea.RoutingKey;
                        Console.WriteLine("-- Purchase Order - Routing Key <{0}> :
{1}, £{2}, {3}, {4}", routingKey, message.CompanyName, message.AmountToPay,
message.PaymentDayTerms, message.PoNumber);
                    }
                }
            }
        }
    }
}
```

Finally, we have the complete code for the consumer application that processes the “payment.*” messages:

```
using System;
using RabbitMQ.Client;

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;

        private const string ExchangeName = "Topic_Exchange";

        static void Main()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName = 
"guest", Password = "guest" };
            using (_connection = _factory.CreateConnection())
            {
                using (var channel = _connection.CreateModel())
                {
                    Console.WriteLine("Publisher listening on all payment topics");
                    Console.WriteLine("-----------------------------------------");
                    Console.WriteLine();

                    channel.ExchangeDeclare(ExchangeName, "topic");
                    var queueName = channel.QueueDeclare().QueueName;

                    channel.QueueBind(queueName, ExchangeName, "payment.*");

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

                    while (true)
                    {
                        var ea = consumer.Queue.Dequeue();
                        var reference = ea.Body.DeSerialize();

                        var order = reference as PurchaseOrder;
                        if (order != null)
                        {
                            Console.WriteLine("Purchase Order Recieved from company 
'{0}'", order.CompanyName);
                        }

                        var payment = reference as Payment;
                        if (payment != null)
                        {
                            Console.WriteLine("Card Payment Recieved from person 
'{0}'", payment.Name);
                        }
                    }
                }
            }
        }
    }
}
```

Example 6: Remote Procedure Call

In this final example, we are going to look at something a little different. So far, all of the examples have focused on one-way operations in which a message is placed onto a queue by a producer application and then, at some point in the future, consumers will take the message off of the queue and process them.

In this example, we are going to introduce the concept of a remote procedure call with RabbitMQ. This is where you post a message onto a queue, have a consumer act on the message, and then a reply is posted back via a queue to the originating producer application.

This example is split into two projects: a client and a server. The client application posts payment messages directly onto a queue. For each message that gets posted, the application waits for a reply from a reply queue. This essentially makes this a synchronous process. A message is posted, received by the server application where it is processed, and then a reply is posted back on a reply queue which the client receives. Only when this has all happened does the application post another message:

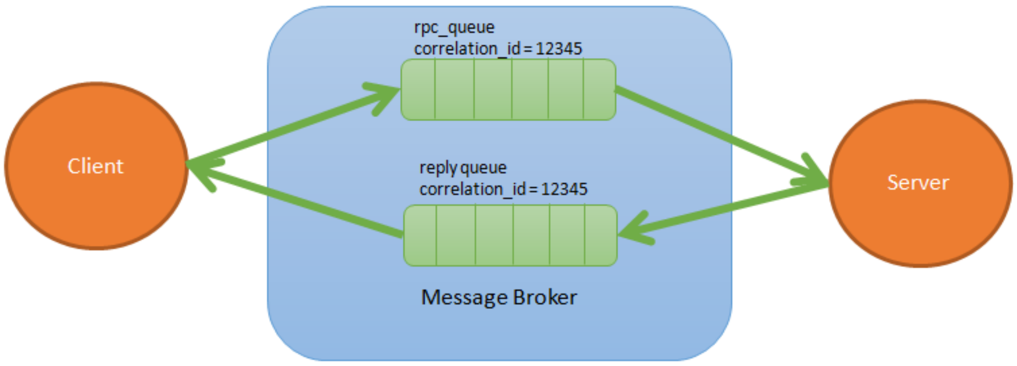


Figure 65: Remote procedure call example

When a message is posted to the server from the client, a ***CorrelationId*** is generated and attached to the message properties. This same ***CorrelationId*** is put into the properties of the reply message. This is useful as it allows you to easily tie together all of the replies with all of the originating messages if you store them off for later retrieval.

This is illustrated in the preceding diagram. The client posts a message to the rpc_queue that has a correlation id of 12345. This message is received by the server and a reply is sent back to the client on the reply_queue with the same correlation id of 12345.

Let's take a look at the code for the client which posts messages onto the queue. First of all, there is code to set up the client connection and queues; we will look at this in a moment. The client makes 10 calls to the ***MakePayment*** method with a new card payment:

```
static void Main()
{
    SetupClient();
    MakePayment(new Payment { AmountToPay = 25.0m, CardNumber = "1234123412341234",
Name = "Mr F Bloggs"});
    MakePayment(new Payment { AmountToPay = 5.0m, CardNumber = "1234123412341234",
Name = "Mr D Wibble" });
    MakePayment(new Payment { AmountToPay = 225.0m, CardNumber = "1234123412341234",
Name = "Mr B Smith" });
    MakePayment(new Payment { AmountToPay = 255.0m, CardNumber = "1234123412341234",
Name = "Mr S Jones" });
    MakePayment(new Payment { AmountToPay = 255.0m, CardNumber = "1234123412341234",
Name = "Mr A Dibbles" });
    MakePayment(new Payment { AmountToPay = 125.0m, CardNumber = "1234123412341234",
Name = "Mr H Howser" });
    MakePayment(new Payment { AmountToPay = 27.0m, CardNumber = "1234123412341234",
Name = "Mr J Jupiter" });
    MakePayment(new Payment { AmountToPay = 925.0m, CardNumber = "1234123412341234",
Name = "Mr Z Zimzibar" });
    MakePayment(new Payment { AmountToPay = 325.0m, CardNumber = "1234123412341234",
Name = "Mr G Goggie" });
    MakePayment(new Payment { AmountToPay = 925.0m, CardNumber = "1234123412341234",
Name = "Mr U Bloggs" });
}
```

The code to set up the code should look familiar: a connection is made with the ***ConnectionFactory*** and then a channel is opened by calling ***CreateModel***. Next, a queue is declared. This queue is the reply queue used for processing replies from the server application. Once this queue has been declared, a ***QueueingBasicConsumer*** is set up for the channel and ***BasicConsume*** is called to start the processing of messages from the reply queue:

```
private static IConnection _connection;
private static IModel _channel;
private static string _replyQueueName;
private static QueueingBasicConsumer _consumer;

private static void SetupClient()
{
    var factory = new ConnectionFactory { HostName = "localhost", UserName = "guest",
Password = "guest" };
    _connection = factory.CreateConnection();
    _channel = _connection.CreateModel();
    _replyQueueName = _channel.QueueDeclare();
    _consumer = new QueueingBasicConsumer(_channel);
    _channel.BasicConsume(_replyQueueName, true, _consumer);
}
```

Now that the connection, channel, and reply queue has been set up, we are ready to start posting our payment messages to the server and handle the replies. The first thing that will happen is, a correlation id will be generated. This id can be any arbitrary string but it is commonly a Globally Unique Identifier (GUID). Once a correlation id has been created, a ***IBasicProperties*** instance is created with the ***CreateBasicProperties*** method. In this basic properties object, the ***ReplyTo*** queue name is inserted along with the ***CorrelationId***.

Next, the message is published to the queue with the ***BasicPublish*** method on the channel. Here you specify the queue name, the properties (containing the reply queue name and the correlation id), and the serialized message:

```
public static string MakePayment(Payment payment)
{
    var corrId = Guid.NewGuid().ToString();
    var props = _channel.CreateBasicProperties();
    props.ReplyTo = _replyQueueName;
    props.CorrelationId = corrId;

    _channel.BasicPublish("", "rpc_queue", props, payment.Serialize());

    while (true)
    {
        Console.WriteLine("----------------------------------------------------------
");
        Console.WriteLine("Payment Made for Card : {0}, for £{1}",
payment.CardNumber, payment.AmountToPay);
        Console.WriteLine("Correlation ID = {0}", corrId);

        var ea = _consumer.Queue.Dequeue();
        if (ea.BasicProperties.CorrelationId != corrId) continue;

        var authCode = Encoding.UTF8.GetString(ea.Body);
        Console.WriteLine("Reply Auth Code : {0}", authCode);
        Console.WriteLine("----------------------------------------------------------
");
        Console.WriteLine("");

        return authCode;
    }
}
```

Once the message has been posted to the queue, the example code goes into a while loop as it waits for a reply from the server. Waiting in an infinite while loop isn’t something that is recommended for a real application, it is fine for this example. In the loop, a message is dequeued from the reply queue. In this example, the reply message is a string representing a payment card authorization number after a payment has been made. This string is converted from a byte array back into a string and then displayed to the console window (along with the correlation id so you can tie the authorization code back to the original payment request).

That covers the code for the client application. Let's now take a look at the server that receives the original message and posts the reply. First of all, the connection is created to RabbitMQ and then the server application goes into a loop where it calls a method to get messages from the queue:

```
private static void Main()
{
    CreateConnection();

    Console.WriteLine("Awaiting Remote Procedure Call Requests");

    while (true)
    {
        GetMessageFromQueue();
    }
}
```

The connection is set up along with the channel as we have seen in all of the other examples. We declare the rpc_queue to ensure that the queue is created. This is the queue on which the messages from the client will be received.

Next, we have a call to ***BasicQos***. The second parameter, which is set to 1, is defining a prefetch count. What this means is, RabbitMQ won't dispatch a new message to a consumer until that consumer has finished processing the message and acknowledged its receipt:

```
private static void CreateConnection()
{
    _factory = new ConnectionFactory { HostName = "localhost", UserName = "guest", 
Password = "guest" };
    _connection = _factory.CreateConnection();
    _channel = _connection.CreateModel();
    _channel.QueueDeclare("rpc_queue", false, false, false, null);
    _channel.BasicQos(0, 1, false);
    _consumer = new QueueingBasicConsumer(_channel);
    _channel.BasicConsume("rpc_queue", false, _consumer);
    _rnd = new Random();
}
```

Once the connection, channel, and queue have been declared, we start to consume from the queue. The message is dequeued from the queue and the message properties are retrieved so that the correlation id can be inserted into the properties of the reply message. This will ensure that the reply can be matched up against the original request:

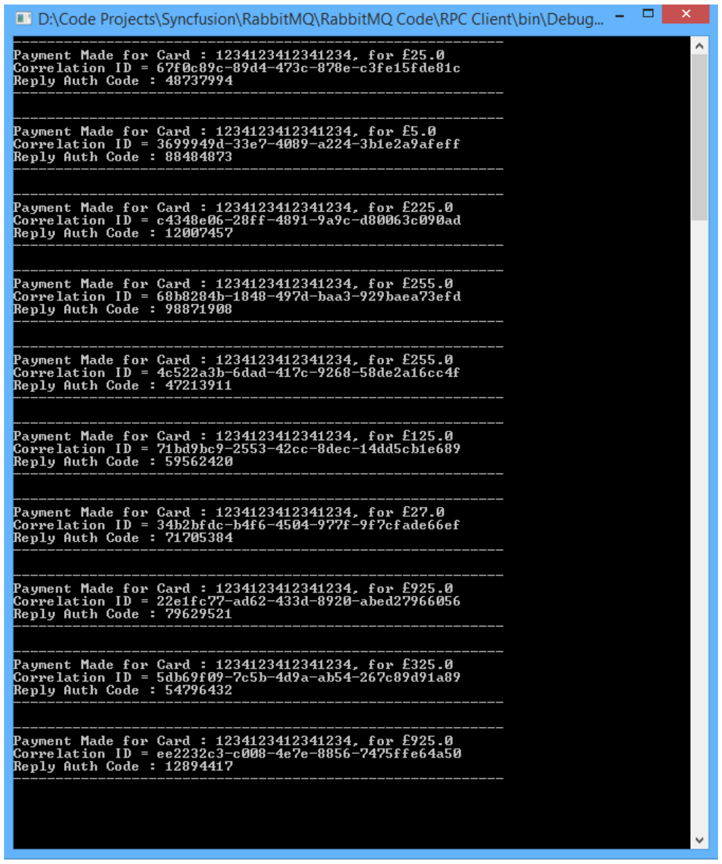
```
private static void GetMessageFromQueue()
{
    string response = null;
    var ea = _consumer.Queue.Dequeue();
    var props = ea.BasicProperties;
    var replyProps = _channel.CreateBasicProperties();
    replyProps.CorrelationId = props.CorrelationId;

    Console.WriteLine("----------------------------------------------------------");

    try
    {
        response = MakePayment(ea);
        Console.WriteLine("Correlation ID = {0}", props.CorrelationId);
    }
    catch (Exception ex)
    {
        Console.WriteLine(" ERROR : " + ex.Message);
        response = "";
    }
    finally
    {
        if (response != null)
        {
            var responseBytes = Encoding.UTF8.GetBytes(response);
            _channel.BasicPublish("", props.ReplyTo, replyProps, responseBytes);
        }
        _channel.BasicAck(ea.DeliveryTag, false);
    }

    Console.WriteLine("----------------------------------------------------------");
    Console.WriteLine("");
}
```

Then, a call is made to ***MakePayment***. In a real application, this would be where you make a payment request with a payment provider but, in this case, we just generate a random number (which serves as a payment authorization code) and output the payment to the console window.

Once this payment has happened, a message is posted to the reply queue with the message properties containing the correlation id. After the message has been sent, the original message is acknowledged so that the next message can be received:

```
private static string MakePayment(BasicDeliverEventArgs ea)
{
    var payment = (Payment) ea.Body.DeSerialize();
    var response = _rnd.Next(1000, 100000000).ToString(CultureInfo.InvariantCulture);
    Console.WriteLine("Payment -  {0} : £{1} : Auth Code <{2}> ", payment.CardNumber,
payment.AmountToPay, response);

    return response;
}
```

To demonstrate this working, run the server application. The application will sit there waiting for requests from the client. Then, run the client application:

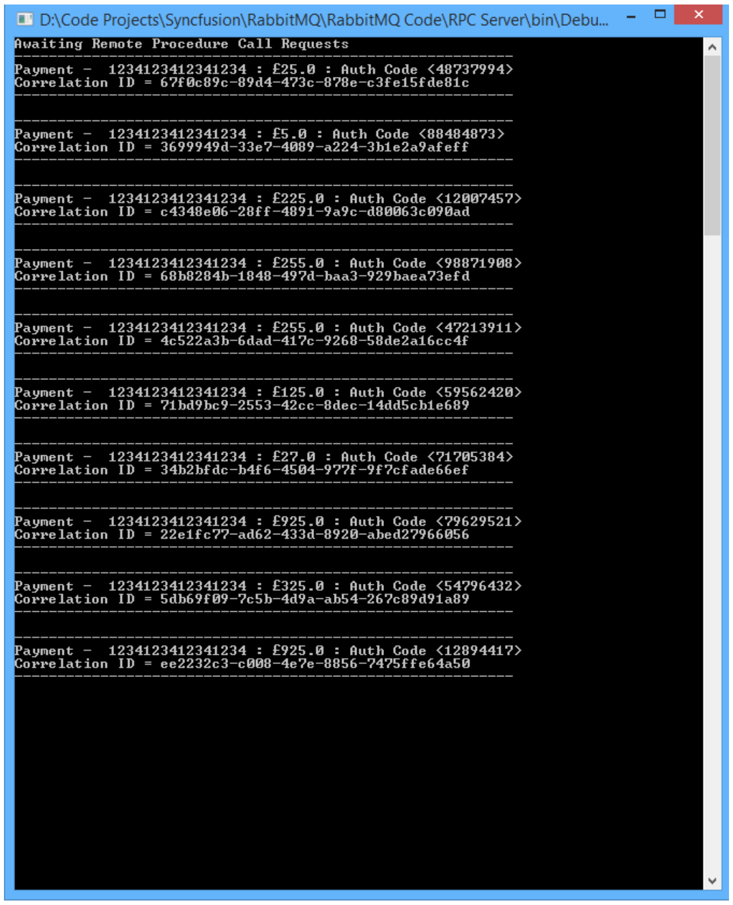


D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\RPC Client\bin\Debug...

```
----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £25.0
Correlation ID = 67f0c89c-89d4-473c-878e-c3fe15fde81c
Reply Auth Code : 48737994
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £5.0
Correlation ID = 3699949d-33e7-4089-a224-3b1e2a9afeff
Reply Auth Code : 88484873
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £225.0
Correlation ID = c4348e06-28ff-4891-9a9c-d80063c090ad
Reply Auth Code : 12007457
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £255.0
Correlation ID = 68b8284b-1848-497d-baa3-929baea73efd
Reply Auth Code : 98871908
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £255.0
Correlation ID = 4c522a3b-6dad-417c-9268-58de2a16cc4f
Reply Auth Code : 47213911
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £125.0
Correlation ID = 71bd9bc9-2553-42cc-8dec-14dd5cb1e689
Reply Auth Code : 59562420
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £27.0
Correlation ID = 34b2bfdc-b4f6-4504-977f-9f7cfade66ef
Reply Auth Code : 71705384
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £925.0
Correlation ID = 22e1fc77-ad62-433d-8920-abed27966056
Reply Auth Code : 79629521
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £325.0
Correlation ID = 5db69f09-7c5b-4d9a-ab54-267c89d91a89
Reply Auth Code : 54796432
----------------------------------------------------------

----------------------------------------------------------
Payment Made for Card : 1234123412341234, for £925.0
Correlation ID = ee2232c3-c008-4e7e-8856-7475ffe64a50
Reply Auth Code : 12894417
----------------------------------------------------------
```

Figure 66: The remote procedure call example client application

You will see the message get sent from the client and then be received and processed by the server. The next message won't get sent until the first message has replied, making this a synchronous operation.


D:\Code Projects\Syncfusion\RabbitMQ\RabbitMQ Code\RPC Server\bin\Debu...

```
Awaiting Remote Procedure Call Requests
----------------------------------------------------------
Payment -  1234123412341234 : £25.0 : Auth Code <48737994>
Correlation ID = 67f0c89c-89d4-473c-878e-c3fe15fde81c
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £5.0 : Auth Code <88484873>
Correlation ID = 3699949d-33e7-4089-a224-3b1e2a9afeff
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £225.0 : Auth Code <12007457>
Correlation ID = c4348e06-28ff-4891-9a9c-d80063c090ad
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £255.0 : Auth Code <98871908>
Correlation ID = 68b8284b-1848-497d-baa3-929baea73efd
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £255.0 : Auth Code <47213911>
Correlation ID = 4c522a3b-6dad-417c-9268-58de2a16cc4f
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £125.0 : Auth Code <59562420>
Correlation ID = 71bd9bc9-2553-42cc-8dec-14dd5cb1e689
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £27.0 : Auth Code <71705384>
Correlation ID = 34b2bfdc-b4f6-4504-977f-9f7cfade66ef
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £925.0 : Auth Code <79629521>
Correlation ID = 22e1fc77-ad62-433d-8920-abed27966056
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £325.0 : Auth Code <54796432>
Correlation ID = 5db69f09-7c5b-4d9a-ab54-267c89d91a89
----------------------------------------------------------

----------------------------------------------------------
Payment -  1234123412341234 : £925.0 : Auth Code <12894417>
Correlation ID = ee2232c3-c008-4e7e-8856-7475ffe64a50
----------------------------------------------------------
```

Figure 67: The remote procedure call example server application

The code below shows the complete code for the remote procedure call client application:

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

namespace RabbitMQ.Examples
{
    class Program
    {
        private static IConnection _connection;
        private static IModel _channel;
        private static string _replyQueueName;
        private static QueueingBasicConsumer _consumer;

        static void Main()
        {
            SetupClient();
            MakePayment(new Payment { AmountToPay = 25.0m, CardNumber =
"1234123412341234", Name = "Mr F Bloggs"});
            MakePayment(new Payment { AmountToPay = 5.0m, CardNumber =
"1234123412341234", Name = "Mr D Wibble" });
            MakePayment(new Payment { AmountToPay = 225.0m, CardNumber =
"1234123412341234", Name = "Mr B Smith" });
            MakePayment(new Payment { AmountToPay = 255.0m, CardNumber =
"1234123412341234", Name = "Mr S Jones" });
            MakePayment(new Payment { AmountToPay = 255.0m, CardNumber =
"1234123412341234", Name = "Mr A Dibbles" });
            MakePayment(new Payment { AmountToPay = 125.0m, CardNumber =
"1234123412341234", Name = "Mr H Howser" });
            MakePayment(new Payment { AmountToPay = 27.0m, CardNumber =
"1234123412341234", Name = "Mr J Jupiter" });
            MakePayment(new Payment { AmountToPay = 925.0m, CardNumber =
"1234123412341234", Name = "Mr Z Zimzibar" });
            MakePayment(new Payment { AmountToPay = 325.0m, CardNumber =
"1234123412341234", Name = "Mr G Goggie" });
            MakePayment(new Payment { AmountToPay = 925.0m, CardNumber =
"1234123412341234", Name = "Mr U Bloggs" });
        }

        private static void SetupClient()
        {
            var factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            _connection = factory.CreateConnection();
            _channel = _connection.CreateModel();
            _replyQueueName = _channel.QueueDeclare();
            _consumer = new QueueingBasicConsumer(_channel);
            _channel.BasicConsume(_replyQueueName, true, _consumer);
        }

        public static string MakePayment(Payment payment)
        {
            var corrId = Guid.NewGuid().ToString();
            var props = _channel.CreateBasicProperties();
            props.ReplyTo = _replyQueueName;
            props.CorrelationId = corrId;

            _channel.BasicPublish("", "rpc_queue", props, payment.Serialize());

            while (true)
            {
                Console.WriteLine("--------------------------------------------------
--------");
                Console.WriteLine("Payment Made for Card : {0}, for £{1}",
payment.CardNumber, payment.AmountToPay);
                Console.WriteLine("Correlation ID = {0}", corrId);

                var ea = _consumer.Queue.Dequeue();
                if (ea.BasicProperties.CorrelationId != corrId) continue;

                var authCode = Encoding.UTF8.GetString(ea.Body);
                Console.WriteLine("Reply Auth Code : {0}", authCode);
                Console.WriteLine("-------------------------------------------------
--------");
                Console.WriteLine("");

                return authCode;
            }
        }
    }
}
```

The code below shows the complete code for the remote procedure call server application:

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

namespace RabbitMQ.Examples
{
    class Program
    {
        private static ConnectionFactory _factory;
        private static IConnection _connection;
        private static IModel _channel;
        private static QueueingBasicConsumer _consumer;
        private static Random _rnd;

        private static void Main()
        {
            CreateConnection();

            Console.WriteLine("Awaiting Remote Procedure Call Requests");

            while (true)
            {
                GetMessageFromQueue();
            }
        }

        private static void GetMessageFromQueue()
        {
            string response = null;
            var ea = _consumer.Queue.Dequeue();
            var props = ea.BasicProperties;
            var replyProps = _channel.CreateBasicProperties();
            replyProps.CorrelationId = props.CorrelationId;

            Console.WriteLine("------------------------------------------------------
----");

            try
            {
                response = MakePayment(ea);
                Console.WriteLine("Correlation ID = {0}", props.CorrelationId);
            }
            catch (Exception ex)
            {
                Console.WriteLine(" ERROR : " + ex.Message);
                response = "";
            }
            finally
            {
                if (response != null)
                {
                    var responseBytes = Encoding.UTF8.GetBytes(response);
                    _channel.BasicPublish("", props.ReplyTo, replyProps,
responseBytes);
                }
                _channel.BasicAck(ea.DeliveryTag, false);
            }

            Console.WriteLine("-----------------------------------------------------
----");
            Console.WriteLine("");
        }

        private static string MakePayment(BasicDeliverEventArgs ea)
        {
            var payment = (Payment) ea.Body.DeSerialize();
            var response = _rnd.Next(1000,
100000000).ToString(CultureInfo.InvariantCulture);
            Console.WriteLine("Payment -  {0} : £{1} : Auth Code <{2}> ",
payment.CardNumber, payment.AmountToPay, response);

            return response;
        }

        private static void CreateConnection()
        {
            _factory = new ConnectionFactory { HostName = "localhost", UserName =
"guest", Password = "guest" };
            _connection = _factory.CreateConnection();
            _channel = _connection.CreateModel();
            _channel.QueueDeclare("rpc_queue", false, false, false, null);
            _channel.BasicQos(0, 1, false);
            _consumer = new QueueingBasicConsumer(_channel);
            _channel.BasicConsume("rpc_queue", false, _consumer);
            _rnd = new Random();
        }

    }
}
```

Closing Notes

Message queuing is an important architectural technique for creating scalable and decoupled solutions. Your choice of message queuing middleware can be an essential decision that your organization has to make. RabbitMQ is a reliable and performant message queuing platform that can absolutely be the backbone of your enterprise’s messaging infrastructure.

The aim of this book was to get you quickly up and running with RabbitMQ so that you can start to reap the benefits of this excellent messaging platform. This book has explained what message queuing is, how it applies to RabbiMQ, how to set up and configure RabbitMQ, and then this book has walked you through a series of examples that will cover the majority of your message queuing needs.