Messaging with RabbitMQ and .NET review part 1: foundations and terminology

AUGUST 2, 2016 1 COMMENT (HTTPS://DOTNETCODR.COM/2016/08/02/MESSAGING-WITH-RABBITMQ-AND-NET-REVIEW-PART-1-FOUNDATIONS-AND-TERMINOLOGY/#COMMENTS)

Introduction

RabbitMQ (https://www.rabbitmq.com/) is a message broker that helps to solve communication between disparate systems in a reliable and maintainable manner. There can be various platforms that need to communicate with each other: a Windows service, a Java servlet based web service, an MVC web application etc. Messaging aims to integrate these systems so that they can exchange information in a decoupled and platform independent fashion.

There have been numerous ways to solve messaging in the past: Java Messaging Service, MSMQ, IBM MQ, but they never really became widespread mostly because they are tied to a specific system, like Windows. Messaging systems based on those technologies were complex, expensive, difficult to connect to and in general difficult to work with. Also, they didn't follow any particular messaging standard; each vendor had their own standards that the customers had to adhere to.

In this new series on RabbitMq we will revisit some concepts and techniques we discussed in the original series here (https://dotnetcodr.com/2014/04/28/messaging-with-rabbitmq-and-net-c-part-1-foundations-and-setup/). As a user commented on the original series, there have been a number of changes, extensions and new concepts in RabbitMq and its .NET client so it's time for a review.

Also, I'm planning to release a couple of blog posts about microservices later this year and RabbitMq is an excellent candidate to solve the messaging portion of the architecture. This series on RabbitMq will therefore serve as a reference material.

Note that much of the original series is still relevant so we'll probably not dive into the same detailed discussion as before. Therefore if you'd like to get to know more about RabbitMq then it's best to view both this and the previous series.

Terminology

RabbitMq is a high availability open-source messaging framework based on the Erlang programming language (https://en.wikipedia.org/wiki/Erlang_(programming_language)). "High availability" means that it's not only very fast when handling messages but it is also scalable. It's a service with a management web UI that you can install on a number of servers. These servers then act in unison as a cluster of message brokers. If one server in the cluster goes down then the others can still handle the incoming messages. Therefore it's similar to a web application which is deployed on 2 or more web servers with a load balancer in front.

RabbitMq by default supports the messaging protocol called AMQP which stands for Advanced Message Queue Protocol (http://www.amqp.org/). The important point here is that AMQP is not some proprietary protocol of a company but rather "an open standard for passing business messages between applications or organizations. It connects systems, feeds business processes with the information they need and reliably transmits onward the instructions that achieve their goals."

Communicating with RabbitMq is quite a straightforward matter using a client library in the programming language of choice. There's of course a .NET library which we'll explore in this series.

In a basic scenario there are two applications that are connected through RabbitMq in a decoupled fashion. Application A sends a message to RabbitMq and application B receives it. Application A can be called the **publisher** or **producer**, or even the **sender**, whereas application B is the **receiver** or **consumer** of the message. This type of communication is often called **producer-consumer** or **publisher-consumer**. Normally application A and B are not coupled in any way in code. Very often application A won't have any knowledge of what kind of receiver applications are waiting for its messages. It is similar to writing a message in Twitter. There may be 0, 1 or more users that read your Twitter update but it's out of your control and you'll probably never know who has read it.

So application A and B are both connected to RabbitMq through a **connection** which is represented by a thread-safe object in the .NET library. This means that the connection object can be reused across multiple threads. A connection contains one ore more channels where each thread in your application should have its own channel. I.e. a channel should not be shared across threads, whereas it's fine to share a single connection.

The default internal workings of RabbitMq reflects the AMQP standard as far as message handling is concerned. Here are some key concepts from the protocol:

- **Message broker**: the messaging server which applications connect to
- **Exchange**: there will be a number of exchanges on the broker which are message routers. A client submits a message to an exchange which will be routed to one or more queues. Therefore the message published by application A will first land in an exchange which doesn't store the message but sends it on to a queue
- **Queue**: a store for messages which normally implements the first-in-first-out pattern, i.e. the message that's been sitting in the queue the longest will be consumed first. This is where application B will pull the messages and **acknowledge** them. The message is then deleted from the queue. A single consumer can monitor one or more queues. An important point here is that the message is held in the queue until the receiver has processed it. Therefore if the consumer is down for whatever reason then the producer can still send the messages to the exchange, a single consumer failure won't break the entire communication system
- **Binding**: a rule that connects the exchange to a queue. The rule also determines which queue the message will be routed to

There are 4 different exchange types:

- **Direct**: a client sends a message to a queue for a particular recipient. The basic version of this exchange type is where a message is routed to the queue without any filter in the binding. The binding can have a filter called the **routing key**. The routing key is a piece of string like "news" or "products". If there are 5 queues bound to an exchange and the message comes with the routing key "europe" then only those queues of the 5 which are set up with the routing key "europe" will get the message. The routing keys must match exactly.
- **Fan-out**: a message is sent to an exchange. The message is then sent to a number of queues which could be bound to that exchange. If a routing key is present then it's ignored.
- **Topic**: a message is sent to a number of queues based on a routing key. The routing key here is somewhat more sophisticated than the one in the direct exchange type. The topic routing key in the queue can include dots, like "consumers.regional.north-america" and the queue can be set up with a routing key with the '*' and '#' placeholders. '*' takes the place of one word like "consumers.regional.*" if a queue is interested in all consumers regardless of the region. '#' covers multiple words like "consumers.#". If there are two queues with these routing keys then both will receive the message with the routing key "consumers.regional.europe". These placeholders are optional, a queue can be set up with the same string "consumers.regional.north-america".
- **Headers** or **header exchange**: the message headers are inspected and the message is routed based on those headers. This is very similar to the topic exchange type but the routing key is included in the header

RabbitMQ is not the only product that implements AMQP: Windows Azure Service Bus, Apache ActiveMQ, StormMQ are some of the other examples. RabbitMQ and the Azure Service Bus are probably enough for most .NET developers. Note that there are different versions of the protocol and some of these products support them to a various degree. At the time of writing this post RabbitMq implemented AMQP version 0.9.

In the next post (https://dotnetcodr.com/2016/08/03/messaging-with-rabbitmq-and-net-review-part-2-installation-and-setup/) we'll install RabbitMq.

View the list of posts on Messaging here (https://dotnetcodr.com/messaging/).

FILED UNDER .NET, MESSAGING TAGGED WITH C#, MESSAGING, RABBITMQ

About Andras Nemes

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

One Response to *Messaging with RabbitMQ and .NET review part 1: foundations and terminology*

Flavio says:

September 29, 2017 at 7:56 pm

I do not directly follow blogs because of my work. However this is an extraordinary blog and I'll follow the topic "messages, and MassTransit…"

PS
Do you think there will be a Camel Apache for .net?

Thank you very much for sharing your knowledge.
Reply