RabbitMQ

Menu

1 "Hello World!"

The simplest thing that does *something*

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

2 Work queues

Distributing tasks among workers (the competing consumers pattern)

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

3 Publish/Subscribe

Sending messages to many consumers at once

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

4 Routing

Receiving messages selectively

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

5 Topics

Receiving messages based on a pattern (topics)

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

6 RPC

Request/reply pattern example

Python Java Ruby PHP C# JavaScript Go Elixir Spring AMQP

7 Publisher Confirms

Reliable publishing with publisher confirms

Java C# PHP

Topics

(using the .NET client)

In the previous tutorial we improved our logging system. Instead of using a `fanout` exchange only capable of dummy broadcasting, we used a `direct` one, and gained a possibility of selectively receiving the logs.

Although using the `direct` exchange improved our system, it still has limitations - it can't do routing based on multiple criteria.

In our logging system we might want to subscribe to not only logs based on severity, but also based on the source which emitted the log. You might know this concept from the `syslog` unix tool, which routes logs based on both severity (info/warn/crit...) and facility (auth/cron/kern...).

That would give us a lot of flexibility - we may want to listen to just critical errors coming from 'cron' but also all logs from 'kern'.

To implement that in our logging system we need to learn about a more complex `topic` exchange.

Prerequisites

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

Where to get help

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

Topic exchange

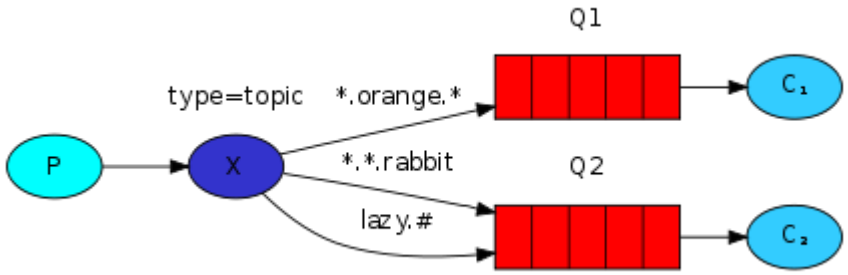
Messages sent to a `topic` exchange can't have an arbitrary `routing_key` - it must be a list of words, delimited by dots. The words can be anything, but usually they specify some features connected to the message. A few valid routing key examples:

" `stock.usd.nyse` ", " `nyse.vmw` ", " `quick.orange.rabbit` ". There can be as many words in the routing key as you like, up to the limit of 255 bytes.

The binding key must also be in the same form. The logic behind the `topic` exchange is similar to a `direct` one - a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys:

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

It's easiest to explain this in an example:



In this example, we're going to send messages which all describe animals. The messages will be sent with a routing key that consists of three words (two dots). The first word in the routing key will describe speed, second a colour and third a species: " `<speed>.<colour>.<species>` ".

We created three bindings: Q1 is bound with binding key " `*.orange.*` " and Q2 with " `*.*.rabbit` " and " `lazy.#` ".

These bindings can be summarised as:

- Q1 is interested in all the orange animals.
- Q2 wants to hear everything about rabbits, and everything about lazy animals.

A message with a routing key set to " `quick.orange.rabbit` " will be delivered to both queues. Message " `lazy.orange.elephant` " also will go to both of them. On the other hand " `quick.orange.fox` " will only go to the first queue, and " `lazy.brown.fox` " only to the second. " `lazy.pink.rabbit` " will be delivered to the second queue only once, even though it matches two bindings. " `quick.brown.fox` " doesn't match any binding so it will be discarded.

What happens if we break our contract and send a message with one or four words, like " `orange` " or " `quick.orange.male.rabbit` "? Well, these messages won't match any bindings and will be lost.

On the other hand " `lazy.orange.male.rabbit` ", even though it has four words, will match the last binding and will be delivered to the second queue.

> **Topic exchange**
>
> Topic exchange is powerful and can behave like other exchanges.
>
> When a queue is bound with " `#` " (hash) binding key - it will receive all the messages, regardless of the routing key - like in `fanout` exchange.
>
> When special characters " `*` " (star) and " `#` " (hash) aren't used in bindings, the topic exchange will behave just like a `direct` one.

Putting it all together

We're going to use a `topic` exchange in our logging system. We'll start off with a working assumption that the routing keys of logs will have two words: " `<facility>.<severity>` ".

The code is almost the same as in the previous tutorial.

The code for `EmitLogTopic.cs` :

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

```
class EmitLogTopic
{
    public static void Main(string[] args)
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: "topic_logs",
                                    type: "topic");

            var routingKey = (args.Length > 0) ? args[0] : "anonymous.info";
            var message = (args.Length > 1)
                          ? string.Join(" ", args.Skip( 1 ).ToArray())
                          : "Hello World!";
            var body = Encoding.UTF8.GetBytes(message);
            channel.BasicPublish(exchange: "topic_logs",
                                 routingKey: routingKey,
                                 basicProperties: null,
                                 body: body);
            Console.WriteLine(" [x] Sent '{0}':'{1}'", routingKey, message);
        }

    }
}
```

The code for `ReceiveLogsTopic.cs`:

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

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

class ReceiveLogsTopic
{
    public static void Main(string[] args)
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: "topic_logs", type: "topic");
            var queueName = channel.QueueDeclare().QueueName;

            if(args.Length < 1)
            {
                Console.Error.WriteLine("Usage: {0} [binding_key...]",
                                         Environment.GetCommandLineArgs()[0]);
                Console.WriteLine(" Press [enter] to exit.");
                Console.ReadLine();
                Environment.ExitCode = 1;
                return;

            }

            foreach(var bindingKey in args)
            {
                channel.QueueBind(queue: queueName,
                                  exchange: "topic_logs",
                                  routingKey: bindingKey);
```

```
            }

            Console.WriteLine(" [*] Waiting for messages. To exit press CTRL+C");

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body.ToArray();
                var message = Encoding.UTF8.GetString(body);
                var routingKey = ea.RoutingKey;
                Console.WriteLine(" [x] Received '{0}':'{1}'",
                                  routingKey,
                                  message);
            };
            channel.BasicConsume(queue: queueName,
                                 autoAck: true,
                                 consumer: consumer);

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


    }
}
```

Run the following examples:

To receive all the logs:

```
cd ReceiveLogsTopic
dotnet run "#"
```

To receive all logs from the facility " `kern` ":

```
cd ReceiveLogsTopic
dotnet run "kern.*"
```

Or if you want to hear only about " `critical` " logs:

```
cd ReceiveLogsTopic
dotnet run "*.critical"
```

You can create multiple bindings:

```
cd ReceiveLogsTopic
dotnet run "kern.*" "*.critical"
```

And to emit a log with a routing key " `kern.critical` " type:

```
cd EmitLogTopic
dotnet run "kern.critical" "A critical kernel error"
```

Have fun playing with these programs. Note that the code doesn't make any assumption about the routing or binding keys, you may want to play with more than two routing key parameters.

(Full source code for EmitLogTopic.cs and ReceiveLogsTopic.cs)

Next, find out how to do a round trip message as a remote procedure call in tutorial 6.

Production [Non-]Suitability Disclaimer

Production (Non-)Suitability Disclaimer

Please keep in mind that this and other tutorials are, well, tutorials. They demonstrate one new concept at a time and may intentionally oversimplify some things and leave out others. For example topics such as connection management, error handling, connection recovery, concurrency and metric collection are largely omitted for the sake of brevity. Such simplified code should not be considered production ready.

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

Getting Help and Providing Feedback

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

Help Us Improve the Docs <3

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