**Publish/Subscribe**

In the [**previous tutorial**](https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html) we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs -- the first will emit log messages and the second will receive and print them.

In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.

Essentially, published log messages are going to be broadcast to all the receivers.

**Exchanges**

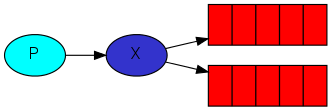
In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.

Let's quickly go over what we covered in the previous tutorials:

* **A *producer* is a user application that sends messages.**
* **A *queue* is a buffer that stores messages.**
* **A *consumer* is a user application that receives messages.**

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an *exchange*. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the *exchange type*.



digraph { bgcolor=transparent; truecolor=true; rankdir=LR; node [style="filled"]; // P1 [label="P", fillcolor="#00ffff"]; X [label="X", fillcolor="#3333CC"]; Q1 [label="{||||}", fillcolor="red", shape="record"]; Q2 [label="{||||}", fillcolor="red", shape="record"]; // P1 -> X; X -> Q1; X -> Q2; }

There are a few exchange types available: **direct, topic, headers and fanout**. We'll focus on the last one -- the **fanout**. Let's create an exchange of this type, and call it logs:

channel.ExchangeDeclare("logs", "fanout");

The fanout exchange is very simple, it broadcasts all the messages it receives to all the queues it knows.

**Listing exchanges**

To list the exchanges on the server you can run the ever useful rabbitmqctl:

$ sudo rabbitmqctl list\_exchanges

Listing exchanges ...

direct

amq.direct direct

amq.fanout fanout

amq.headers headers

amq.match headers

amq.rabbitmq.log topic

amq.rabbitmq.trace topic

amq.topic topic

logs fanout

...done.

In this list there are some amq.\* exchanges and the default (unnamed) exchange. These are created by default, but it is unlikely you'll need to use them at the moment.

|  |  |
| --- | --- |
| **Previously** | **Now** |
| Using exchange: “” uses the default,  Messages are routed to routingKey “hello: | Exchange: “logs”  Messages are not routed to any queue |
| **var** message = GetMessage(args);  **var** body = Encoding.UTF8.GetBytes(message);  channel.BasicPublish(exchange: "",  routingKey: "hello",  basicProperties: **null**,  body: body); | **var** message = GetMessage(args);  **var** body = Encoding.UTF8.GetBytes(message);  channel.BasicPublish(exchange: "logs",  routingKey: "",  basicProperties: **null**,  body: body); |
| Providing a queue name was important for sender and producer, in this case queue “hello” | In this case we don’t assign a specific queu, we want to hear about ALL log messages, not only a subset of them. We are also interested only in currently flowing messages not in the old ones. |

In the case for our logger. We want to hear about all log messages, not just a subset of them. We're also interested only in currently flowing messages not in the old ones. To solve that we need two things.

**Firstly,** whenever we connect to Rabbit we need a fresh, empty queue. To do this we could create a queue with a random name, or, even better - let the server choose a random queue name for us.

**Secondly**, once we disconnect the consumer the queue should be automatically deleted.

In the .NET client, when we supply no parameters to queueDeclare() we create a non-durable, exclusive, autodelete queue with a generated name:

**var** queueName = channel.QueueDeclare().QueueName;

At that point queueName contains a random queue name. For example it may look like amq.gen-JzTY20BRgKO-HjmUJj0wLg.

**Bindings**



digraph { bgcolor=transparent; truecolor=true; rankdir=LR; node [style="filled"]; // P1 [label="P", fillcolor="#00ffff"]; X [label="X", fillcolor="#3333CC"]; Q1 [label="{||||}", fillcolor="red", shape="record"]; Q2 [label="{||||}", fillcolor="red", shape="record"]; // P1 -> X; X -> Q1 [label="binding"]; X -> Q2 [label="binding"]; }

We've already created a fanout exchange and a queue. Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a *binding*.

channel.QueueBind(queue: queueName,

exchange: "logs",

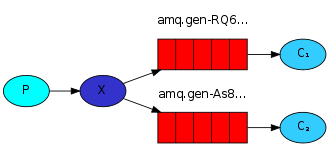
routingKey: "");

From now on the logs exchange will append messages to our queue.

**Listing bindings**

You can list existing bindings using, you guessed it, rabbitmqctl list\_bindings.

**Putting it all together**



digraph { bgcolor=transparent; truecolor=true; rankdir=LR; node [style="filled"]; // P [label="P", fillcolor="#00ffff"]; X [label="X", fillcolor="#3333CC"]; subgraph cluster\_Q1 { label="amq.gen-RQ6..."; color=transparent; Q1 [label="{||||}", fillcolor="red", shape="record"]; }; subgraph cluster\_Q2 { label="amq.gen-As8..."; color=transparent; Q2 [label="{||||}", fillcolor="red", shape="record"]; }; C1 [label=<C<font point-size="7">1</font>>, fillcolor="#33ccff"]; C2 [label=<C<font point-size="7">2</font>>, fillcolor="#33ccff"]; // P -> X; X -> Q1; X -> Q2; Q1 -> C1; Q2 -> C2; }

The producer program, which emits log messages, doesn't look much different from the previous tutorial. The most important change is that we now want to publish messages to our logs exchange instead of the nameless one. We need to supply a routingKey when sending, but its value is ignored for fanout exchanges.

EmitLog.cs

**using** System;

**using** RabbitMQ.Client;

**using** System.Text;

**class** **EmitLog**

{

**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: "logs", type: "fanout");

**var** message = GetMessage(args);

**var** body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "logs",

routingKey: "",

basicProperties: **null**,

body: body);

Console.WriteLine(" [x] Sent {0}", message);

}

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

Console.ReadLine();

}

**private** **static** **string** **GetMessage**(**string**[] args)

{

**return** ((args.Length > 0)

? **string**.Join(" ", args)

: "info: Hello World!");

}

}

After establishing the connection we declared the exchange. This step is necessary as publishing to a non-existing exchange is forbidden.

The messages will be lost if no queue is bound to the exchange yet, but that's okay for us; if no consumer is listening yet we can safely discard the message.

**ReceiveLogs.cs:**

**using** System;

**using** RabbitMQ.Client;

**using** RabbitMQ.Client.Events;

**using** System.Text;

**class** **ReceiveLogs**

{

**public** **static** **void** **Main**()

{

**var** factory = **new** ConnectionFactory() { HostName = "localhost" };

using(**var** connection = factory.CreateConnection())

using(**var** channel = connection.CreateModel())

{

channel.ExchangeDeclare(exchange: "logs", type: "fanout");

**var** queueName = channel.QueueDeclare().QueueName;

channel.QueueBind(queue: queueName,

exchange: "logs",

routingKey: "");

Console.WriteLine(" [\*] Waiting for logs.");

**var** consumer = **new** EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

**var** body = ea.Body;

**var** message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] {0}", message);

};

channel.BasicConsume(queue: queueName,

noAck: **true**,

consumer: consumer);

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

Console.ReadLine();

}

}

}

Compile as before and we're done.

$ csc /r:"RabbitMQ.Client.dll" EmitLogs.cs

$ csc /r:"RabbitMQ.Client.dll" ReceiveLogs.cs

If you want to save logs to a file, just open a console and type:

$ ReceiveLogs.exe > logs\_from\_rabbit.log

If you wish to see the logs on your screen, spawn a new terminal and run:

$ ReceiveLogs.exe

And of course, to emit logs type:

$ EmitLog.exe

Using rabbitmqctl list\_bindings you can verify that the code actually creates bindings and queues as we want. With two ReceiveLogs.cs programs running you should see something like:

$ sudo rabbitmqctl list\_bindings

Listing bindings ...

logs exchange amq.gen-JzTY20BRgKO-HjmUJj0wLg queue **[]**

logs exchange amq.gen-vso0PVvyiRIL2WoV3i48Yg queue **[]**

...done.

Now data from exchange logs goes to two queues with server-assigned names.

To find out how to listen for a subset of messages, let's move on to [**tutorial 4**](https://www.rabbitmq.com/tutorials/tutorial-four-dotnet.html)