RabbitMQ by Pivotal

1 "Hello World!"

The simplest thing that does *something*

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

Spring AMQP

2 Work queues

Distributing tasks among workers (the competing consumers pattern)

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

Spring AMQP

3 Publish/Subscribe

Sending messages to many consumers at once

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

Spring AMQP

4 Routing

Receiving messages selectively

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

Spring AMQP

5 Topics

Introduction

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

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

RabbitMQ, and messaging in general, uses some jargon.

Prerequisites

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

Where to get help

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

Producing means nothing more than sending. A program that sends messages is a *producer* :
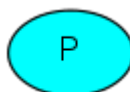
A queue is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a *queue*. A *queue* is only bound by the host's memory & disk limits, it's essentially a large message buffer.

Many *producers* can send messages that go to one queue, and many *consumers* can try to receive data from one *queue*. This is how we represent a queue:

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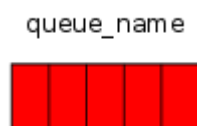
Consuming has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages:

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

"Hello World"

(using the Java Client)

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

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

The Java client library

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

Download the [client library](#) and its dependencies ([SLF4J API](#) and [SLF4J Simple](#)). Copy those files in your working directory, along the tutorials Java files.

Please note SLF4J Simple is enough for tutorials but you should use a full-blown logging library like [Logback](#) in production.

(The RabbitMQ Java client is also in the central Maven repository, with the groupId `com.rabbitmq` and the artifactId `amqp-client`.)

Now we have the Java client and its dependencies, we can write some code.

Sending

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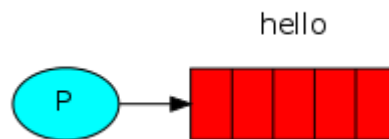
We'll call our message publisher (sender) `Send` and our message consumer (receiver) `Recv`. The publisher will connect to RabbitMQ, send a single message, then exit.

In `Send.java`, we need some classes imported:

```java
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
```

Set up the class and name the queue:

```java
public class Send {
  private final static String QUEUE_NAME = "hello";
  public static void main(String[] argv) throws Exception {
      ...
  }
}
```

then we can create a connection to the server:

```
ConnectionFactory factory = new ConnectionFactory();
```

```java
factory.setHost("localhost");
try (Connection connection = factory.newConnection();
     Channel channel = connection.createChannel()) {

}
```

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

Next we create a channel, which is where most of the API for getting things done resides. Note we can use a try-with-resources statement because both `Connection` and `Channel` implement `java.io.Closeable`. This way we don't need to close them explicitly in our code.

To send, we must declare a queue for us to send to; then we can publish a message to the queue, all of this in the try-with-resources statement:

```java
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
String message = "Hello World!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
System.out.println(" [x] Sent '" + message + "'");
```

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

Here's the whole Send.java class.

> **Sending doesn't work!**
>
> If this is your first time using RabbitMQ and you don't see the "Sent" message then you may be left scratching your head wondering what could be wrong. Maybe the broker was started without enough free disk space (by default it needs at least 200 MB free) and is therefore refusing to accept messages. Check the broker logfile to confirm and reduce the limit if necessary. The configuration file documentation will show you how to set `disk_free_limit`.

Receiving

That's it for our publisher. Our consumer is pushed messages from RabbitMQ, so unlike the publisher which publishes a single message, we'll keep it running to listen for messages and print them out.
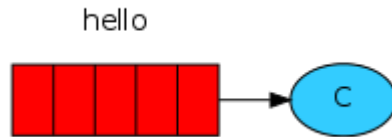
The code (in `Recv.java`) has almost the same imports as `Send`:

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
```

The extra `DefaultConsumer` is a class implementing the `Consumer` interface we'll use to buffer the messages pushed to us by the server.

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

```java
public class Recv {

  private final static String QUEUE_NAME = "hello";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

  }
}
```

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

Why don't we use a try-with-resource statement to automatically close the channel and the connection? By doing so we would simply make the program move on, close everything, and exit! This would be awkward because we want the process to stay alive while the consumer is listening asynchronously for messages to arrive.

We're about to tell the server to deliver us the messages from the queue. Since it will push us messages asynchronously, we provide a callback in the form of an object that will buffer the messages until we're ready to use them. That is what a `DeliverCallback` subclass does.

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received '" + message + "'");
};
channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
```

[Here's the whole Recv.java class](#).

Putting it all together

You can compile both of these with just the RabbitMQ java client on the classpath:

```
javac -cp amqp-client-5.5.1.jar Send.java Recv.java
```

To run them, you'll need `rabbitmq-client.jar` and its dependencies on the classpath. In a terminal, run the consumer (receiver):

```
java -cp .:amqp-client-5.5.1.jar:slf4j-api-1.7.25.jar:slf4j-simple-1.7.25.jar Recv
```

then, run the publisher (sender):

```
java -cp .:amqp-client-5.5.1.jar:slf4j-api-1.7.25.jar:slf4j-simple-1.7.25.jar Send
```

On Windows, use a semicolon instead of a colon to separate items in the classpath.

The consumer will print the message it gets from the publisher via RabbitMQ. The

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

Listing queues

You may wish to see what queues RabbitMQ has and how many messages are in them. You can do it (as a privileged user) using the `rabbitmqctl` tool:

```
sudo rabbitmqctl list_queues
```

On Windows, omit the sudo:

```
rabbitmqctl.bat list_queues
```

Time to move on to [part 2](#) and build a simple *work queue*.

Hint

To save typing, you can set an environment variable for the classpath e.g.

```
export CP=.:amqp-client-5.5.1.jar:slf4j-api-1.7.25.jar:slf4j-simple-1.7.25.jar
java -cp $CP Send
```

or on Windows:

```
set CP=.;amqp-client-5.5.1.jar;slf4j-api-1.7.25.jar;slf4j-simple-1.7.25.jar
java -cp %CP% Send
```

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!