The Wayback Machine - https://web.archive.org/web/20160612170616/http://www.rabbitmq.com:80/tutorials/tutorial-one-python.html

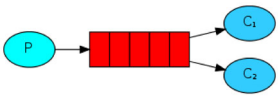
1 "Hello World!"

The simplest thing that does *something*

Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

2 Work queues

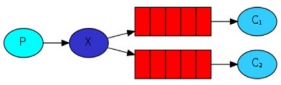
Distributing tasks among workers



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

3 Publish/Subscribe

Sending messages to many consumers at once



Python
Java
Ruby
PHP
C#
Javascript

Introduction

RabbitMQ is a message broker. The principal idea is pretty simple: it accepts and forwards messages. You can think about it as a post office: when you send mail to the post box you're pretty sure that Mr. Postman will eventually deliver the mail to your recipient. Using this metaphor RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is the fact 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.

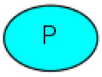
Prerequisites

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

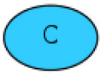
If you're having trouble going through this tutorial you can **contact us** through the mailing list.

Producing means nothing more than sending. A program that sends messages is a *producer*. We'll draw it like that, with "P":



A queue is the name for a mailbox. It lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can be stored only inside a *queue*. A *queue* is not bound by any limits, it can store as many messages as you like – it's essentially an infinite buffer. Many *producers* can send messages that go to one queue, many *consumers* can try to receive data from one *queue*. A queue will be drawn as like that, with its name above it:

Consuming has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages. On our drawings it's shown with "C":



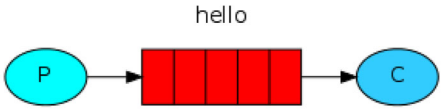
Note that the producer, consumer, and broker do not have to reside on the same machine; indeed in most applications they don't.

Hello World!

(using the pika 0.10.0 Python client)

Our "Hello world" won't be too complex – let's send a message, receive it and print it on the screen. To do so we need two programs: one that sends a message and one that receives and prints it.

Our overall design will look like:



IPR2024-01311
Genius Sports v. Sportscastr
Genius Sports Ex. 1065

Go
Elixir
Objective-C

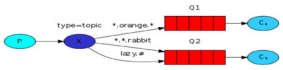
4 Routing

Receiving messages selectively

Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

5 Topics

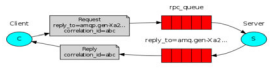
Receiving messages based on a pattern



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

6 RPC

Remote procedure call implementation



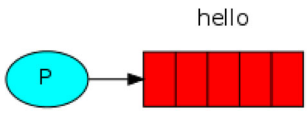
Python
Java
Ruby
PHP
C#
Javascript
Go

Producer sends messages to the "hello" queue. The consumer receives messages from that queue.

RabbitMQ libraries

RabbitMQ speaks 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**. In this tutorial series we're going to use **Pika**, which is the Python client recommended by the RabbitMQ team. To install it you can use the `pip` package management tool.

Sending



Our first program `send.py` will send a single message to the queue. The first thing we need to do is to establish a connection with RabbitMQ server.

```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
               'localhost'))
channel = connection.channel()
```

We're connected now, 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, before sending we need to make sure the recipient queue exists. If we send a message to non-existing location, RabbitMQ will just trash the message. Let's create a queue to which the message will be delivered, let's name it *hello*:

```
channel.queue_declare(queue='hello')
```

At that point we're ready to send a message. Our first message will just contain a string *Hello World!* and we want to send it to our *hello* queue.

In RabbitMQ a message can never be sent directly to the queue, it always needs to go through an *exchange*. But let's not get dragged down by the details – you can read more about *exchanges* in **the third part of this tutorial**. All we need to know now is how to use a default exchange identified by an empty string. This exchange is special – it allows us to specify exactly to which queue the message should go. The queue name needs to be specified in the `routing_key` parameter:

```
channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
```

Before exiting the program we need to make sure the network buffers were flushed and our message was actually delivered to RabbitMQ. We can do it by gently closing the connection.

```
connection.close()
```

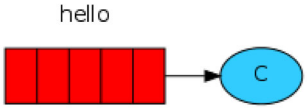
Sending doesn't work!

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 1Gb free) and is therefore refusing to accept messages. Check the broker logfile to confirm and reduce the limit if

Elixir

necessary. The **configuration file documentation** will show you how to set `disk_free_limit`.

Receiving



Our second program `receive.py` will receive messages from the queue and print them on the screen.

Again, first we need to connect to RabbitMQ server. The code responsible for connecting to Rabbit is the same as previously.

The next step, just like before, is to make sure that the queue exists. Creating a queue using `queue_declare` is idempotent – we can run the command as many times as we like, and only one will be created.

```
channel.queue_declare(queue='hello')
```

You may ask why we declare the queue again – we have already declared it in our previous code. We could avoid that if we were sure that the queue already exists. For example if `send.py` program was run before. But we're not yet sure which program to run first. In such cases it's a good practice to repeat declaring the queue in both programs.

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
Listing queues ...
hello    0
...done.
```

(omit sudo on Windows)

Receiving messages from the queue is more complex. It works by subscribing a `callback` function to a queue. Whenever we receive a message, this `callback` function is called by the Pika library. In our case this function will print on the screen the contents of the message.

```
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
```

Next, we need to tell RabbitMQ that this particular callback function should receive messages from our *hello* queue:

```
channel.basic_consume(callback,
                      queue='hello',
                      no_ack=True)
```

For that command to succeed we must be sure that a queue which we want to subscribe to exists. Fortunately we're confident about that – we've created a queue above – using `queue_declare`.

The `no_ack` parameter will be described **later on**.

And finally, we enter a never-ending loop that waits for data and runs callbacks whenever necessary.

```
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

Putting it all together

Full code for `send.py`:

```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
        host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

(send.py source)

Full `receive.py` code:

```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
        host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

channel.basic_consume(callback,
                      queue='hello',
                      no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

(receive.py source)

Now we can try out our programs in a terminal. First, let's send a message using our `send.py` program:

```
$ python send.py
[x] Sent 'Hello World!'
```

The producer program `send.py` will stop after every run. Let's receive it:

```
$ python receive.py
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Hello World!'
```

Hurray! We were able to send our first message through RabbitMQ. As you might have noticed, the `receive.py` program doesn't exit. It will stay ready to receive further messages, and may be interrupted with Ctrl-C.

Try to run `send.py` again in a new terminal.

We've learned how to send and receive a message from a named queue. It's time to move on to **part 2** and build a simple *work queue*.

Copyright © 2007-2016 Pivotal Software, Inc. All rights reserved | Privacy Policy | This Site is Open Source | Pivotal is Hiring Sitemap | Contact