Advanced Message Queue Protocol

भारतीय प्रौद्योगिकी संस्थान (का.हि.वि.) वाराणसी

योगः कर्मसु कौशलम्

INDIAN INSTITUTE OF TECHNOLOGY (B.H.U.) VARANASI

संस्कार ही शिक्षा

EDUCATION IS CHARACTER

By : Sidhant Sarraf(17074015)

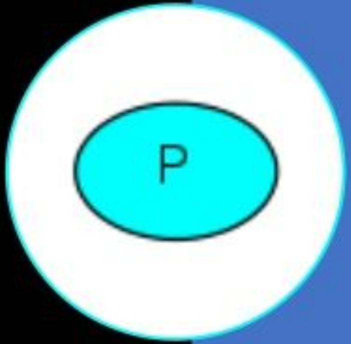
Under Guidance of Dr. Hari Prabhat Gupta

What is RabbitMq ?

- RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, Message Queuing Telemetry Transport, and other protocols

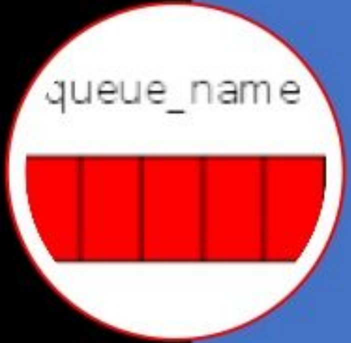
Understanding some Terminology

- *Producing* means nothing more than sending. A program that sends messages is a *producer* :

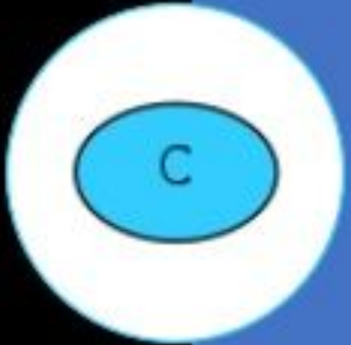


Queue

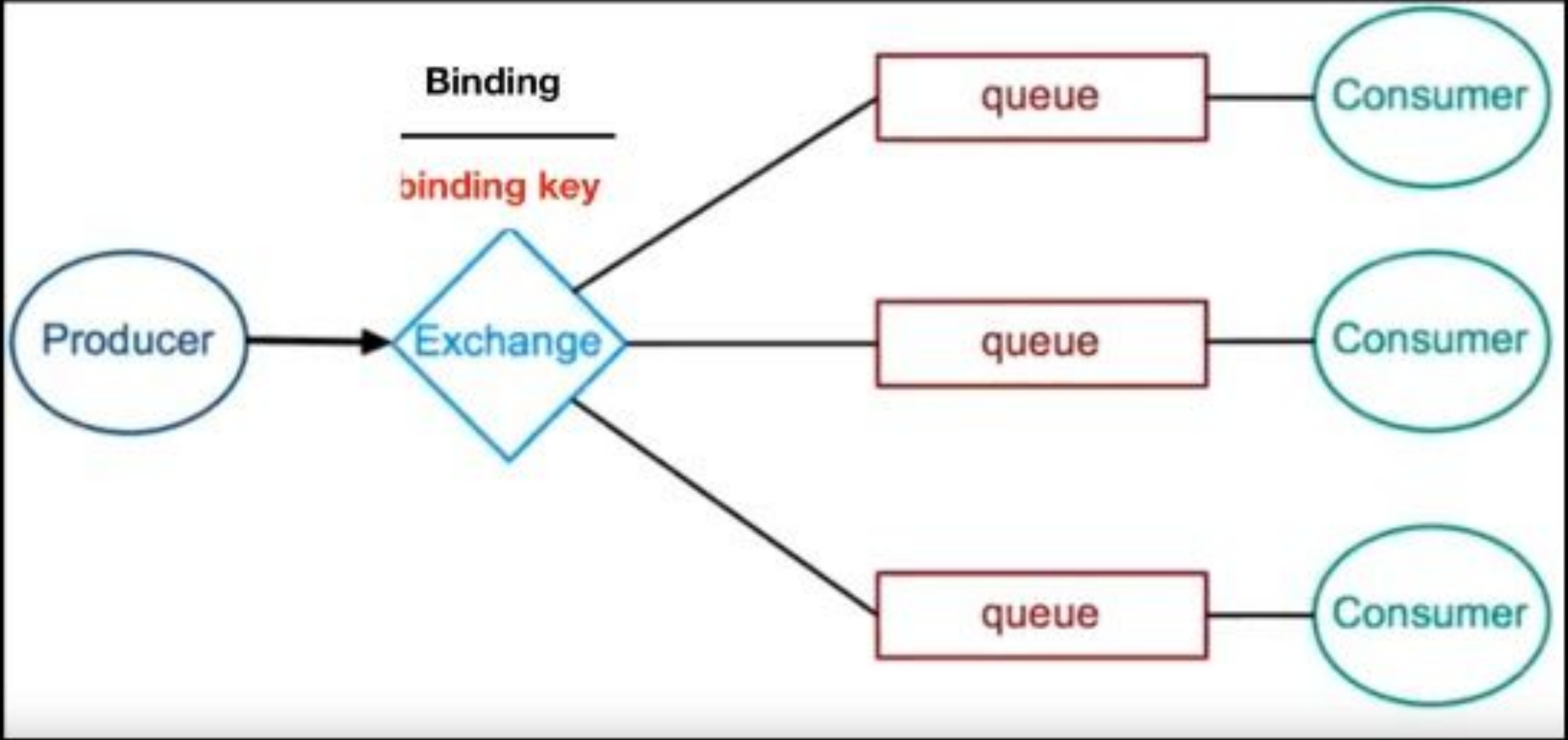
- *A queue* is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a *queue*. A *queue* is only bound by the host's memory & disk limits, it's essentially a large message buffer. Many *producers* can send messages that go to one queue, and many *consumers*can try to receive data from one *queue*. This is how we represent a queue:

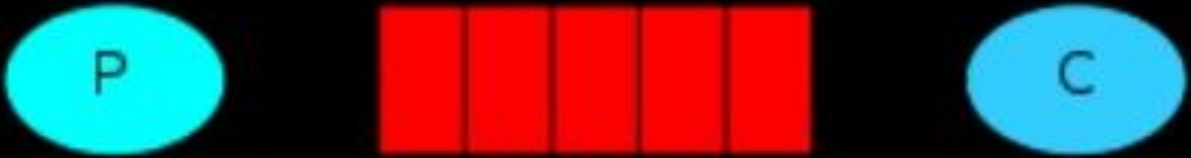


- *Consuming* has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages



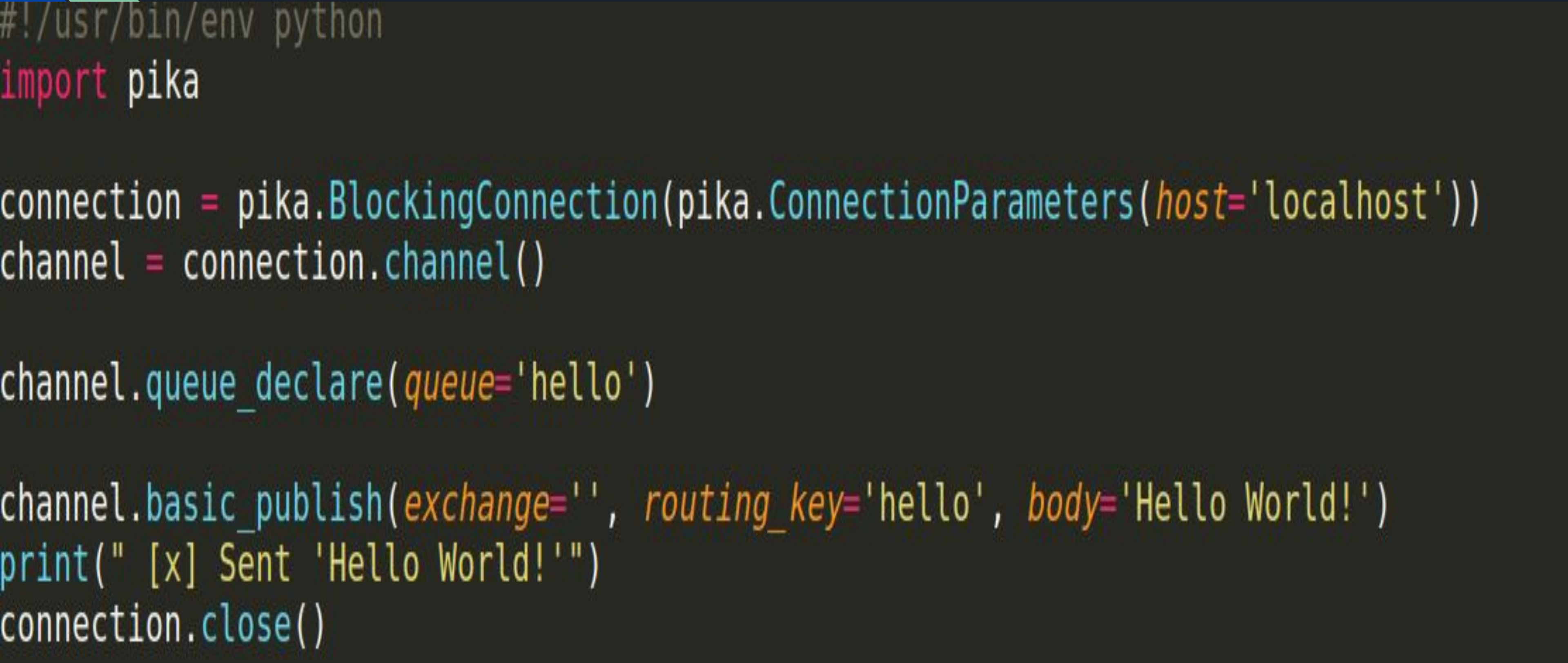
Example 1 : getting started with Rabbit Mq UI





First program

Send.py



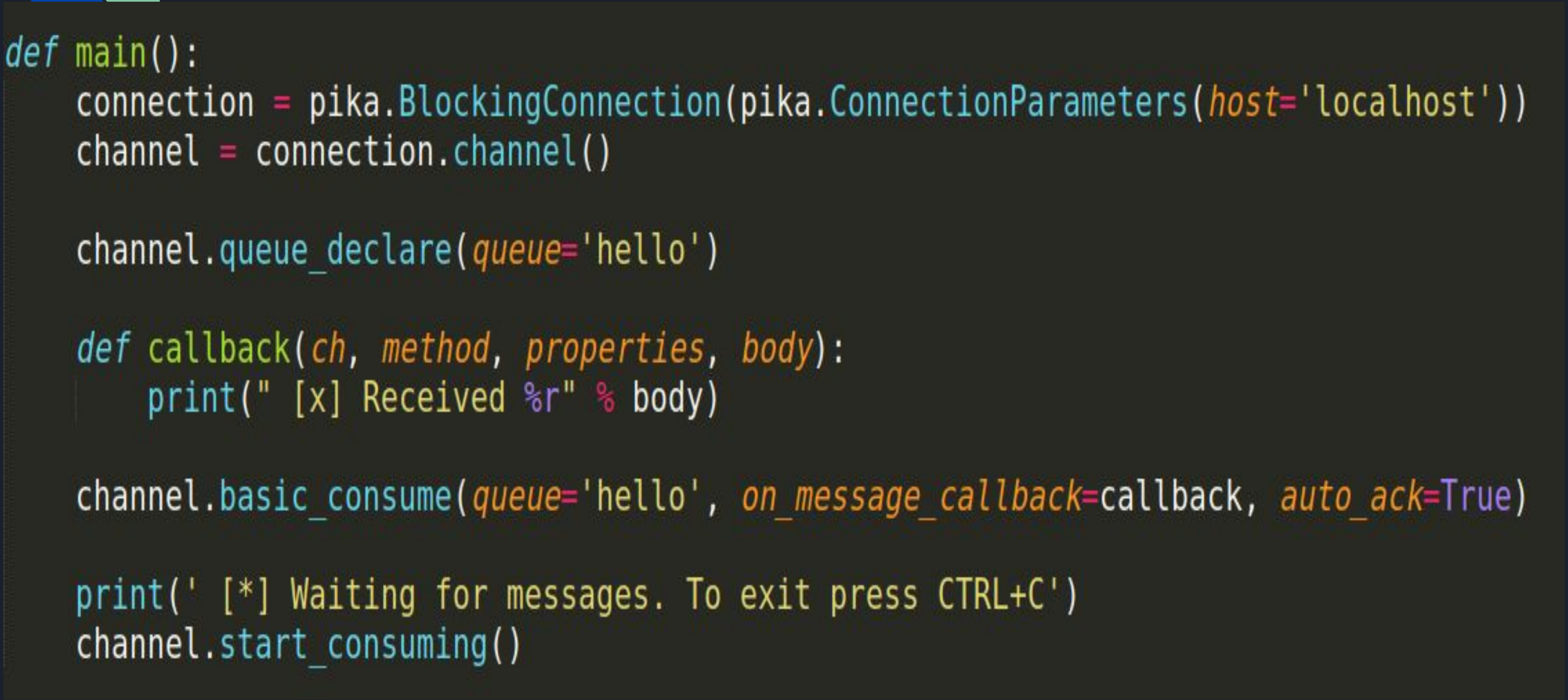
```
#!/usr/bin/env python
import pika

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()
```

Receive.py



```python
def main():
    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(queue='hello', on_message_callback=callback, auto_ack=True)

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

Message acknowledgment

In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back by the consumer to tell RabbitMQ that a particular message had been received, processed and that RabbitMQ is free to delete it.

If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

```
ch.basic_ack(delivery_tag = method.delivery_tag)
```

Message durability

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

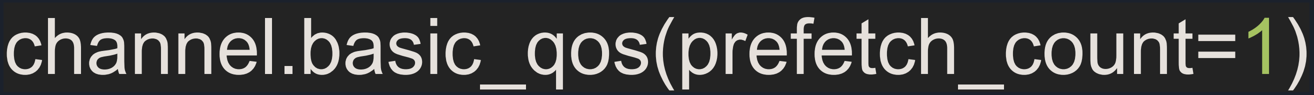
```
channel.queue_declare(queue='hello', durable=True)
```

Now we need to mark our messages as persistent - by supplying a `delivery_mode` property with a value `2`.

Fair dispatch

So, we have shown you that a message is send to different worker in round robin fashion so it may happen that a worker would be busy with his task and some other worker would be free but due to its round robin fashion it may not distribute task uniformly that is worker who is free should be assigned with some task.

In order to defeat that we can use the `Channel#basic_qos` channel method with the `prefetch_count=1` setting. This uses the `basic.qos` protocol method to tell RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.



```
channel.basic_qos(prefetch_count=1)
```

Exchanges

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*.

There are a few exchange types available: `direct`, `topic`, `headers` and `fanout`.

Fanout Exchange

The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows.

```
channel.exchange_declare(exchange='logs',  exchange_type='fanout')
```

Temporary queues

As you may remember previously we were using queues that had specific names (remember `hello` and `task_queue`?).

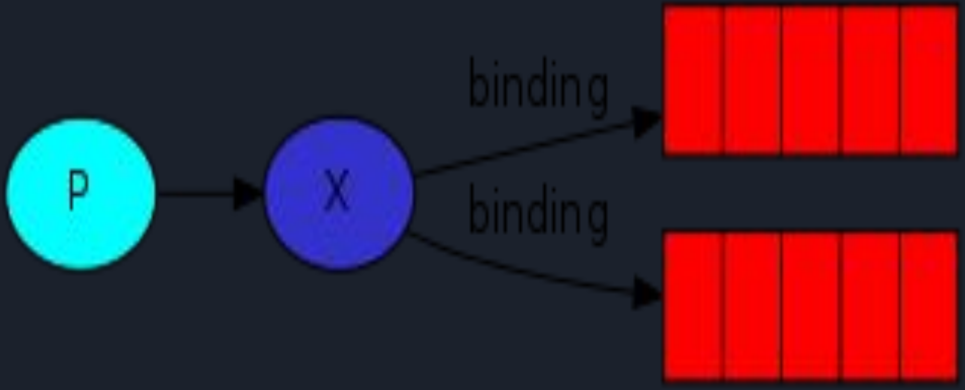
whenever we connect to Rabbit we need a fresh, empty queue. To do it we could create a queue with a random name, or, even better - let the server choose a random queue name for us. We can do this by supplying empty `queue` parameter to `queue_declare`:

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

Secondly, once the consumer connection is closed, the queue should be deleted. There's an `exclusive` flag for that:

```
result = channel.queue_declare(queue='', exclusive=True)
```

Bindings



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.queue_bind(exchange='logs',queue=result.method.queue)
```

Bindings

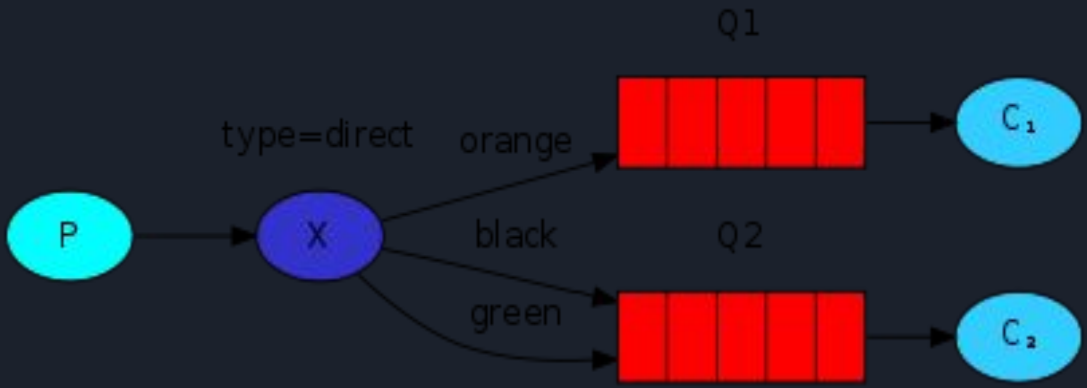
Bindings can take an extra `routing_key` parameter. To avoid the confusion with a `basic_publish` parameter we're going to call it a `binding key`. This is how we could create a binding with a key:

```
channel.queue_bind(exchange=exchange_name, queue=queue_name, routing_key='black')
```

The meaning of a binding key depends on the exchange type. The `fanout` exchanges, which we used previously, simply ignored its value.

Direct exchange

The routing algorithm behind a `direct` exchange is simple - a message goes to the queues whose `binding key` exactly matches the `routing key` of the message.



Topic exchange

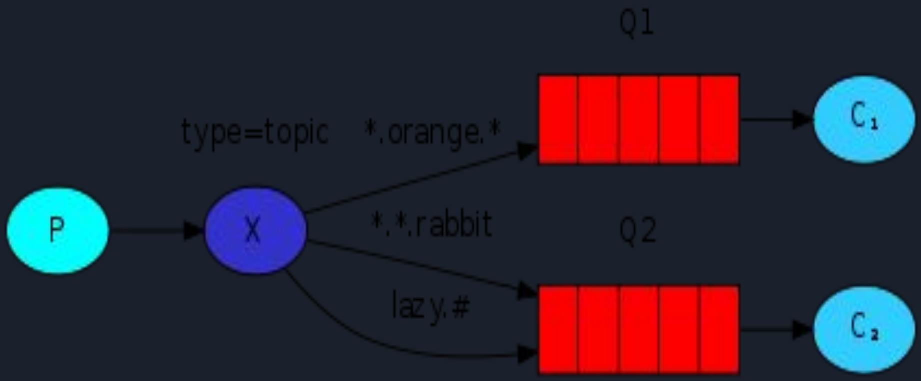
Messages sent to a `topic` exchange can't have an arbitrary `routing_key` - it must be a list of words, delimited by dots.

The 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.

Topic exchange

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.