**Types of Queues:**

1. Simple Queue (Default Exchange- aka ‘nameless exchange’)
2. Work Queues (Default Exchange- aka ‘nameless exchange’)
3. Publish/Subscribe (Fanout Exchange)
4. Routing (Direct exchange)
5. Topics (Topic exchange) : If we want to subscribe to not only queues based on the routing key, but also based on the source which produced the message/event.
6. RPC (Remote Procedure Call) : If we need to run a function on a remote computer and wait for the result? This pattern is commonly known as Remote Procedure Call or RPC.

## Temporary queues

Giving a queue a name is important when you want to share the queue between producers and consumers. But that's not the case everytime.

If we want to receive about all 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. To do this with the spring-amqp client, we defined and AnonymousQueue, which creates a non-durable, exclusive, autodelete queue with a generated name.

Bindings



Once we've created an exchange and a queue, we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a *binding*.

### **Message acknowledgment**

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. Spring-amqp by default takes a conservative approach to message acknowledgement. If the listener throws an exception the container calls:

channel.basicReject(deliveryTag, requeue)

Requeue is true by default unless you explicitly set:

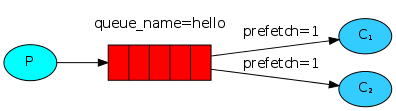
defaultRequeueRejected=false

or the listener throws an AmqpRejectAndDontRequeueException. This is typically the bahavior you want from your listener. In this mode there is no need to worry about a forgotten acknowledgement. After processing the message the listener calls:

channel.basicAck()

### **Fair dispatch vs Round-robin dispatching**

However, "Fair dispatch" is the default configuration for spring-amqp. The SimpleMessageListenerContainer defines the value for DEFAULT\_PREFETCH\_COUNT to be 1. If the DEFAULT\_PREFECTH\_COUNT were set to 0 the behavior would be round robin messaging as described above.



However, with the prefetchCount set to 1 by default, this tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatcha 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.

### **Message durability**

With spring-amqp there are reasonable default values in the MessageProperties that account for message durability. In particular you can check the table for [common properties](http://docs.spring.io/spring-amqp/reference/htmlsingle/#_common_properties) You'll see two relevant to our discussion here on durability:

| Property | default | Description |
| --- | --- | --- |
| durable | true | When declareExchange is true the durable flag is set to this value |
| deliveryMode | PERSISTENT | PERSISTENT or NON\_PERSISTENT to determine whether or not RabbitMQ should persist the messages |

#### Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do fsync(2) for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use [publisher confirms](https://www.rabbitmq.com/confirms.html).

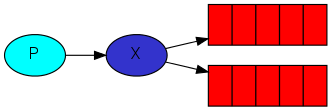
## Exchanges

Full messaging model in Rabbit:

* 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*.

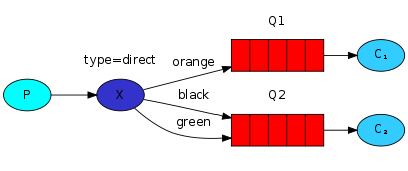


There are a few exchange types available: direct, topic, headers and fanout. We'll focus on the last one -- the fanout.

## 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.

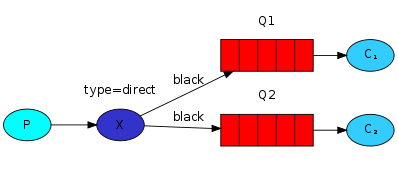
To illustrate that, consider the following setup:



In this setup, we can see the direct exchange X with two queues bound to it. The first queue is bound with binding key orange, and the second has two bindings, one with binding key black and the other one with green.

In such a setup a message published to the exchange with a routing key orange will be routed to queue Q1. Messages with a routing key of black or green will go to Q2. All other messages will be discarded.

Multiple bindings



It is perfectly legal to bind multiple queues with the same binding key. In that case, the direct exchange will behave like fanout and will broadcast the message to all the matching queues.

**Limitations** - **it can't do routing based on multiple criteria.**

## Publishing messages/events

## We'll send messages/events to a direct exchange. We will supply a routing key. That way the receiving program will be able to select the messages/events it wants to receive (or subscribe to).

## Subscribing

## We need to create a new binding for each type of message/event we're interested in.

## 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.

## Remote Procedure Call or RPC

### Callback queue

## In general doing RPC over RabbitMQ is easy. A client sends a request message and a server replies with a response message. In order to receive a response we need to send a 'callback' queue address with the request. Spring-amqp's RabbitTemplate handles the callback queue for us when we use the above 'convertSendAndReceive()' method. There is no need to do any other setup when using the RabbitTemplate. For a thorough explanation please see [Request/Reply Message](http://docs.spring.io/spring-amqp/reference/htmlsingle/#request-reply).

### Correlation Id

Typically the native client would create a callback queue for every RPC request. That's pretty inefficient so an alternative is to create a single callback queue per client.

That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the correlationId property is used. Spring-amqp automatically sets a unique value for every request. In addition it handles the details of matching the response with the correct correlationID.