

Getting started with

RabbitMQ

and CloudAMQP



L. Johansson & J. Rhodin

Getting Started with RabbitMQ and CloudAMQP

Lovisa Johansson

Book version: 2.0

Published: 2017-05-09

Copyright © 84 Codes AB

Table of Contents

[Introduction](#)

[What is RabbitMQ?](#)

[The Management Interface](#)

[Exchanges, routing keys and bindings](#)

[RabbitMQ and Ruby](#)

[RabbitMQ and Node.js](#)

[RabbitMQ and Python](#)

[A microservice architecture built upon RabbitMQ](#)

[RabbitMQ Troubleshooting](#)

[More Resources](#)

Introduction

This book gives a gentle introduction to RabbitMQ and how to use it on the CloudAMQP online platform. This book originated as a series of blogposts on the CloudAMQP website. The continued success of these posts led us to the idea to summarize the content in book form, to make it easy to digest for different readers.

This book is an evolving project and new versions will be released.

Thank you to all CloudAMQP users for your feedback and continued support. It has been a great motivator to see customer projects succeed.

What is RabbitMQ?

The first chapter of **Getting Started with RabbitMQ and CloudAMQP** explains what RabbitMQ and message queueing is - a way of exchanging data between processes, applications, and servers.



The chapter will give a brief understanding of messaging and important concepts will be defined. The chapter will also explain the steps to go through when setting up a connection and publishing/consuming messages to/from a queue.

RabbitMQ is a message-queueing software called a *message broker* or *queue manager*. Simply said; It is a software where queues can be defined, applications may connect to the queue and transfer a message onto it.

A message can include any kind of information. It could, for example, have information about a process/task that should start on another application (that could be on another server), or it could be just a simple text message. The queue-manager software stores the messages until a receiving application connects and takes a message off the queue. The receiving application then processes the message in an appropriate manner.



Figure: A typical message queue where messages are being sent from a sender to a receiver.

RabbitMQ Example

A message brooking solution can act like a middleman for various services (e.g., a web application, as in this example). They can be used to reduce loads and delivery times by web application servers since tasks, which would normally take quite a bit of time to process, can be delegated to a third party whose only job is to perform them.

In this chapter, we follow a scenario where a web application allows users to upload information to a website. The site will handle this information and generate a PDF and email it back to the user. Handling the information, generating the PDF and sending the email will in this example case take several seconds and that is one of the reasons of why a message queue will be used.

When the user has entered user information into the web interface, the web application will put a "PDF processing" - task and all information into a message and the message will be placed onto a queue defined in RabbitMQ.

The basic architecture of a message queue is simple: there are client applications called producers that create messages and deliver them to the broker (the message queue). Other applications, called consumers, connect to the queue and subscribe to the messages to be processed. A software

can be a producer, or consumer, or both a consumer and a producer of messages. Messages that are placed onto the queue are stored until the consumer retrieves them.



Figure: A sketch of a typical RabbitMQ workflow

When and why should you use RabbitMQ?

Message queueing allow web servers to respond to requests quickly, instead of being forced to perform resource-heavy procedures on the spot. Message queueing is also good when you want to distribute a message to multiple recipients for consumption or for balancing loads between workers.

The consumer can take a message of the queue and start the processing of the PDF at the same time as the producer is queueing up new messages on the queue. The consumer can be on a totally different server than the publisher, or they can be located on the same server. Requests can be created in one programming language and handled in another programming language - the two applications will only communicate through the messages they are sending to each other. Due to that, the two applications will have a low coupling between the sender and the receiver.

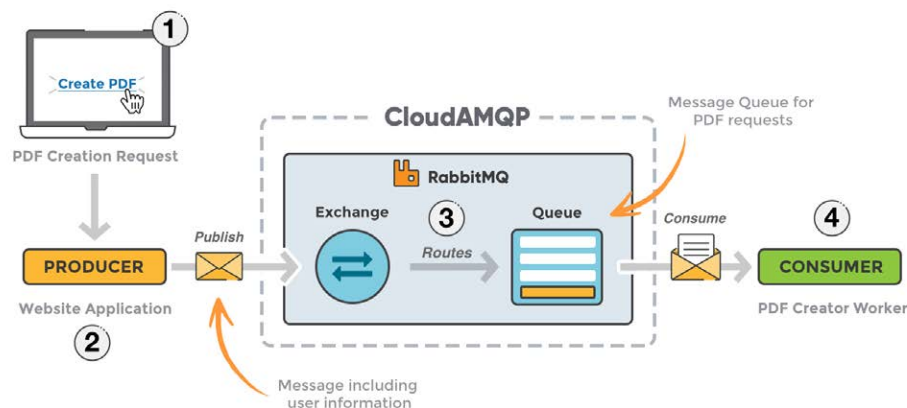


Figure: Application architecture example with RabbitMQ.

- The user sends a PDF creation request to the web application.
- The web application (the producer) sends a message to RabbitMQ, including data from the request, like name and email.
- An exchange accepts the messages from a producer application and routes them to correct message queues for PDF creation.
- The PDF processing worker (the consumer) receives the task and starts the processing of the PDF.

Exchanges

Messages are not published directly to a queue, instead, the producer sends messages to an exchange. An exchange is responsible for the routing of the messages to the different queues. An exchange accepts messages from the producer application and routes them to message queues with the help of bindings and routing keys. A binding is the link between a queue and an exchange.

Message flow in RabbitMQ

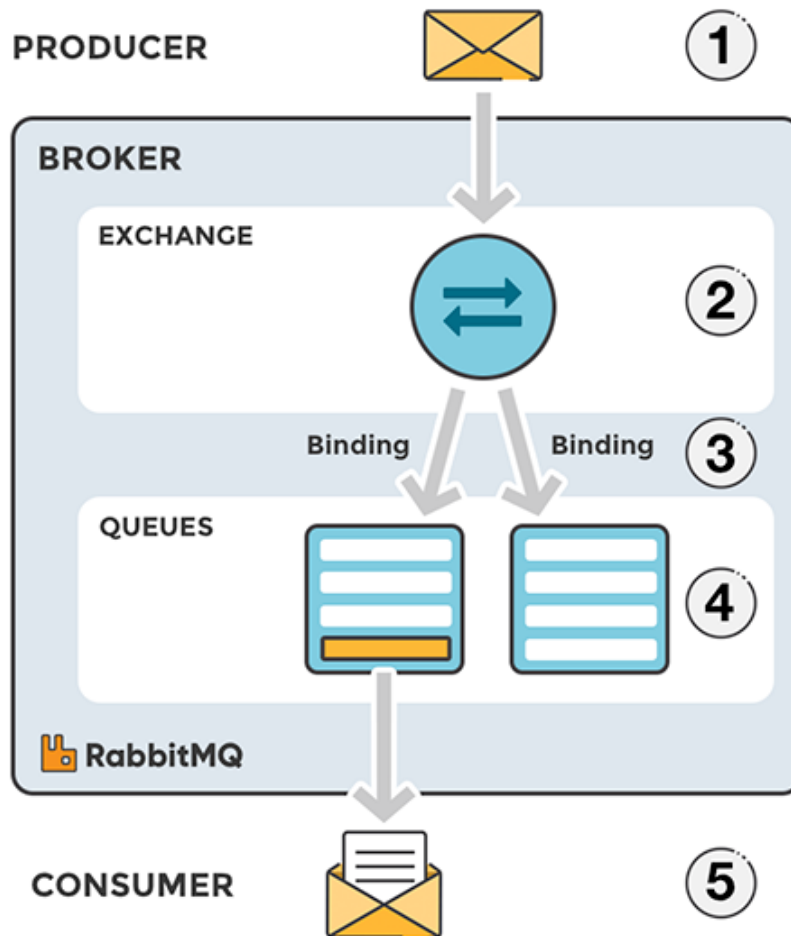


Figure: Illustration of the message flow in RabbitMQ, with five steps highlighted.

1. The producer publishes a message to an exchange. When you create the exchange, you have to specify the type of it. The different types of exchanges are explained in detail later on.
2. The exchange receives the message and is now responsible for the routing of the message. The exchange takes different message attributes into account, such as routing key, depending on the exchange type.
3. Bindings have to be created from the exchange to queues. In this case, we see two bindings to two different queues from the exchange. The

exchange routes the message into the queues, depending on message attributes.

4. The messages stay in the queue until they are handled by a consumer
5. The consumer handles the message.

Types of exchanges

Direct exchanges are used in chapter 4, with sample code in Ruby, Node.js and Python. Deeper understanding about the different exchange types, binding keys, routing keys and how/when you should use them can be found in chapter 3 about exchanges: [Chapter 3: Exchanges, routing keys and bindings](#).

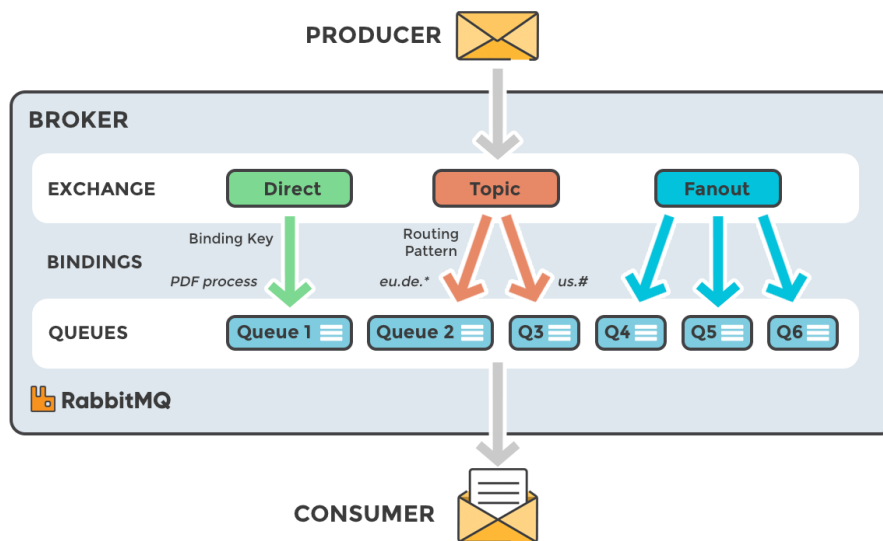


Figure: Three different exchanges: Direct, Topic, and Fanout.

- **Direct:** A direct exchange delivers messages to queues based on a message routing key. In a direct exchange, the message is routed to the queue whose binding key exactly matches the routing key of the message. If the queue is bound to the exchange with the binding key *pdfprocess*, a message published to the exchange with a routing key *pdfprocess* will be routed to that queue.
- **Fanout:** A fanout exchange routes messages to all of the queues that are bound to it.
- **Topic:** The topic exchange does a wildcard match between the routing key and the routing pattern specified in the binding.
- **Headers:** Headers exchanges use the message header attributes for routing.

RabbitMQ and server concepts

Here are some important concepts that need to be described before we dig deeper into RabbitMQ. The default virtual host, the default user, and the default permissions are used in the examples that follow, but it is still good to have a feeling of what it is.

- **Producer:**Application that sends the messages.
- **Consumer:**Application that receives the messages.
- **Queue:**Buffer that stores messages.
- **Message:**Information that is sent from the producer to a consumer through RabbitMQ.
- **Connection:**A connection is a TCP connection between your application and the RabbitMQ broker.
- **Channel:**A channel is a virtual connection inside a connection. When you are publishing or consuming messages or subscribing to a queue, it is all done over a channel.
- **Exchange:**Receives messages from producers and pushes them to queues depending on rules defined by the exchange type. In order to receive messages, a queue needs to be bound to at least one exchange.
- **Binding:**A binding is a link between a queue and an exchange.
- **Routing key:**The routing key is a key that the exchange looks at to decide how to route the message to queues. The routing key is like an address for the message.
- **AMQP:**AMQP (Advanced Message Queuing Protocol) is the main protocol used by RabbitMQ for messaging.
- **Users:**It is possible to connect to RabbitMQ with a given username and password. Every user can be assigned permissions such as rights to read, write and configure privileges within the instance. Users can also be assigned permissions to specific virtual hosts.
- **Vhost, virtual host:**A Virtual host provides a way to segregate applications using the same RabbitMQ instance. Different users can have different access privileges to different vhosts and queues, and exchanges can be created so they only exist in one vhost.

At the beginning of this chapter, we had one producer (the website application) and one consumer (the PDF processing application). If the PDF processing application crashes, or if there is a lot of PDF requests coming in the same time, messages would continue to stack up in the queue until the consumer starts again. It would then process all the messages, one by one.

Set up a RabbitMQ instance

To be able to follow this guide you need to set up a CloudAMQP instance. CloudAMQP is a hosted RabbitMQ solution, meaning that all you need to do is sign up for an account and create an instance. You do not need to set up and install RabbitMQ or care about cluster handling, CloudAMQP will do that for you. CloudAMQP can be used for free with the plan **Little Lemur**. Go to the [plan](#) page and sign up for any plan, and create an instance.

When your instance is created, click on "details" for your instance to find your username, password and connection URL for your cloud-hosted RabbitMQ instance.

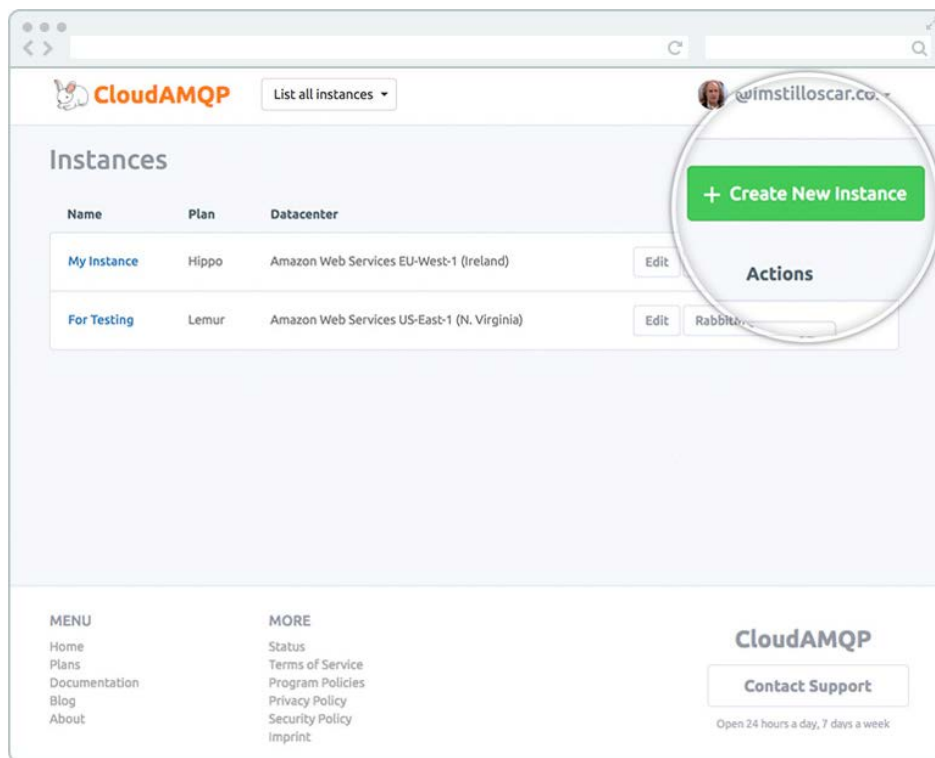
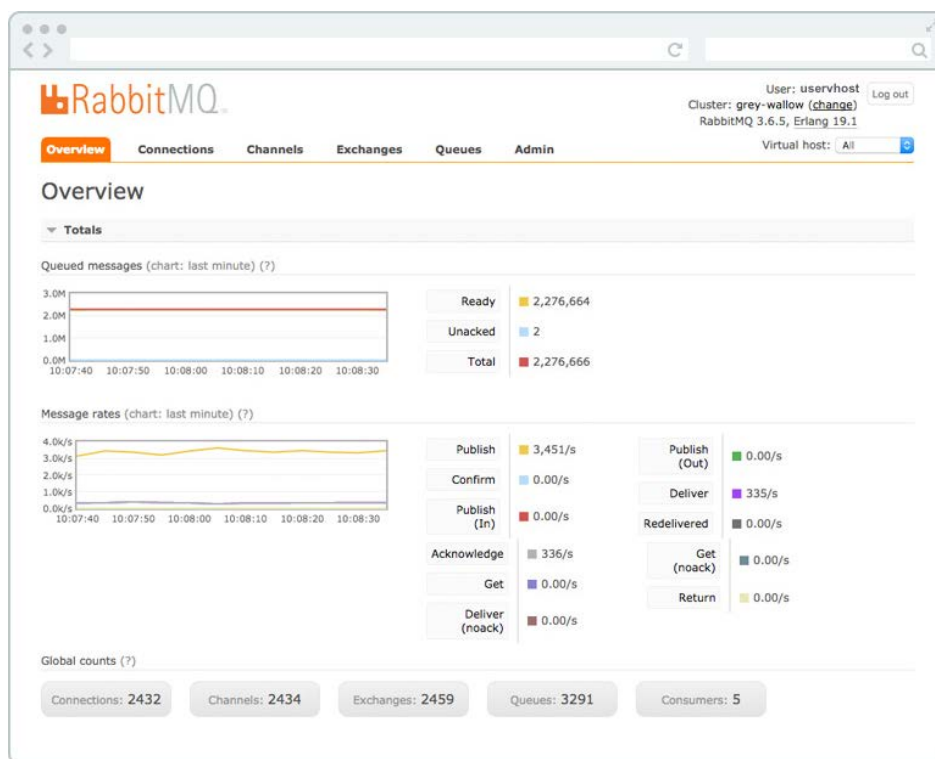


Figure: Instances in the CloudAMQP web interface.



Detailed information for an instance in the CloudAMQP Console.

Getting started with RabbitMQ

Immediately after a RabbitMQ instance has been created it is possible to send messages across languages, platforms, and operating systems. This way of handling messages decouple your processes and creates a highly

scalable system. You can now start by opening the management interface to get an overview of your RabbitMQ server.

The Management Interface - Management and Monitoring

RabbitMQ provides a web user interface (UI) for management and monitoring of your RabbitMQ server. A link to the management interface can be found on the details page for your CloudAMQP instance.

From the management interface, it is possible to handle, create, delete and list queues. It is also possible to monitor queue length, check message rate, change and add user permissions, and much more.

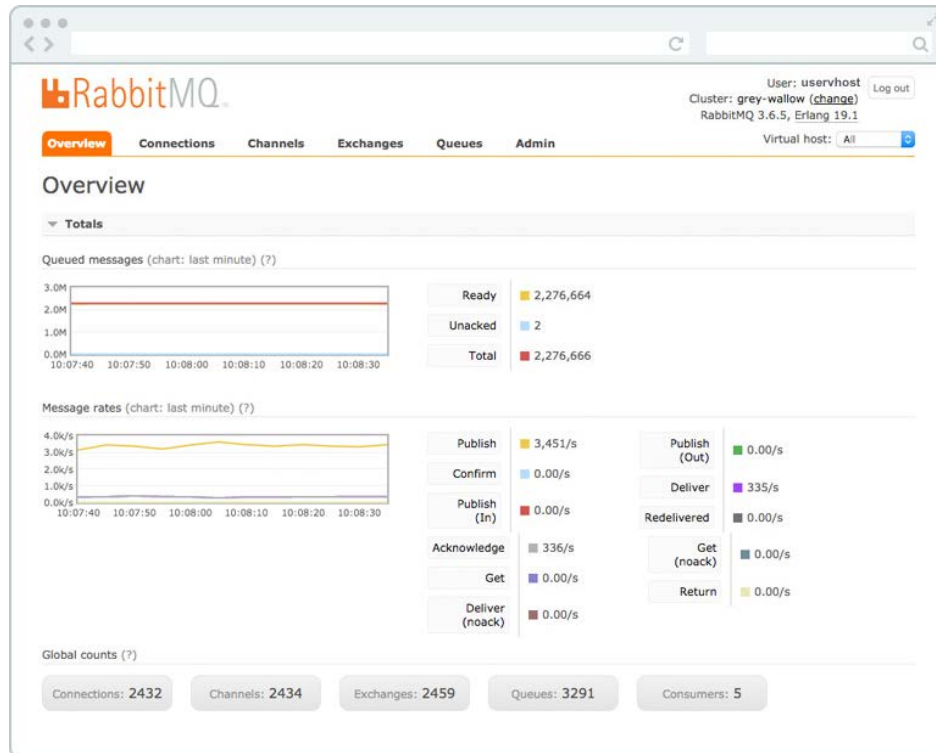


Figure: The overview window in the RabbitMQ management interface.

More information about the management interface can be found in [chapter 2](#).

Publish and subscribe messages

RabbitMQ speaks the AMQP protocol by default. To be able to communicate with RabbitMQ you need a library that understands the same protocol as RabbitMQ. You need to download the client-library for the programming language that you intend to use for your applications. A client-library is an application programming interface (API) for use in writing client applications. A client library has several methods that can be used, in this case, to communicate with RabbitMQ. The methods should be used when you, for example, connect to the RabbitMQ broker (using the given parameters, host name, port number, etc.) or when you declare a queue or an exchange.

There is a choice of libraries for all major programming languages.

Steps to follow when setting up a connection and publishing a message/consuming a message:

1. First of all, we need to set up/create a connection object. Here, the username, password, connection URL, port, etc., will be specified. A TCP connection will be set up between the application and RabbitMQ.
2. Secondly, a channel needs to be opened. A channel needs to be created in the TCP connection. The connection interface can be used to open a channel and when the channel is opened it can be used to send and receive messages.
3. Declare/create a queue. Declaring a queue will cause it to be created if it does not already exist. All queues need to be declared before they can be used.
4. **In subscriber/consumer:** Set up exchanges and bind a queue to an exchange. All exchanges need to be declared before they can be used. An exchange accepts messages from a producer application and routes them to message queues. For messages to be routed to queues, queues need to be bound to an exchange.
5. **In publisher:** Publish a message to an exchange
In subscriber/consumer: Consume a message from a queue.
6. Close the channel and the connection.

Sample code

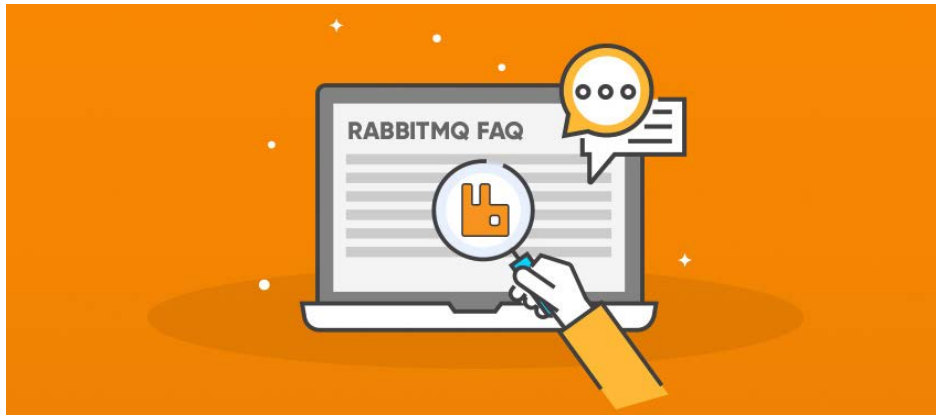
Sample code will be given in later chapters, with [Ruby](#), [Node.js](#), and [Python](#). It is possible to use different programming languages on different parts of the system. The subscriber could, for example, be written in Node.js, and the subscriber in Python.

CloudAMQP - Industry-Leading Experts at RabbitMQ

[Get a managed RabbitMQ server for FREE](#)

The Management Interface

This chapter covers the different views that you can find in the RabbitMQ Management Interface. The RabbitMQ Management is a user-friendly interface that let you monitor and handle your RabbitMQ server from a web browser.



Among other things queues, connections, channels, exchanges, users and user permissions can be handled - created, deleted and listed in the browser. You can monitor message rates and send/receive messages manually. RabbitMQ Management is a plugin that can be enabled for RabbitMQ. It gives a single static HTML page that makes background queries to the HTTP API for RabbitMQ. Information from the management interface can be useful when you are debugging your applications or when you need an overview of the whole system. If you see that the number of unacked messages starts to get high, it could mean that your consumers are getting slow. If you need to check if an exchange is working, you can try to send a test message.

The RabbitMQ management interface is enabled by default in CloudAMQP, and a link can be found on the details page for your CloudAMQP instance.

All the tabs from the menu are explained in this post. Screenshots from the views are shown for: [Overview](#), [Connections, and channels](#), [Exchanges](#), [Queues](#), and [Admin - users and permissions](#). A simple [example](#) will also show how to set up a queue an exchange and add a binding between them.

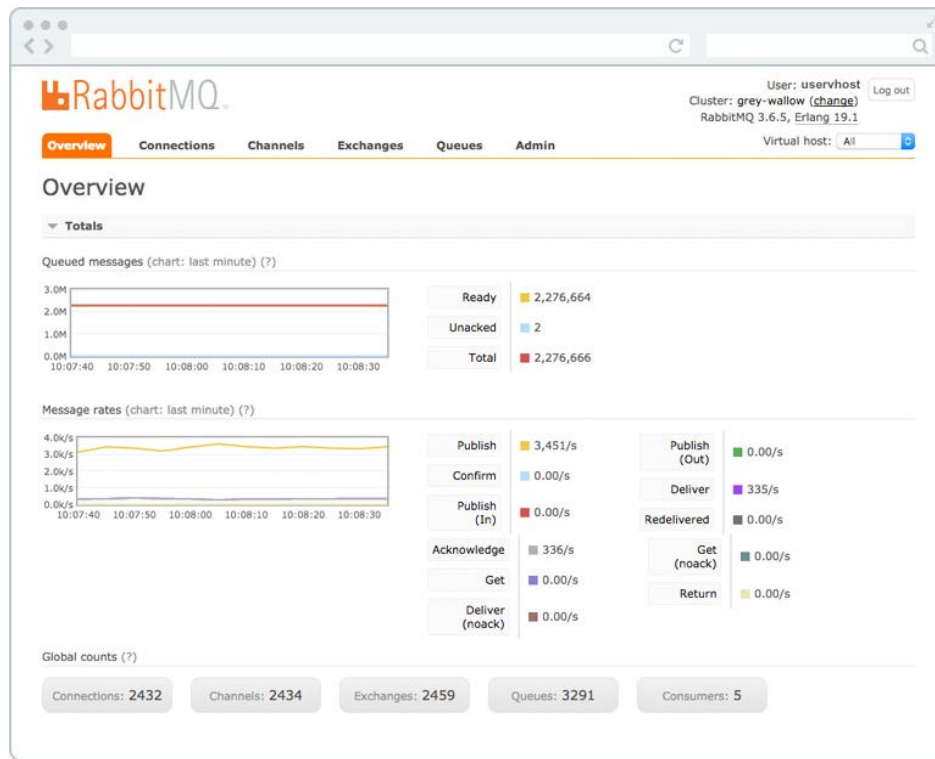


Figure: The RabbitMQ management interface.

Concepts

In the first chapter, the default virtual host, user and permissions were used in the examples. In this chapter, different views in the management interface are shown, and all the examples are still only working with the default values - still there are some important concepts that are good to be familiar with.

- **Users:** Users can be added from the management interface, and every user can be assigned permissions such as rights to read, write and configure privileges. Users can also be assigned permissions to specific virtual hosts.
- **Vhost, virtual host:** Virtual hosts provide a way to segregate applications using the same RabbitMQ instance. Different users can have different access privileges to different vhosts and queues, and exchanges can be created so they only exist in one vhost.
- **Cluster:** A cluster consists of a set of connected computers that work together. If the RabbitMQ instance consists of more than one node - it is called a RabbitMQ cluster. A cluster is a group of nodes i.e., a group of computers.
- **Node:** A node is a single computer in the RabbitMQ cluster.

Overview

The overview shows two charts, one for queued messages and one with the message rate. You can change the time interval shown in the chart by clicking the text (*chart: last minute*) above the charts. Information about all different statuses for messages can be found by pressing "?".

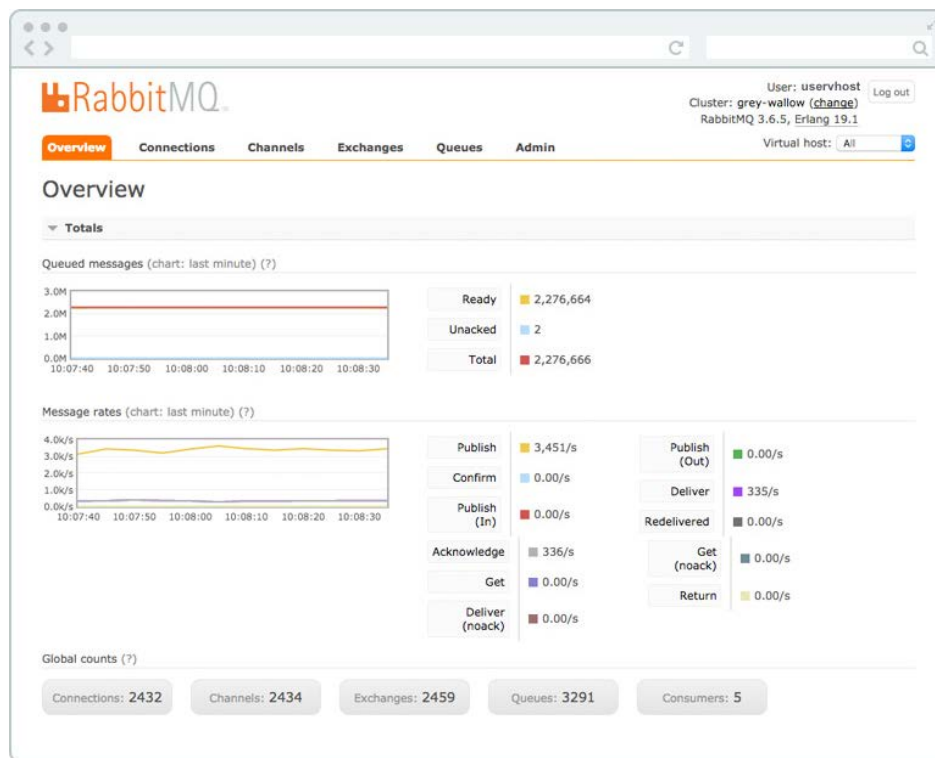


Figure: The RabbitMQ overview window, with one chart for queued messages one chart for the message rate.

Queued messages

A chart of the total number of queued messages for all your queues. **Ready** shows the number of messages that are available to be delivered. **Unacked** are the number of messages for which the server is waiting for acknowledgement.

Messages rate

A chart with the rate of how the messages are handled. **Publish** shows the rate at which messages are entering the server and **Confirm** shows a rate at which the server is confirming.

Global Count

The total number of connections, channels, exchanges, queues and consumers for ALL virtual hosts the current user has access to.

Nodes

Nodes show information about the different nodes in the RabbitMQ cluster (a cluster is a group of nodes i.e., a group of computers), or information about one single node if only one node is used. Here information about server memory, number of Erlang processes per node, and other node-specific information can be found. **Info** show i.e., further information about the node and enabled plugins.

Node					
Node: rabbit@localhost (More about this node)					
File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Info
6 138 available	539 1048576 available	49MB 1.6GB high watermark	22GB 48MB low watermark	Disc 9	Stats

Figure: Node-specific information.

Import/export definitions

It is possible to import and export configuration definitions. When you download the definitions, you get a JSON representation of your broker (your RabbitMQ settings). This can be used to restore exchanges, queues, virtual hosts, policies, and users. This feature can be used as a backup. Every time you make a change in the config, you can keep the old settings, just in case.

Import / export definitions

Export

Filename for download:

Download broker definitions (?)

Import

Definitions file:
 Ingen fil är vald.

Upload broker definitions (?)

HTTP API | Command Line

Update every 5 seconds

Figure: Import/export definitions as a JSON file.

Connections and channels

A connection is a TCP connection between your application and the RabbitMQ broker. A channel is a virtual connection inside a connection.

RabbitMQ connections and channels can be in different **states**; starting, tuning, opening, running, flow, blocking, blocked, closing, closed. If a connection enters flow-control this often means that the client is being rate-limited in some way; A good article to read when that is happening can be found [here](#).

Connections

The connections tab shows the connections established to the RabbitMQ server. **vhost** shows in which vhost the connection operates, the **username** the user associated with the connection. **Channel** tells the number of channels using the connection. **SSL/TLS** indicate whether the connection is secured with SSL.

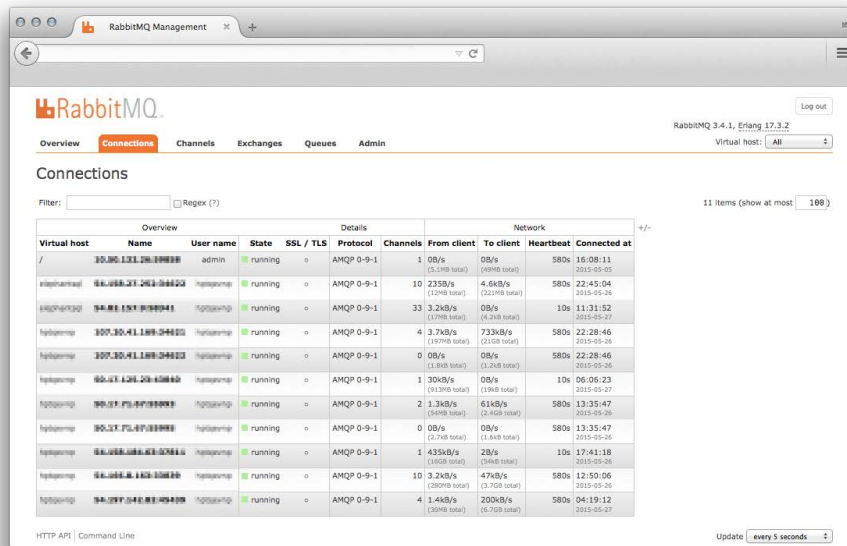


Figure: The Connections tab in the RabbitMQ management interface.

If you click on one of the connections, you get an overview of that specific connection. You can view channels within the connection and its data rates. You can see client properties, and you can close the connection.

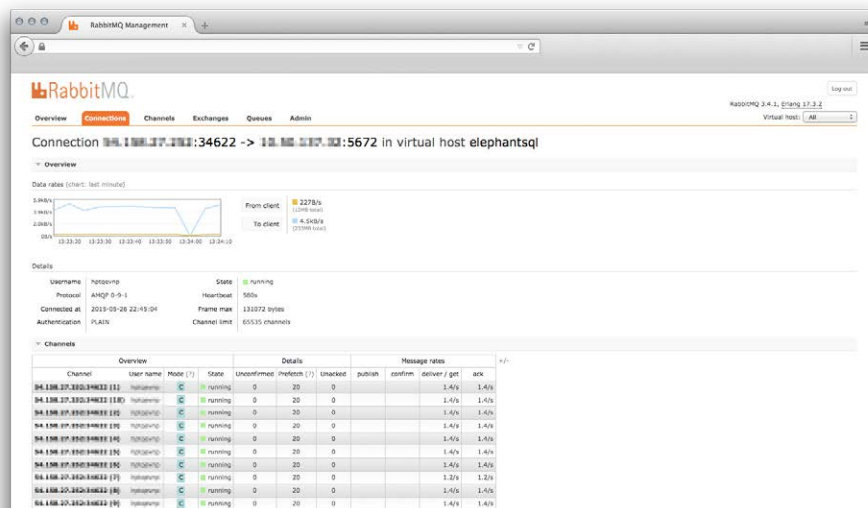


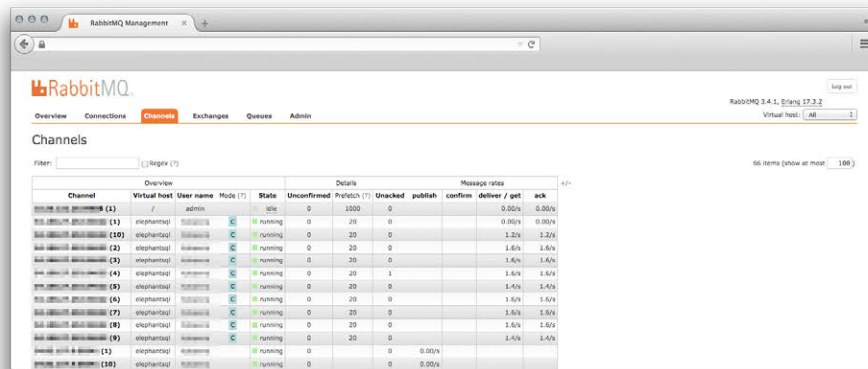
Figure: Connection information for a specific connection.

More information about the attributes associated with a connection can be found [here](#) in the manual page for `rabbitmqctl`, the command line tool for managing a RabbitMQ broker.

Channels

The channel tab shows information about all current channels. The **vhost** shows in which vhost the channel operates, the **username** the user associated with the channel. The **mode** tells the channel guarantee mode. It can be **inconfirm** or **transactional** mode. When a channel is in **confirm** mode,

both the broker and the clientcount messages. The broker then confirms messages as it handles them. Confirm mode is activated once the `confirm.select` method is used on a channel.



Channel	Virtual host	User name	Mode	State	Unconfirmed	Prefetch	Unacked	publish	confirm	deliver	get	ack
...

Figure: The Channels tab with information about all current channels.

If you click on one of the channels, you get a detailed overview of that specific channel. From here you can see the message rate and the number of logical consumers retrieving messages via the channel.

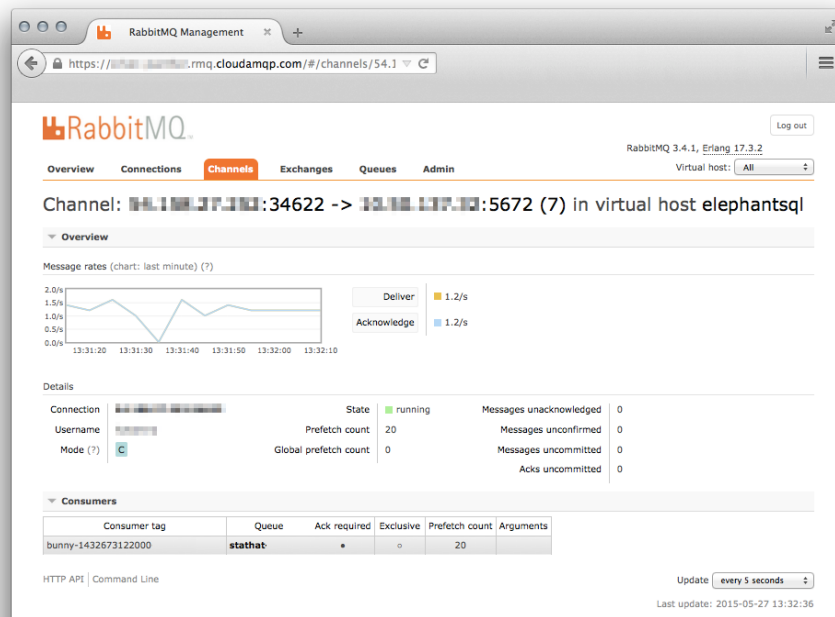


Figure: Detailed information about a specific channel.

More information about the attributes associated with a channel can be found [here](#) in the manual page for `rabbitmqctl`, the command line tool for managing a RabbitMQ broker.

Exchanges

An exchange receives messages from producers and pushes them to

queues. The exchange must know exactly what to do with a message it receives. All exchanges can be listed from the exchange tab. **Virtual host** shows the vhost for the exchange, **type** is the exchange type such as direct, topic, headers, fanout. **Features** show the parameters for the exchange (e.g. **D** stands for durable, and **AD** for auto-delete). Features and types can be specified when the exchange is created. In this list, there are some amq.* exchanges and the default (unnamed) exchange. These are created by default.

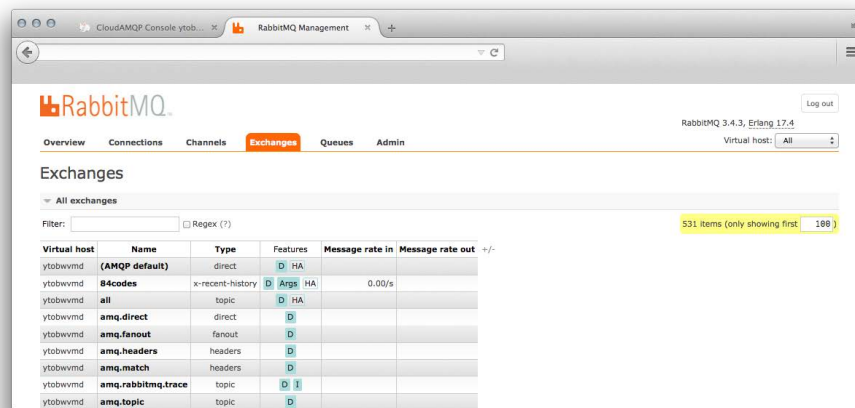


Figure: The exchanges tab in the RabbitMQ management interface.

By clicking on the exchange name, a detailed page about the exchange is shown. You can see and add bindings to the exchange. You can also publish a message to the exchange or delete the exchange.

Exchange: amq.direct in virtual host

Overview

Message rates (chart: last minute) (?)

Currently idle

Details

Type

direct

Features

durable: true

Policy

Bindings

Publish message

Delete this exchange

HTTP API

Command Line

Figure: Detailed view of an exchange.

Queues

The queue tab shows the queues for all or one selected vhost.

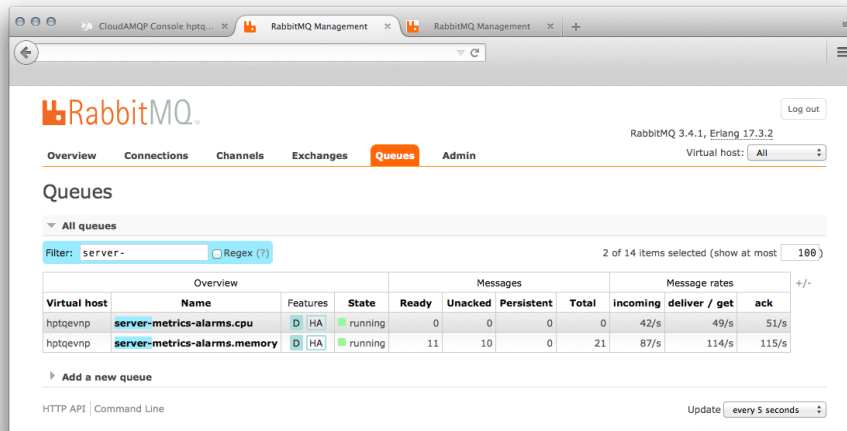


Figure: The queues tab in the RabbitMQ management interface.

Queues have different parameters and arguments depending on how they were created. The *features* column shows the parameters that belong to the queue. It could be features like *Durable queue* (which ensure that RabbitMQ will never lose the queue), *Message TTL* (which tells how long a message published to a queue can live before it is discarded), *Auto expire* (which tells how long a queue can be unused for before it is automatically deleted), *Max length* (which tells how many (ready) messages a queue can contain before it starts to drop them) and *Max length bytes* (which tells the total body size for ready messages a queue can contain before it starts to drop them).

You can also create a queue from this view.

If you press on any chosen queue from the list of queues, all information about the queue is shown like in the pictures that follow below.

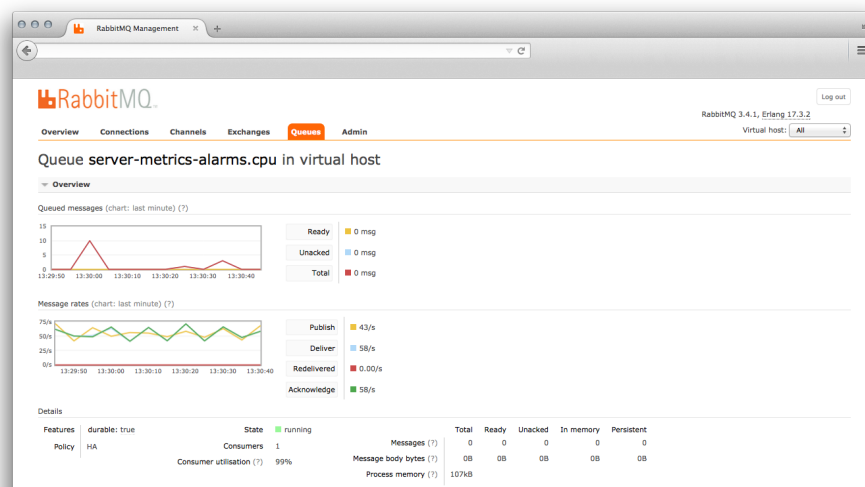


Figure: Specific information about a single queue.

The first two charts include the same information as the overview, but that

just show the number of queued messages, and the messagerates for that specific queue.

Consumers

The consumers field shows the consumers/channels that are connected to the queue.



Consumers					
Channel	Consumer tag	Ack required	Exclusive	Prefetch count	Arguments
 34621 (1)	bunny-1432672144000- 	•	○	10	

Figure: Consumers connected to a queue.

Bindings

A binding can be created between an exchange and a queue. All active bindings to the queue are shown under bindings. You can also create a new binding to a queue from here or unbind a queue from an exchange.

Bindings

From	Routing key	Arguments
(Default: exchange binding)		
exchange	routingKey	<div>Unbind</div>

⇓

This queue

Add binding to this queue

From exchange:

Routing key:

Arguments:

=

String

Bind

Figure: The bindings interface.

Publish message

It is possible to manually publish a message to the queue from "publish message". The message will be published to the default exchange with the queue name as given routing key - meaning that the message will be sent to the queue. It is also possible to publish a message to an exchange from the exchange view.

Publish message

Message will be published to the default exchange with routing key `server-metrics-alarms.cpu`, routing it to this queue.

Delivery mode: 1 - Non-persistent

Headers: (?) = String

Properties: (?) =

Payload:

Publish message

Figure: Manually publishing a message to the queue.

Get message

It is possible to manually inspect the message in the queue. "Get message" gets the message to you, and if you mark it as "requeue" RabbitMQ puts it back to the queue in the same order.

Get messages

Warning: getting messages from a queue is a destructive action. (?)

Requeue: ☒

Encoding: Auto string / base64 (?)

Messages:

Get Message(s)

Figure: Manually inspect a message in the queue.

Delete or Purge queue

A queue can be deleted by clicking the delete button, and you can empty the queue by clicking purge.

Delete / purge

Delete Purge

Figure: Delete or purge a queue from the web interface.

Admin

From the Admin view it is possible to add users and change user permissions. You can set up vhosts, policies, federation and shovels. Information about shovels can be found here:

<https://www.rabbitmq.com/shovel.html> and <https://www.cloudamqp.com/docs/shovel.html>. Information about federation

can be found here: <https://www.cloudamqp.com/blog/2015-03-24-rabbitmq-federation.html>

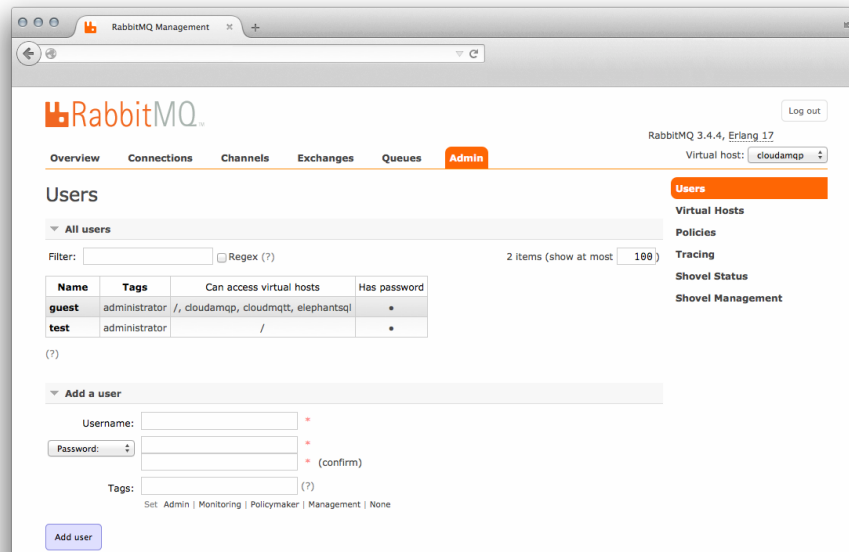


Figure: The Admin interface where users can be added.

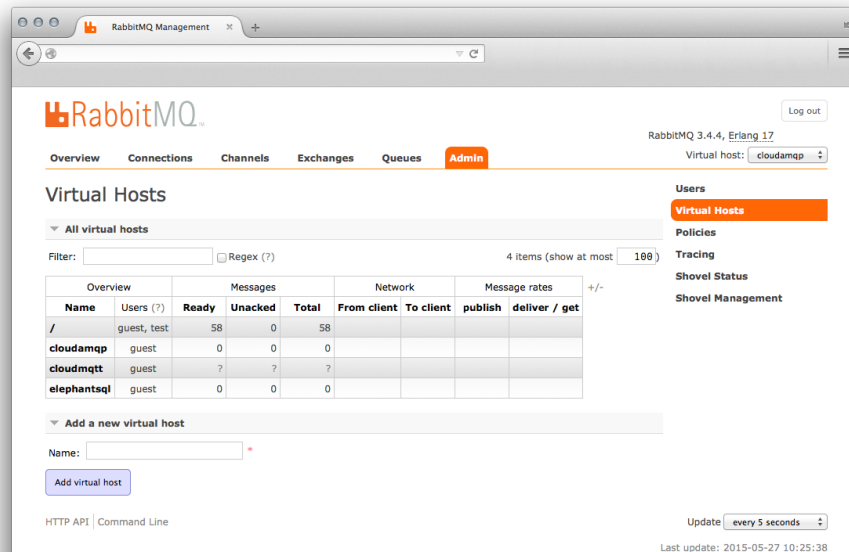


Figure: Virtual Hosts can be added from the Admin tab.

Example

This example shows how you can create a queue "example-queue" and an exchange called example.exchange.

Add a new queue

Virtual host:

/

Name:

rabbitmq-example

*

Durability:

Durable

Auto delete: (?)

No

Arguments:

=

String

Add Message TTL (?) | Auto expire (?) | Max length (?) | Max length bytes (?)

Dead letter exchange (?) | Dead letter routing key (?)

Add queue

Figure: Queue view: Add queue

Add a new exchange

Virtual host:

/

Name:

example.exchange

*

Type:

direct

Durability:

Durable

Auto delete: (?)

No

Internal: (?)

No

Arguments:

=

String

Add Alternate exchange (?)

Add exchange

Figure: Exchange view: Add exchange

The exchange and the queue are connected by a binding called "pdfprocess". Messages published to the exchange with the routing key "pdfprocess" will end up in the queue.

Add binding from this exchange

To queue

:

rabbitmq-example

*

Routing key:

pdfprocess

Arguments:

=

String

Bind

Figure: Press on the exchange or on the queue, go to "Add binding from this exchange" or "Add binding to this queue"

Publish message

Routing key: pdfprocess

Delivery mode: 1 - Non-persistent

Headers: (?)

=

String

Properties: (?)

=

Payload:

Hello CloudAMQP

Publish message

Figure: Publish a message to the exchange with the routing key "pdfprocess"

Queue rabbitmq-example in virtual host /

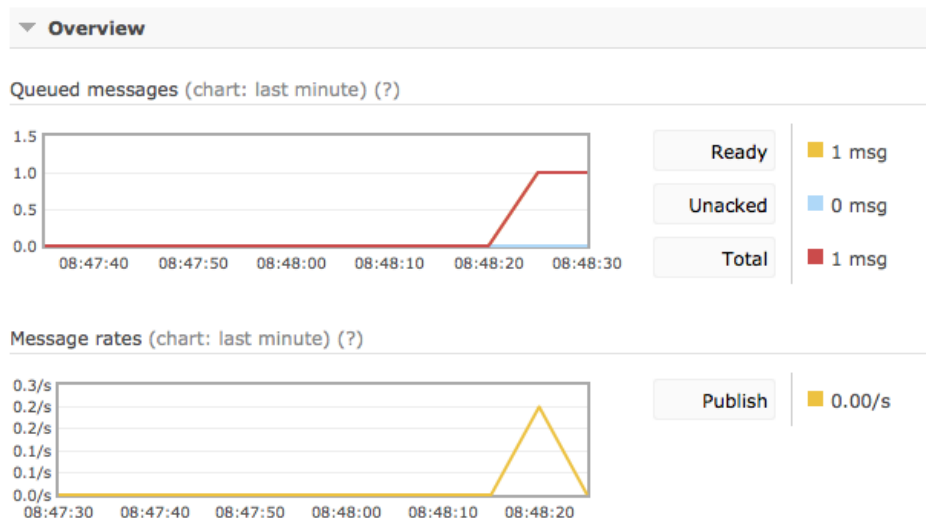
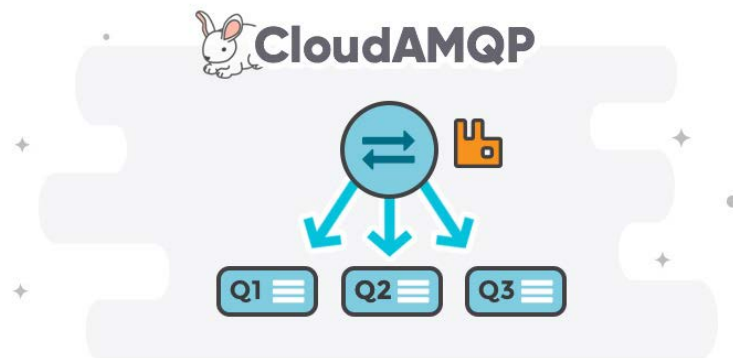


Figure: Queue overview for example-queue when a message is published.

A lot of things can be viewed and handled from the management interface, and it will give you a good overview of your system. By looking into the management interface, you will get a good understanding about RabbitMQ and how everything is related.

CloudAMQP - Industry-Leading Experts at RabbitMQ

Get a managed RabbitMQ server for [FREE](#)



Messages are not published directly to a queue, instead, the producer sends messages to an exchange. Exchanges are message routing agents, defined per virtual host within RabbitMQ. An exchange is responsible for the routing of the messages to the different queues. An exchange accepts messages from the producer application and routes them to message queues with help of header attributes, bindings, and routing keys.

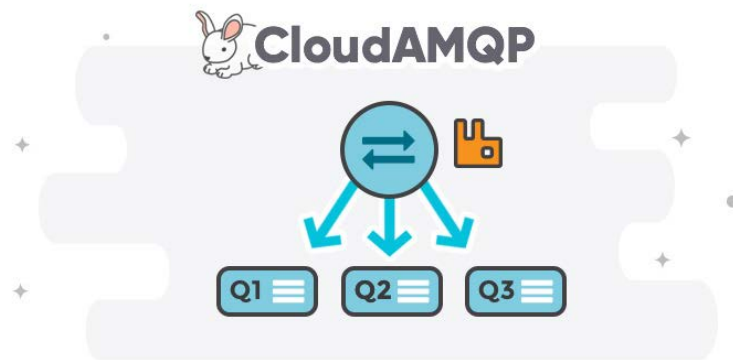
A **binding** is a "link" that you set up to bind a queue to an exchange.

The **routing key** is a message attribute. The exchange might look at this key when deciding how to route the message to queues (depending on exchange type).

Exchanges, connections and queues can be configured with parameters

Exchanges, routing keys and bindings

What is an exchange, a binding and a routing key? How are exchanges and queues associated with each other? When should I use them and how? This chapter explains the different types of exchanges in RabbitMQ and scenarios for how you can use them.



Messages are not published directly to a queue, instead, the producer sends messages to an exchange. Exchanges are message routing agents, defined per virtual host within RabbitMQ. An exchange is responsible for the routing of the messages to the different queues. An exchange accepts messages from the producer application and routes them to message queues with help of header attributes, bindings, and routing keys.

A **binding** is a "link" that you set up to bind a queue to an exchange.

The **routing key** is a message attribute. The exchange might look at this key when deciding how to route the message to queues (depending on exchange type).

Exchanges, connections and queues can be configured with parameters such as *durable*, *temporary*, and *auto delete* upon creation. Durable exchanges will survive server restarts and will last until they are explicitly deleted. Temporary exchanges exist until RabbitMQ is shutdown. Auto deleted exchanges are removed once the last bound object unbinds from the exchange.

In RabbitMQ, there are four different types of exchanges that route the message differently using different parameters and bindings setups. Clients

can create their own exchanges, or use the predefined default exchanges, the exchanges created when the server starts for the first time.

Standard RabbitMQ message flow

1. The producer publishes a message to the exchange.
2. The exchange receives the message and is now responsible for the routing of the message.
3. A binding has to be set up between the queue and the exchange. In this case, we have bindings to two different queues from the exchange. The exchange routes the message into the queues.
4. The messages stay in the queue until they are handled by a consumer.
5. The consumer handles the message.

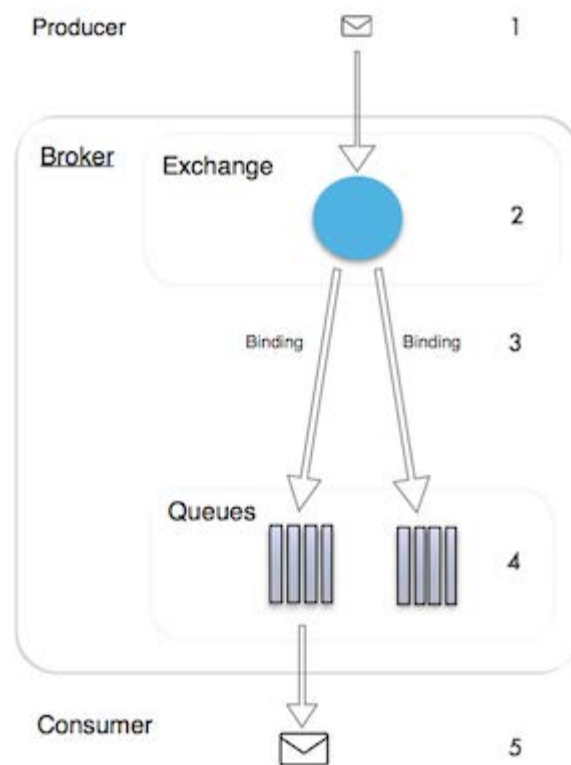


Figure: The standard RabbitMQ message flow with five steps.

Exchange types

Direct Exchange

A direct exchange delivers messages to queues based on a message routing key. The routing key is a message attribute added into the message header by the producer. The routing key can be seen as an "address" that the exchange uses to decide how to route the message. **A message goes to the queue(s) whose binding key exactly matches the routing key of the message.**

The direct exchange type is useful when you would like to distinguish messages published to the same exchange using a simple string identifier.

The default exchange AMQP brokers must provide for the direct exchange is "amq.direct".

Imagine that queue A (create_pdf_queue) in the image below (Direct Exchange Figure) is bound to a direct exchange (pdf_events) with the binding key *pdf_create*. When a new message with routing key *pdf_create* arrives at the direct exchange, the exchange routes it to the queue where the *binding_key = routing_key*, in the case to queue A (create_pdf_queue).

Scenario 1

- Exchange: pdf_events
- Queue A: create_pdf_queue
- Binding key between exchange (pdf_events) and Queue A (create_pdf_queue): pdf_create

Scenario 2

- Exchange: pdf_events
- Queue B: pdf_log_queue
- Binding key between exchange (pdf_events) and Queue B (pdf_log_queue): pdf_log

Example

Example: A message with routing key *pdf_log* is sent to the exchange *pdf_events*. The message is routed to pdf_log_queue because the routing key (pdf_log) matches the binding key (pdf_log).

If the message routing key does not match any binding key, the message will be discarded.

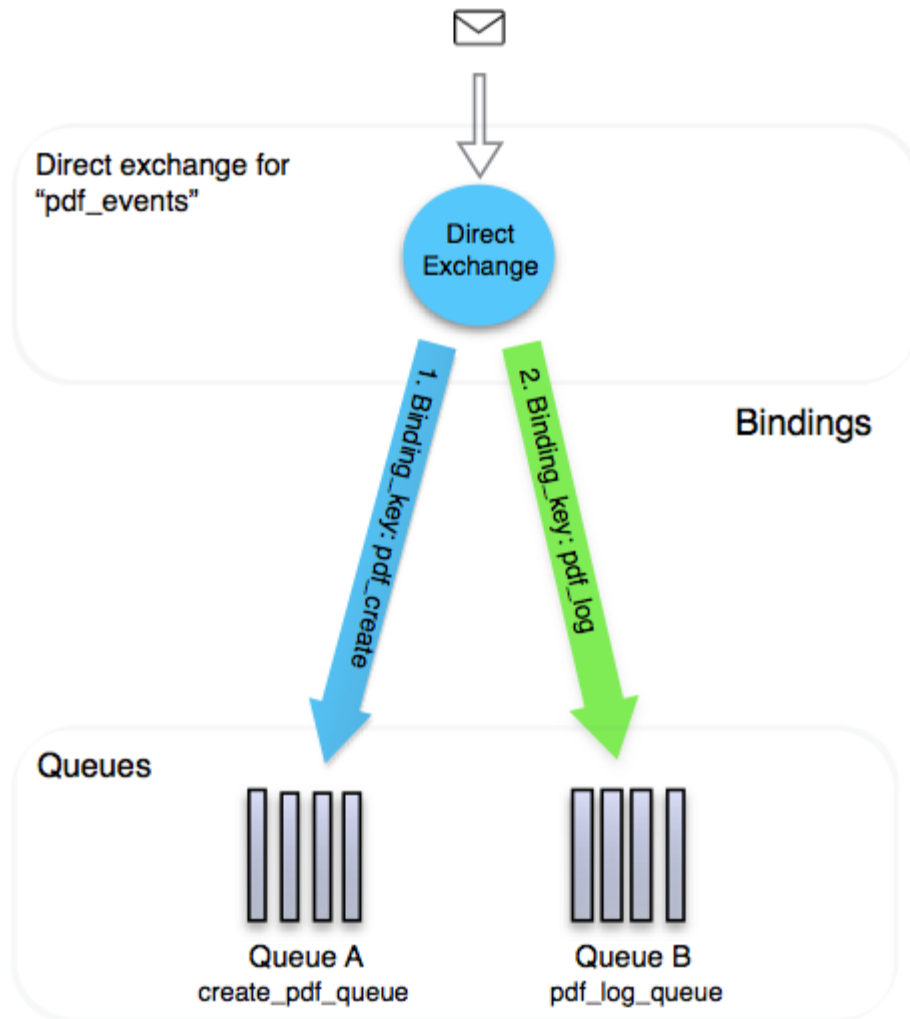


Figure: Direct Exchange Figure: A message goes to the queues whose binding key exactly matches the routing key of the message.

Default exchange

The default exchange is a pre-declared direct exchange with no name, usually referred to by the empty string `""`. When you use the default exchange, your message will be delivered to the queue with a name equal to the routing key of the message. Every queue is automatically bound to the default exchange with a routing key which is the same as the queue name.

Topic Exchange

Topic exchanges route messages to queues based on wildcard matches between the routing key and something called the routing pattern specified by the queue binding. Messages are routed to one or many queues based on a matching between a message routing key and this pattern.

The routing key must be a list of words, delimited by a period (`.`), examples

are `agreements.us` and `agreements.eu.stockholm` which in this case identifies agreements that are set up for a company with offices in lots of different locations. The routing patterns may contain an asterisk ("`*`") to match a word in a specific position of the routing key (e.g. a routing pattern of "`agreements.*.b.*`" will only match routing keys where the first word is "agreements" and the fourth word is "b"). A pound symbol ("`#`") indicates match on zero or more words (e.g. a routing pattern of "`agreements.eu.berlin.#`" matches any routing keys beginning with "agreements.eu.berlin").

The consumers indicate which topics they are interested in (like subscribing to a feed for an individual tag). The consumer creates a queue and sets up a binding with a given routing pattern to the exchange. **All messages with a routing key that match the routing pattern will be routed to the queue and stay there until the consumer consumes the message.**

The default exchange AMQP brokers must provide for the topic exchange is "`amq.topic`".

Scenario 1

The topic exchange figure shows an example where consumer A is interested in all the agreements in Berlin.

- Exchange: `agreements`
- Queue A: `berlin_agreements`
- Routing pattern between exchange (`agreements`) and Queue A (`berlin_agreements`): `agreements.eu.berlin.#`
- Example of message routing key that will match: `agreements.eu.berlin` and `agreements.eu.berlin.headstore`

Scenario 2

Consumer B is interested in all the agreements.

- Exchange: `agreements`
- Queue B: `all_agreements`
- Routing pattern between exchange (`agreements`) and Queue B (`all_agreements`): `agreements.#`
- Example of message routing key that will match: `agreements.eu.berlin` and `agreements.us`

Scenario 3

Consumer C is interested in all agreements for European headstores.

- Exchange: `agreements`
- Queue C: `headstore_agreements`
- Routing pattern between exchange (`agreements`) and Queue C (`headstore_agreements`): `agreements.eu.*.headstore`

- Example of message routing keys that will match:
 agreements.eu.berlin.headstore and agreements.eu.stockholm.headstore

Example

A message with routing key `agreements.eu.berlin` is sent to the exchange `agreements`. The message is routed to the queue `berlin_agreements` because the routing pattern of `"agreements.eu.berlin.#"` matches any routing keys beginning with `"agreements.eu.berlin"`. The message is also routed to the queue `all_agreements` because the routing key (`agreements.eu.berlin`) matches the routing pattern (`agreements.#`).

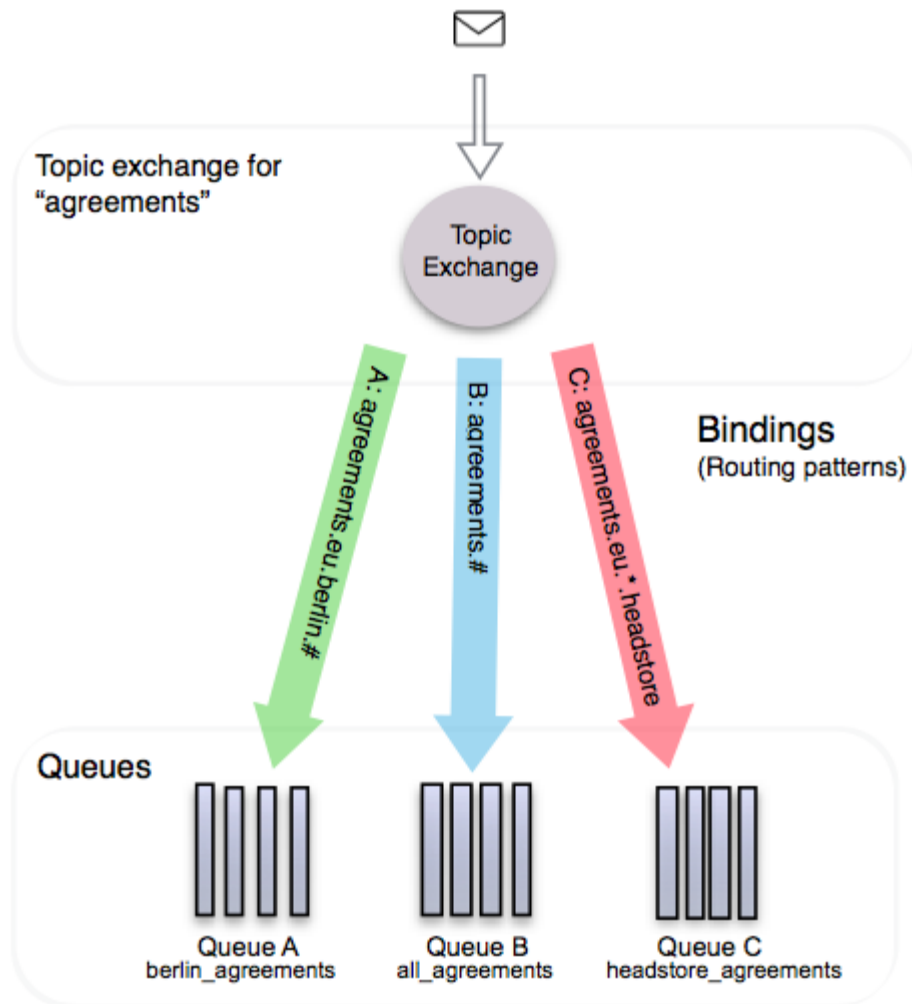


Figure: Topic Exchange: Messages are routed to one or many queues based on a matching between a message routing key and the routing pattern.

Fanout Exchange

The fanout exchange copies and routes a received message to all queues that are bound to it regardless of routing keys or pattern matching

as with direct and topic exchanges. Keys provided will simply be ignored.

Fanout exchanges can be useful when the same message needs to be sent to one or more queues with consumers who may process the same message in different ways.

The Fanout Exchange figure shows an example where a message received by the exchange is copied and routed to all three queues that are bound to the exchange. It could be sport or weather news updates that should be sent out to each connected mobile device when something happens.

The default exchange AMQP brokers must provide for the fanout exchange is "amq.fanout".

Scenario 1

- Exchange: sport_news
- Queue A: Mobile client queue A
- Binding: Binding between the exchange (sport_news) and Queue A (Mobile client queue A)

Example

A message is sent to the exchange *sport_news*. The message is routed to all queues (Queue A, Queue B, Queue C) because all queues are bound to the exchange. Provided routing keys are ignored.

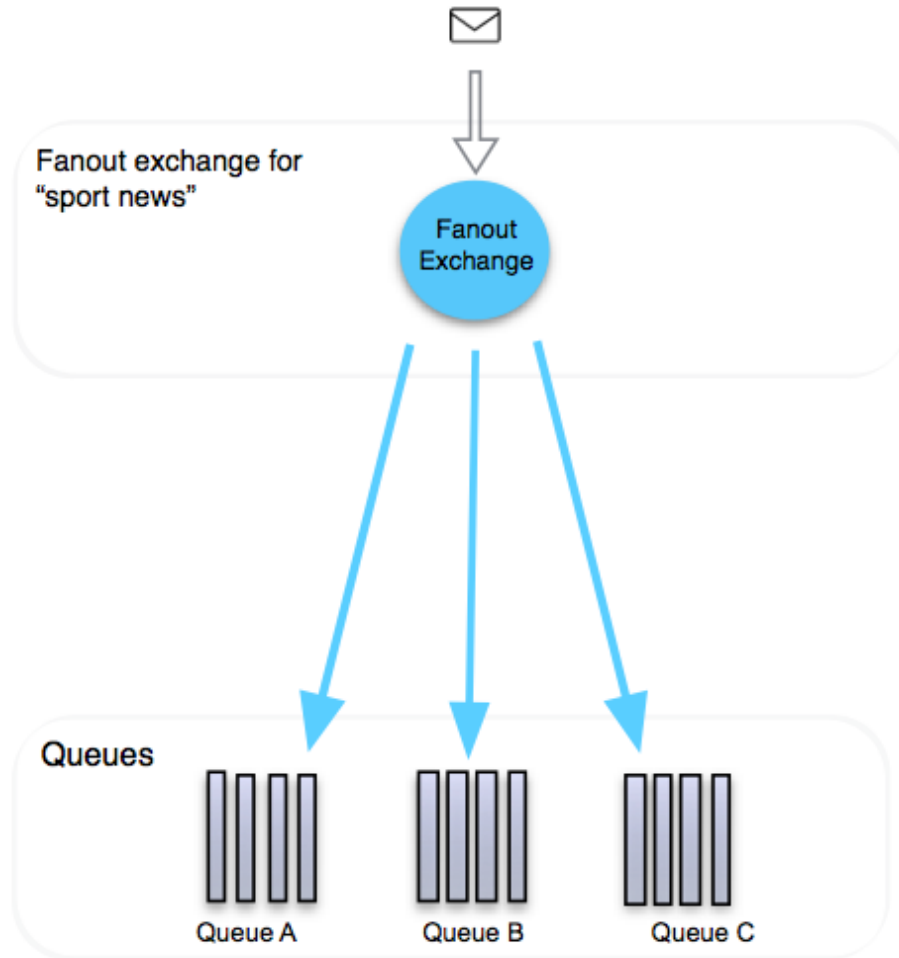


Figure: Fanout Exchange: Received messages are routed to all queues that are bound to the exchange.

Headers Exchange

Headers exchanges route based on arguments containing headers and optional values. Headers exchanges are very similar to topic exchanges, but it routes based on header values instead of routing keys. A message is considered matching if the value of the header equals the value specified upon binding.

A special argument named "x-match" tells if all headers must match, or just one. The "x-match" property can have two different values: "any" or "all", where "all" is the default value. A value of "all" means all header pairs (key, value) must match and a value of "any" means at least one of the header pairs must match. Headers can be constructed using a wider range of data types - integer or hash for example instead of a string. The headers exchange type (used with "any") is useful for directing messages which may contain a subset of known (unordered) criteria.

The default exchange AMQP brokers must provide for the headers exchange is "amq.headers".

- Exchange: Binding to Queue A with arguments (key = value): format = pdf, type = report
- Exchange: Binding to Queue B with arguments (key = value): format = pdf, type = log
- Exchange: Binding to Queue C with arguments (key = value): format = zip, type = report

Scenario 1

Message 1 is published to the exchange with header arguments (key = value): "format = pdf", "type = report" and "x-match = all"

Message 1 is delivered to Queue A - since all key/value pairs match

Scenario 2

Message 2 is published to the exchange with header arguments of (key = value): "format = pdf" and "x-match = any"

Message 2 is delivered to Queue A and Queue B - since the queue is configured to match any of the headers (format or log).

Scenario 3

Message 3 is published to the exchange with header arguments of (key = value): "format = zip", "type = log" and "x-match = all"

Message 3 is not delivered to any queue - since the queue is configured to match all of the headers (format or log).

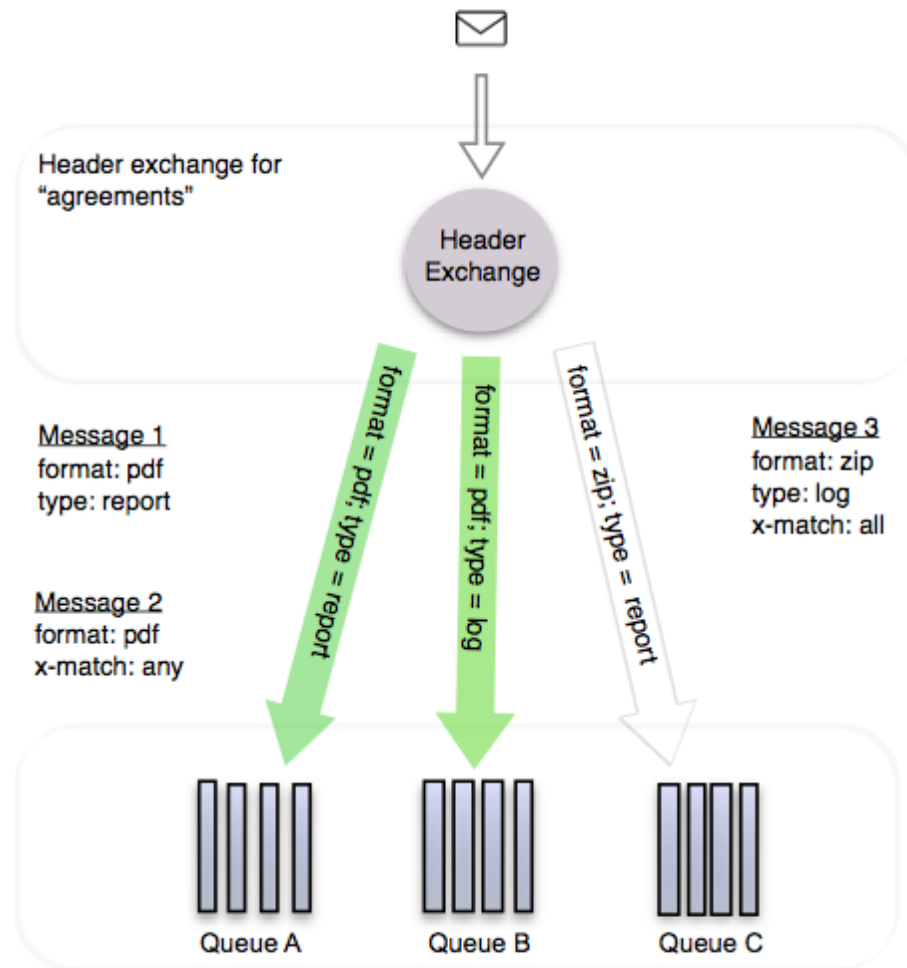


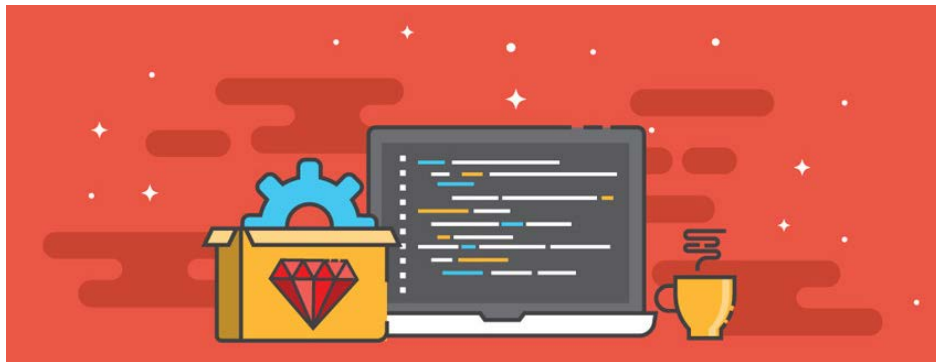
Figure: Headers exchange routes messages to queues that are bound using arguments (key and value) containing headers and optional values.

Dead Letter Exchange

If no matching queue can be found for the message, the message will be silently dropped. RabbitMQ provides an AMQP extension known as the "Dead Letter Exchange" - the dead letter exchange provides functionality to capture messages that are not deliverable.

CloudAMQP - Industry-Leading Experts at RabbitMQ

[Get a managed RabbitMQ server for FREE](#)



Start by downloading the client-library for Ruby. Ruby developers have a number of options for AMQP client libraries. In this example [Bunny](#) will be used, a synchronous client for publishing and consuming messages.

You need a RabbitMQ instance to get started. A free RabbitMQ instance can be set up for test in CloudAMQP, read about how to set up an instance in [chapter 1](#).

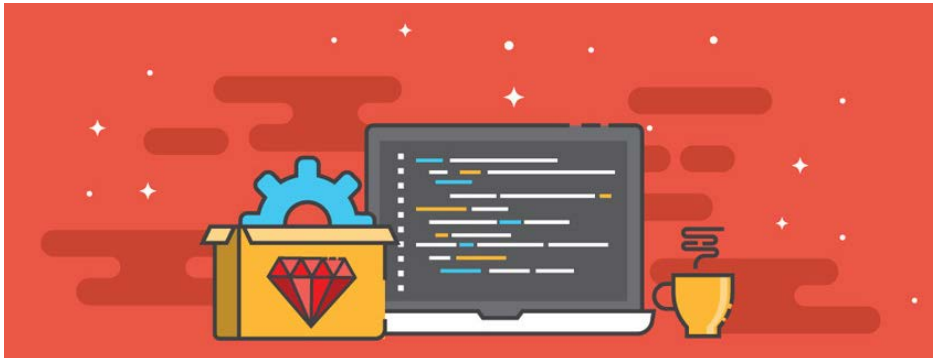
When running the full `example_publisher.rb` code below, a connection will be established between the RabbitMQ instance and your application. Queues and exchanges will be declared and created if they do not already exist and finally a message will be published.

The `example_consumer.rb` code sets up a connection and subscribes to a queue. The messages are handled one by one and sent to the PDF processing method.

The full code for the publisher and the consumer is first given, after that the code will be divided into blocks and explained per code block.

RabbitMQ and Ruby

This part of the book creates a web application that allows users to upload user information. The website will handle the information, generate a PDF and email it back to the user. Generating the PDF and sending the email will in this example case take several seconds.



Start by downloading the client-library for Ruby. Ruby developers have a number of options for AMQP client libraries. In this example [Bunny](#) will be used, a synchronous client for publishing and consuming messages.

You need a RabbitMQ instance to get started. A free RabbitMQ instance can be set up for test in CloudAMQP, read about how to set up an instance in [chapter 1](#).

When running the full `example_publisher.rb` code below, a connection will be established between the RabbitMQ instance and your application. Queues and exchanges will be declared and created if they do not already exist and finally a message will be published.

The `example_consumer.rb` code sets up a connection and subscribes to a queue. The messages are handled one by one and sent to the PDF processing method.

The full code for the publisher and the consumer is first given, after that the code will be divided into blocks and explained per code block.

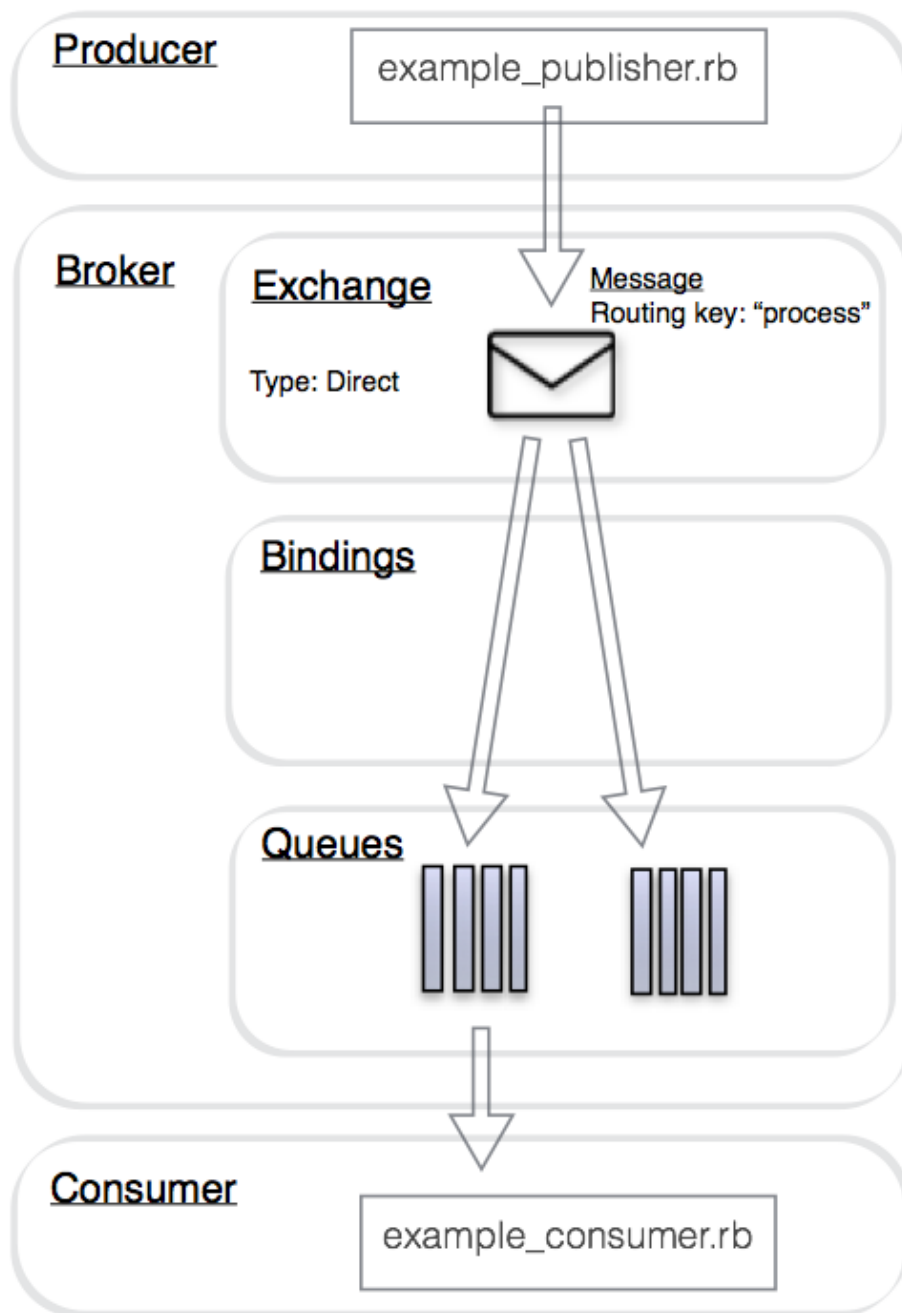


Figure: The Ruby application architecture.

Full code

```
#example_publisher.rb
require "rubygems"
require "bunny"
require "json"

#Returns a connection instance
```

```

conn = Bunny.new ENV['CLOUDAMQP_URL']
#The connection will be established when start is called
conn.start

#create a channel in the TCP connection
ch = conn.create_channel

#Declare a queue with a given name, examplequeue. In this exam
ple is a durable shared queue used.
q = ch.queue("examplequeue", :durable => true)

#Bind a queue to an exchange
q.bind(x, :routing_key => "process")

#For messages to be routed to queues, queues need to be bound
to exchanges.
x = ch.direct("example.exchange", :durable => true)
information_message = "{\"email\": \"example@mail.com\", \"name
\": \"name\", \"size\": \"size\"}"

x.publish(information_message,
  :timestamp      => Time.now.to_i,
  :routing_key    => "process"
)

sleep 1.0
conn.close

```

```

#example_consumer.rb
require "rubygems"
require "bunny"
require "json"

#Returns a connection instance
conn = Bunny.new ENV['CLOUDAMQP_URL']
#The connection will be established when start is called
conn.start

#Create a channel in the TCP connection
ch = conn.create_channel
#Declare a queue with a given name, examplequeue. In this exam
ple is a durable shared queue used.
q = ch.queue("examplequeue", :durable => true)

#Method for the PDF processing

```

```
def pdf_processing(json_information_message)
  puts "Handling pdf processing for:"
  puts json_information_message['email']
  sleep 5.0
  puts "pdf processing done"
end

#Set up the consumer to subscribe from the queue
q.subscribe(:block => true) do |delivery_info, properties, payload|
  json_information_message = JSON.parse(payload)
  pdf_processing(json_information_message)
end
```

Tutorial source code - Publisher

Set up a connection

```
#example_consumer.rb
require "rubygems"
require "bunny"
require "json"

#Returns a connection instance
conn = Bunny.new ENV['CLOUDAMQP_URL']

#The connection will be established when start is called
conn.start
```

`Bunny.new` returns a connection instance. Use the `CLOUDAMQP_URL` as connectionURL, it can be found on the details page for your CloudAMQP instance. The `CLOUDAMQP_URL` is a string including data for the instance, such as username, password, hostname, and vhost. The connection will be established when start is called `conn.start`

Create a channel

```
#Create a channel in the TCP connection
ch = conn.create_channel
```

A channel needs to be created in the TCP connection. A channel is a virtual connection inside a connection, and it is used to send AMQP commands to the broker. When you are publishing or consuming messages or subscribing to a queue is it all done over a channel.

Declare a queue

```
#Declare a queue with a given name
q = ch.queue("examplequeue", :durable => true)
```

`ch.queue` is used to declare a queue with a particular name, in this case, the queue is called *examplequeue*. The queue in this example is marked as durable, meaning that RabbitMQ will never lose our queue.

Bind the queue to an exchange

```
#For messages to be routed to queues, queues need to be bound
to exchanges.
x = ch.direct("example.exchange", :durable => true)

#Bind a queue to an exchange
q.bind(x, :routing_key => "process")
```

A direct exchange will be used, a direct exchange delivers messages to queues based on a message *routing key*. We will use the routing key "process". The exchange is first created and then bound to the queue.

Publish a message

```
information_message = "{\"email\": \"example@mail.com\", \"name\": \"name\", \"size\": \"size\"}"

x.publish(information_message,
  :timestamp      => Time.now.to_i,
  :routing_key    => "process"
)
```

`information_message` is all the information that will be sent to the exchange. The direct exchanges use the message routing key for routing, meaning the message producers need to specify the routing key in the message.

Close the connection

```
sleep 1.0
conn.close
```

`conn.close` will automatically close all channels of the connection.

Tutorial source code - Consumer

Sample code for pdf processing

```
#Method for the pdf processing
def pdf_processing(json_information_message)
  puts "Handling pdf processing for:"
  puts json_information_message['email']
  sleep 5.0
  puts "pdf processing done"
end
```

The method `pdf_processing` is a "todo" method that will sleep for 5 seconds to simulate the pdf processing.

Set up the consumer

```
#Set up the consumer to subscribe from the queue
q.subscribe(:block => true) do |delivery_info, properties, payload|
  json_information_message = JSON.parse(payload)
  pdf_processing(json_information_message)
end
```

`subscribe` takes a block and processes it. It will be called every time a message arrives.

More information about Ruby and CloudAMQP can be found in the online documentation [here](#).

CloudAMQP - Industry-Leading Experts at RabbitMQ

[Get a managed RabbitMQ server for FREE](#)

RabbitMQ and Node.js

Chapter 4.2 of **Getting Started with RabbitMQ and CloudAMQP** explains how to get started with RabbitMQ and Node.js.



This tutorial follows the scenario used in the previous chapter where a web application allows users to upload user information. The website will handle the information and generate a PDF and email it back to the user. Generating the PDF and sending the email will in this scenario take several seconds.

Getting started with RabbitMQ and Node.js

Start by downloading the client-library for Node.js. Node developers have a number of options for AMQP client libraries. In this example `amqplib` will be used. Start by adding `amqplib` as a dependency to your `package.json` file.

You need a RabbitMQ instance to get started. A free RabbitMQ instance can be set up for test in CloudAMQP, read about how to set up an instance in [chapter 1](#).

When running the full code given, a connection will be established between the RabbitMQ instance and your application. Queues and exchanges will be declared and created if they do not already exist and finally a message will be published. The publish method will queue messages internally if the connection is down and resend them later. The consumer subscribes to the queue. The messages are handled one by one and sent to the PDF processing method.

A new message will be published every second. A default exchange, identified by the empty string (`""`) will be used. The default exchange means that messages are routed to the queue with the name specified by `routing_key`, if it exists. (The default exchange is a direct exchange with no

name)

Full code can be downloaded from [GitHub](#).

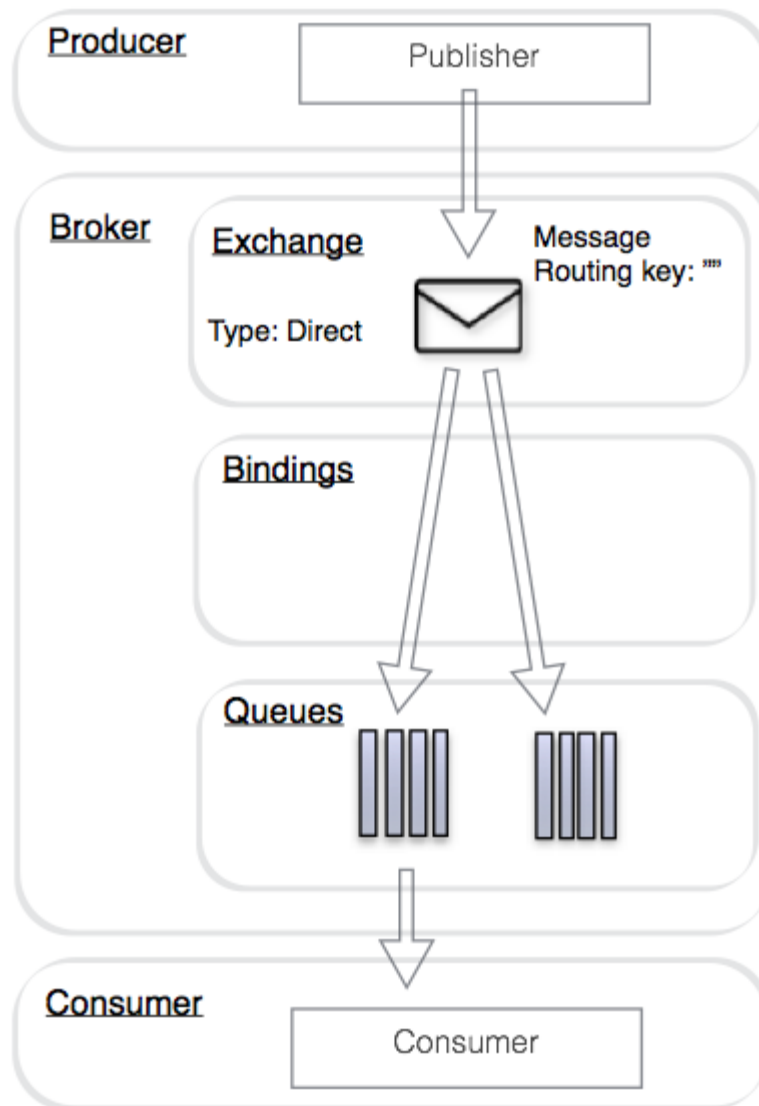


Figure: RabbitMQ and Node.js application architecture.

Tutorial source code

Load amqplib

```
# Access the callback-based API
var amqp = require('amqplib/callback_api');
var amqpConn = null;
```

Set up a connection

```
function start() {
  amqp.connect(process.env.CLOUDAMQP_URL + "?heartbeat=60", fu
```

```

function start(err, conn) {
  if (err) {
    console.error("[AMQP]", err.message);
    return setTimeout(start, 1000);
  }
  conn.on("error", function(err) {
    if (err.message !== "Connection closing") {
      console.error("[AMQP] conn error", err.message);
    }
  });
  conn.on("close", function() {
    console.error("[AMQP] reconnecting");
    return setTimeout(start, 1000);
  });
  console.log("[AMQP] connected");
  amqpConn = conn;
  whenConnected();
}

```

The `start` function will establish a connection to RabbitMQ. If the connection is closed or fails to be established, it will try to reconnect.

`amqpConn` will hold the connection and channels will be set up in the connection.

`whenConnected` will be called when a connection is established.

```

function whenConnected() {
  startPublisher();
  startWorker();
}

```

The function `whenConnected` calls two functions, one function that starts the publisher and one that starts the worker (the consumer).

Start the publisher

```

var pubChannel = null;
var offlinePubQueue = [];
function startPublisher() {
  amqpConn.createConfirmChannel(function(err, ch) {
    if (closeOnErr(err)) return;
    ch.on("error", function(err) {
      console.error("[AMQP] channel error", err.message);
    });
  });
}

```



```

    ch.on("close", function() {
        console.log("[AMQP] channel closed");
    });

    pubChannel = ch;
    while (true) {
        var m = offlinePubQueue.shift();
        if (!m) break;
        publish(m[0], m[1], m[2]);
    }
    });
}

```

`createConfirmChannel` opens a channel which uses "confirmation mode". A channel in confirmation mode requires each published message to be 'acked' or 'nacked' by the server, thereby indicating that it has been handled.

`offlinePubQueue` is an internal queue for messages that could not be sent when the application was offline. The application will check this queue and send the messages in the queue if a message is added to the queue.

Publish

```

function publish(exchange, routingKey, content) {
    try {
        pubChannel.publish(exchange, routingKey, content, { persistent: true },
            function(err, ok) {
                if (err) {
                    console.error("[AMQP] publish", err);
                }
                offlinePubQueue.push([exchange, routingKey, content]);
                pubChannel.connection.close();
            }
        );
    } catch (e) {
        console.error("[AMQP] publish", e.message);
        offlinePubQueue.push([exchange, routingKey, content]);
    }
}

```

The `publish` function will publish a message to an exchange with a given routing key. If an error occurs the message will be added to the internal queue, `offlinePubQueue`.

Consumer

```
// A worker that acks messages only if processed successfully
function startWorker() {
  amqpConn.createChannel(function(err, ch) {
    if (closeOnErr(err)) return;
    ch.on("error", function(err) {
      console.error("[AMQP] channel error", err.message);
    });
    ch.on("close", function() {
      console.log("[AMQP] channel closed");
    });

    ch.prefetch(10);
    ch.assertQueue("jobs", { durable: true }, function(err, _ok) {
      if (closeOnErr(err)) return;
      ch.consume("jobs", processMsg, { noAck: false });
      console.log("Worker is started");
    });
  });
}
```

`amqpConn.createChannel` creates a channel on the connection.

`ch.assertQueue` assert a queue into existence. `ch.consume` sets up a consumer with a callback to be invoked with each message it receives. The function called for each message is called `processMsg`

```
function processMsg(msg) {
  work(msg, function(ok) {
    try {
      if (ok)
        ch.ack(msg);
      else
        ch.reject(msg, true);
    } catch (e) {
      closeOnErr(e);
    }
  });
}
```

`processMsg` processes the message. It will call the work function and wait for it to finish.

```
function work(msg, cb) {  
  console.log("PDF processing of ", msg.content.toString());  
  cb(true);  
}
```

The `work` function includes the handling of the message information and the creation of the PDF. It is in this example a todo-function.

Close the connection on error

```
function closeOnErr(err) {  
  if (!err) return false;  
  console.error("[AMQP] error", err);  
  amqpConn.close();  
  return true;  
}
```

Publish

```
setInterval(function() {  
  publish("", "jobs", new Buffer("work work work"));  
}, 1000);  
  
start();
```

A new message will be published every second. A default exchange, identified by the empty string ("") will be used. The default exchange means that messages are routed to the queue with the name specified by `routing_key` if it exists. (The default exchange is a direct exchange with no name)

More information about Node.js and CloudAMQP can be found in the online documentation [here](#).

CloudAMQP - Industry-Leading Experts at RabbitMQ

[Get a managed RabbitMQ server for FREE](#)

RabbitMQ and Python

Chapter 4.3 of **Getting Started with RabbitMQ and CloudAMQP** explains how to get started with RabbitMQ and Python.



This tutorial follows the scenario used in the previous two chapters, where a web application allows users to enter user information into a website. The website will handle the information and generate a PDF and email it back to the user. Generating the PDF and sending the email will in this scenario take several seconds.

Getting started with RabbitMQ and Python

Start by downloading the client-library for Python. The recommended library for Python is [Pika](#). Put `pika==0.9.14` in your `requirements.txt` file.

You need a RabbitMQ instance to get started. Read about how to set up an instance in [chapter 1](#).

When running the full code given, a connection will be established between the RabbitMQ instance and your application. Queues and exchanges will be declared and created if they do not already exist and finally a message will be published. The consumer subscribes to the queue, and the messages are handled one by one and sent to the PDF processing method.

A default exchange, identified by the empty string ("") will be used. The default exchange means that messages are routed to the queue with the name specified by `routing_key` if it exists. (The default exchange is a direct exchange with no name)

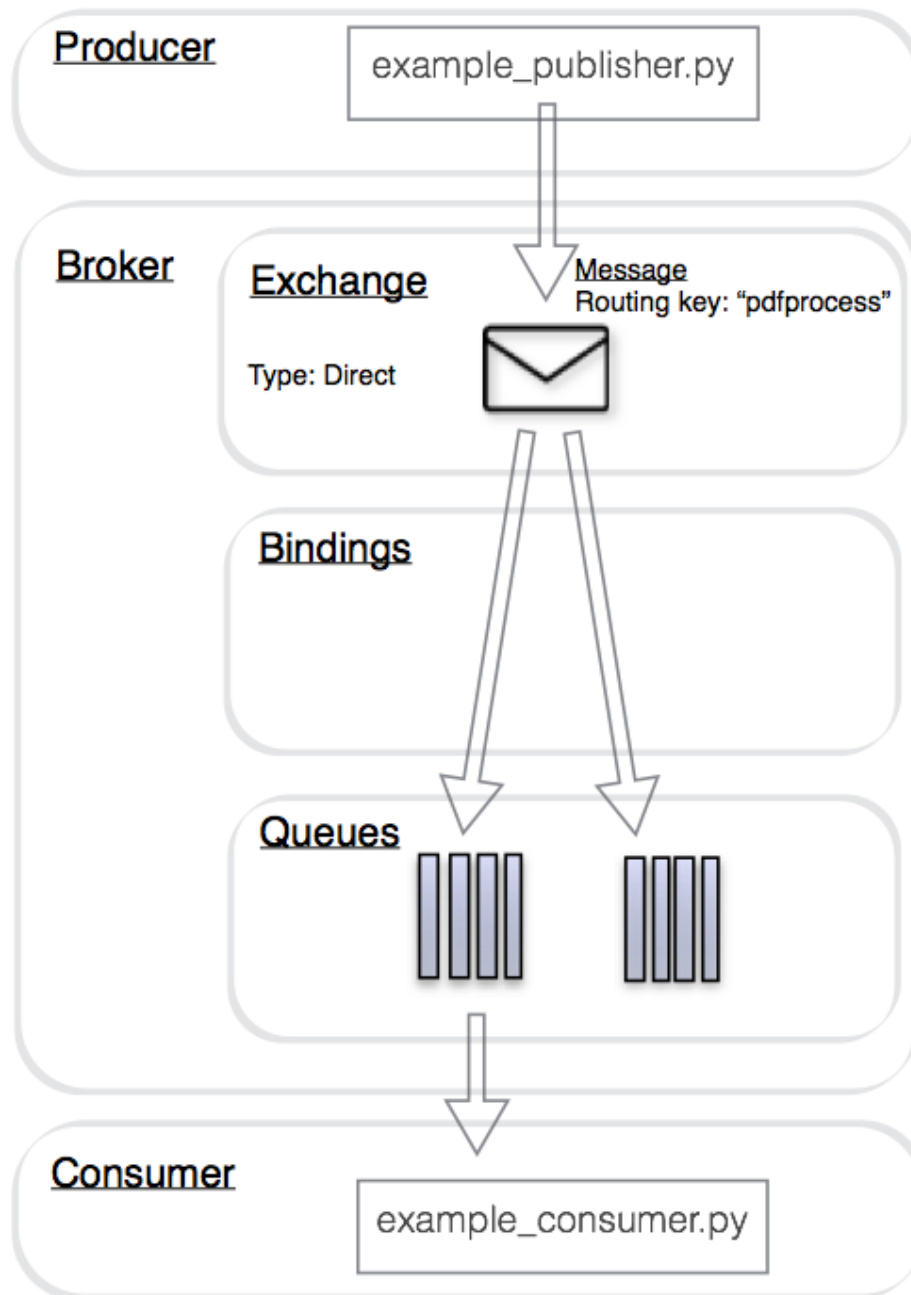


Figure: Python and RabbitMQ application architecture

Full code

```
# example_publisher.py
import pika, os, logging
logging.basicConfig()

# Parse CLOUDAMQP_URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
```

```

params.socket_timeout = 5

connection = pika.BlockingConnection(params) # Connect to CloudAMQP
channel = connection.channel() # start a channel
channel.queue_declare(queue='pdfprocess') # Declare a queue
# send a message

channel.basic_publish(exchange='', routing_key='pdfprocess', body='User information')
print " [x] Message sent to consumer"
connection.close()

```

```

# example_consumer.py
import pika, os, logging, time
logging.basicConfig()

def pdf_process_function(msg):
    print "PDF processing"
    time.sleep(5) # delays for 5 seconds
    print "PDF processing finished";
    return;

# Parse CLOUDAMQP_URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
params.socket_timeout = 5
connection = pika.BlockingConnection(params) # Connect to CloudAMQP
channel = connection.channel() # start a channel

# create a function which is called on incoming messages
def callback(ch, method, properties, body):
    pdf_process_function(body)

# set up subscription on the queue
channel.basic_consume(callback,
    queue='pdfprocess',
    no_ack=True)

# start consuming (blocks)
channel.start_consuming()
connection.close()

```

Tutorial source code - Publisher

```
# example_consumer.py
import pika, os, logging

# Parse CLOUDAMQP_URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
params.socket_timeout = 5
```

Load client library and set up configuration parameters. The `DEFAULT_SOCKET_TIMEOUT` is set to 0.25s, we would recommend to raise this parameter to about 5s to avoid connection timeout, `params.socket_timeout = 5` Other connection parameter options for Pika can be found here: [Connection Parameters](#).

Set up a connection

```
connection = pika.BlockingConnection(params) # Connect to CloudAMQP
```

`pika.BlockingConnection` establishes a connection with the RabbitMQ server.

Start a channel

```
channel = connection.channel()
```

`connection.channel` create a channel in the TCP connection.

Declare a queue

```
channel.queue_declare(queue='pdfprocess') # Declare a queue
```

`channel.queue_declare` creates a queue to which the message will be delivered. The queue will be given the name `pdfprocess`.

Publish a message

```
channel.basic_publish(exchange='', routing_key='pdfprocess', body='User information')
print " [x] Message sent to consumer"
```

`channel.basic_publish` publish the message to the channel with the given exchange, routing key and body. A default exchange, identify by the empty string ("") will be used. The default exchange means that messages are routed to the queue with the name specified by `routing_key` if it exists. (The default exchange is a direct exchange with no name).

Close the connection

```
connection.close()
```

The connection will be closed after the message has been published.

Consumer

Worker function

```
def pdf_process_function(msg):  
    print "PDF processing"  
    time.sleep(5) # delays for 5 seconds  
    print "PDF processing finished";  
    return;
```

`pdf_process_function` is a todo-function. It will sleep for 5 seconds to simulate the PDF-creation.

Function called for incoming messages

```
# create a function which is called on incoming messages  
def callback(ch, method, properties, body):  
    pdf_process_function(body)
```

The `callback` function will be called on every message received from the queue. The function will call a function that simulate the PDF-processing.

```
# set up subscription on the queue  
channel.basic_consume(callback,  
    queue='pdfprocess',  
    no_ack=True)
```

`basic_consume` binds messages to the consumer callback function.

```
channel.start_consuming() # start consuming (blocks)  
connection.close()
```


`start_consuming` starts to consume messages from the queue.

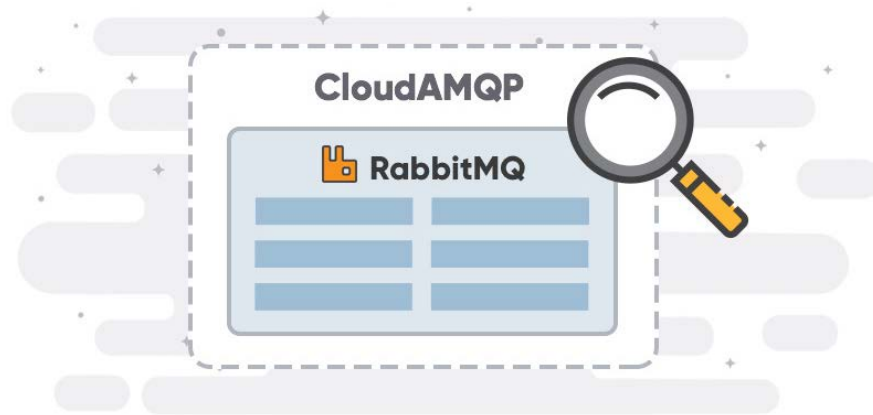
More information about Python and CloudAMQP can be found[here](#). You can also find information about celery in the online documentation[here](#).

CloudAMQP - Industry-Leading Experts at RabbitMQ

[Get a managed RabbitMQ server for FREE](#)

A microservice architecture built upon RabbitMQ

This chapter gives a simple overview of the automated process behind CloudAMQP. The polyglot workplace where microservices written in different languages communicate through RabbitMQ.



CloudAMQP never had a traditional monolithic setup. It's built from scratch on small, independent, manageable services that communicate with each other. These services are all highly decoupled and focused on their task. This chapter gives an overview and a deeper insight to the automated process behind CloudAMQP. It describes some of our microservices and how we are using RabbitMQ as message broker when communicating between services.

Background of CloudAMQP

A few years ago Carl Hörberg (CEO @CloudAMQP) saw the need of a hosted RabbitMQ solution. At the time he was working at a consultancy company where he was using RabbitMQ in combination with Heroku and AppHarbor. He was looking for a hosted RabbitMQ solution himself, but he could not find any. Short after he started CloudAMQP, which entered the market in 2012.

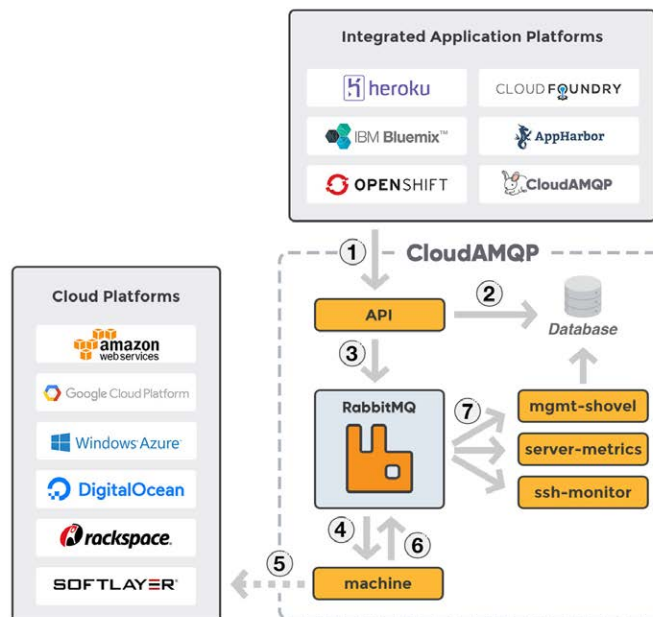
The automated process behind CloudAMQP

CloudAMQP is built upon multiple small microservices, where we use RabbitMQ as our messaging system. RabbitMQ is responsible for distributing

events to the services that listen for them. We have the option to send a message without having to know if another service is able to handle it immediately or not. Messages can wait until the responsible service is ready. A service publishing a message does not need know anything about the inner workings of the services that will process that message. We follow the pub-sub (publish-subscribe) pattern and we rely on retry upon failure.

When you create a CloudAMQP instance you get the option to choose a plan and how many nodes you would like to have. The cluster will behave a bit different depending on the cluster setup. You also get the option to create your instance in a dedicated VPC and select RabbitMQ version.

A dedicated RabbitMQ instance can be created via the CloudAMQP control panel, or by adding the CloudAMQP add-on from any of our integrated platforms, Heroku, Bluemix, AppHarbor, CloudFoundry, or OpenShift.



When a client creates a new dedicated instance, an HTTP request is sent from the reseller to a service called CloudAMQP-API (1). The HTTP request include all information specified by the client: plan, server name, data center, region, number of nodes etc. as you see in the picture above. CloudAMQP-API will handle the request, save information into a database (2), and finally send a `account.create-message` to one of our RabbitMQ-clusters (3).

Another service, called CloudAMQP-machine, is subscribing to `account.create`. CloudAMQP-machine takes the `account.create-message` of the queue and performs actions for the new account (4).

CloudAMQP-machine trigger a whole lot of scripts and processes. First it creates the new server(s) in the chosen datacenter via an HTTP request (5). We use different underlying instance types depending on data center, plan

and number of nodes. CloudAMQP-machine is responsible of all configuration of the server, setting up RabbitMQ, mirror nodes, handle clustering for RabbitMQ etc etc, all depending on number of nodes and chosen datacenter.

CloudAMQP-machine send a account.created-message back to the RabbitMQcluster when the cluster is created and configured. The message is sent on the topic exchange (6). The great thing with the topic exchange is that a whole lot of services can subscribe to the event. We have a few services that are listening to account.created-messages (7). Those services will all set up a connection to the new server. Here are three examples of services that are receiving the message and starts to work towards the new servers.

CloudAMQP-server-metrics: continuously gathering server metrics, such as CPU and disk space, from all servers.

CloudAMQP-mgmt-shovel: continuously ask the new cluster about RabbitMQ specific data, such as queue length, via the RabbitMQ management HTTP API.

CloudAMQP-SSH-monitor: is responsible for monitoring a number of processes that needs to be running on all servers.

CloudAMQP-server-metrics

We do have a lot of other services communicating with the services described above and with the new server(s) created for the client. As said before, CloudAMQP-server-metrics is collecting metrics (CPU/memory/disk data) for ALL running servers. The collected metric data from a server is sent to RabbitMQ queues defined for the specific server e.g. server.`.vmstat` and server.`.free` where hostname is the name of the server.

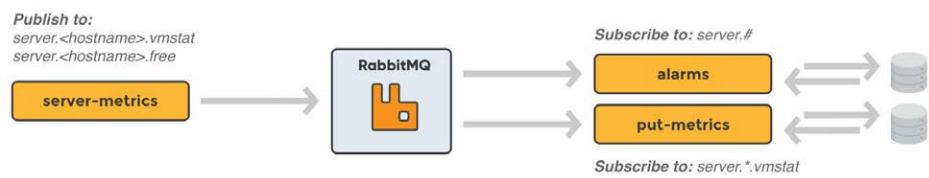
CloudAMQP-alarm

We have different services that are subscribing to this data. One of these services is called CloudAMQP-alarm. A user is able to enable/disable CPU and memory alarms for an instance. CloudAMQP-alarm check the server metrics subscribed from RabbitMQ against the alarm thresholds for the given server and it notifies the owner of the server if needed.

CloudAMQP-put-metrics

CloudAMQP is integrated with monitoring tools; DataDog, Logentries, AWS Cloudwatch, and Librato. A user who has an instance up and running have the option to enable these integrations. Our microservice CloudAMQP-put-metrics checks the server metrics subscribed from RabbitMQ, and is

responsible of sending metrics data to tools that the client has integrated with.



RabbitMQ Troubleshooting

High CPU usage during a period of time might lead to steal time and your machine might start throttle. If the cluster runs low of either RAM or disk space publishing will be halted until your consumers have processed enough messages. This chapter describes the most common errors and how to avoid them before they even happen.



We have during years seen different reasons for crashed or unresponsive CloudAMQP/RabbitMQ servers. Too many queued messages, high message rate during a long time or frequently opened and closed connections has been the most common reasons for high CPU usage. To protect against this we have added and made some tools available that helps address performance issues promptly and automatically, before they impact your business.

First of all - we recommend all users on dedicated plans to activate CPU and Memory alarms through the control panel. When CPU alarm is enabled you will receive a notification when you are using more than 80% of the available CPU. When Memory alarm is enabled you will receive a notification when you are using more than 90% of the available memory. [Read more about our alarms in the monitoring pages.](#)

Troubleshooting RabbitMQ

General tips and tools to keep in mind once you start troubleshooting your RabbitMQ server.

Server Metrics

Server Metrics will show performance metrics from your server. CloudAMQP shows CPU Usage Monitoring and Memory Usage Monitoring. Do you see any CPU or memory spikes? Do you notice some changes in the graph during a specific time? Did you do any changes in your code around that time?

RabbitMQ Management interface

If you are able to open the RabbitMQ mgmt interface - do that and check that everything is looking normal; Everything from number of queues to number of channels to messages in your queues.

Event stream

The event stream allows you to see the latest 1000 events from your RabbitMQ cluster. New events will be added to the collection in real time. The event stream can be useful when you need to gain insight into what is happening in your cluster. It is particularly good to debug if you are running into high CPU-usage, for instance if you have rapid opening or closing of connections or channels or a setup for shovel that is not working, etcetera.

RabbitMQ Log stream

RabbitMQ Log Stream show a live log from RabbitMQ. Do you see new unknown errors in your log?

Most common errors/mistakes

1. Too many queued messages

RabbitMQ's queues are fastest when they're empty. If the queue receives messages at a faster rate than the consumers can handle, the queue will start growing and eventually it will get slower. As the queue grows, it will require more memory. A high RAM usage could indicate that the number of queued messages rapidly went up. When this happens RabbitMQ will start flushing messages to disk in order to free up RAM and when that happens queueing speeds will deteriorate. The CPU cost per message will be much higher then when the queue was empty (the messages now have to be queued up). CPU I/O Wait will probably get high, which indicates the percentage of time the CPU have to wait on the disk.

For optimal performance queues should always stay around 0.

You should also check if your consumers are connected - consumers alarm can be activated from the console. The alarm will be triggered to send notifications when the number of consumers for a queue is less than or equal to a given number of consumers.

If you are running RabbitMQ version 3.4 or lower - upgrade - with RabbitMQ 3.5 and newer you can basically queue until the disk runs out unlike in 3.4 where the message index had to be kept in memory at all time. You can upgrade the version from the console (next to where you reboot/restart the instance).

We recommend you to enable the queue length alarm from the console or, if possible, to set a max-length or a max-ttl on messages in your queues.

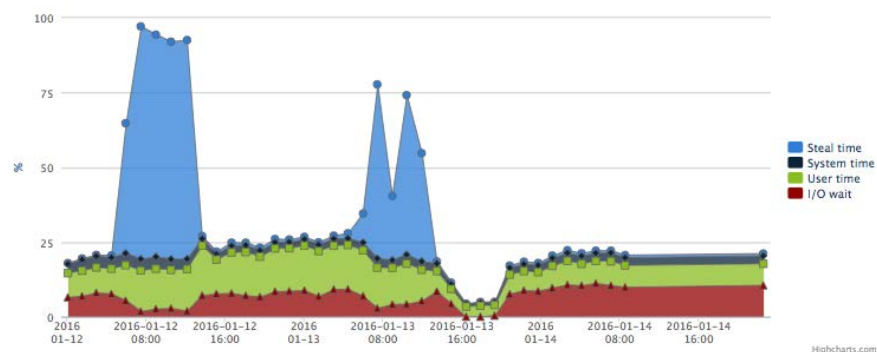
If you already have too many messages in the queue - you need to start consuming more messages or get your queues drained. If you are able to turn off your publishers then do that until you have managed to consume the messages from the queue.

Too many unacknowledged messages.

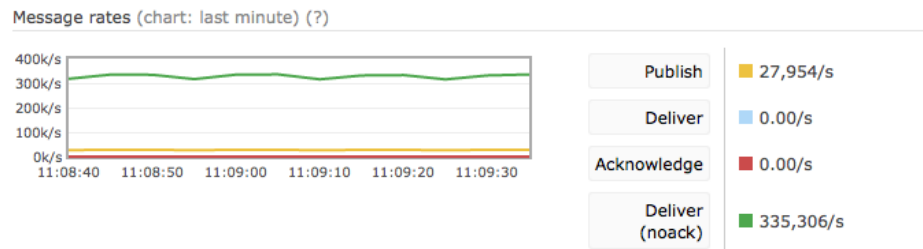
All unacknowledged messages have to reside in RAM on the servers. If you have too many unacknowledged messages you will run out of RAM. An efficient way to limit unacknowledged messages is to limit how many messages your clients prefetch. Read more about consumer prefetch [here](#).

2. Too high message throughput

CPU User time indicates the percentage of time your program spends executing instructions in the CPU - in this case, the time the CPU spent running RabbitMQ. If CPU User time is high, it could be due to high message throughput.



Dedicated plans in CloudAMQP are not artificially limited in any way, the maximum performance is determined by the underlying instance type. Every plan has a given Max msgs/s, that is the approximate burst speed we have measured during benchmark testing. Burst speed means that you temporarily can enjoy the highest speed but after a certain amount of time or once the quota has been exceeded, you might automatically be knocked down to a slower speed.



The benchmark tests are done with RabbitMQ PerfTest tool with one queue, one publisher and one customer. Durable queues/exchanges has been used and transient messages with default message size has been used. There are a lot of other factors too that plays in, like type of routing, if you ack or auto-ack, datacenter, if you publish with confirmation or not etc. If you for example are publishing large persistent messages, it will result in a lot of **I/O wait time** for the CPU. Read more about Load testing and performance measurements in RabbitMQ [here](#).

Decrease the average load or upgrade/migrate

When you have too high message throughput you should either try to decrease the average load, or upgrade/migrate to a larger underlying instance type. If you want to rescale your cluster, go to the CloudAMQP Control Panel, and choose edit for the instance you want to reconfigure. Here you have the ability to add or remove nodes, and change plan all together. More about migration can be found [here](#).

Python Celery

Celery sends a lot of "unnecessary" messages, due to their gossip, mingle and events. You can disable it by adding `--without-gossip --without-mingle --without-heartbeat` to the Celery worker command line arguments and add `CELERY_SEND_EVENTS = False` to your settings.py. Take a look at our [Celery documentation](#) for up to date information about Celery Settings.

3. Too many queues

RabbitMQ can handle a lot of queues, but each queue will of course require some resources. CloudAMQP plans are not artificially limited in any way, the maximum performance is determined by the underlying instance type and so are the number of queues. Too many queues will also use a lot of resources in the stats/mgmt db. If the queues suddenly starts to pile up, you might have a queue leak. If you can't find the leakage you can add a queue-ttl policy. The cluster will clean up it self for "dead-queues" - old queues will automatically be removed. You can read more about TTL [here](#).

Python Celery

If you use Python Celery, make sure that you either consume the results or disable the result backend (set `CELERY_RESULT_BACKEND = None`).

Connections and channels

Each connection uses at least one hundred kB, even more if TLS is used.

4. Frequently opening and closing connections

If **CPU System time** is high, you should check how you are handling your connections.

RabbitMQ connections are a lot more resource heavy than channels. Connections should be long lived, and channels can be opened and closed more frequently. Best practise is to reuse connections and multiplex a connection between threads with channels - in other words, if you have as many connections as channels you might need to take a look at your architecture. You should ideally only have one connection per process, and then use a channel per thread in your application.

If you really need many connections - the CloudAMQP team can configure the TCP buffer sizes to accommodate more connections for you. Send us an email when you have created your instance and we will configure the changes.

Number of channels per connection

It is hard to give a general best number of channels per connection - it all depends on your application. Generally the number of channels should be larger, but not much larger than number of connections. On our shared clusters (little lemur and tough tiger) we have a limit of 200 channels per connections.

6. AWS Big Bunny Users with high steal time

If you are on plan Big Bunny on AWS you are on [T2-instances](#). If you, for example, have too high message throughput during too long time you will run

out of [CPU credits](#). When that happens, steal time is added and your instance starts throttle (your instance will be running slower than it should). If you notice high steal time you need to upgrade to a larger instance or identify the reason for your CPU usage.

7. Memory spikes on RabbitMQ 3.6.0

If you are running RabbitMQ 3.6.0, we recommend you to upgrade to a new version of RabbitMQ. RabbitMQ 3.6.0 is not a good version of RabbitMQ - it has a lot of issues that are fixed in new versions. It's recommended to upgrade via CloudAMQP Control Panel, just click upgrade and select a version higher than 3.6.6. in the node tab for your instance.

More Resources

Here we've collected some of our favorite resources to learn more about RabbitMQ and CloudAMQP.

- **RabbitMQ in Action - Distributed Messaging for Everyone**, by Alvaro Videla and Jason J.Q. Williams, is a great book on the topic of RabbitMQ.
- **Instant RabbitMQ Messaging Application Development How-to**, by Andrew Keig is another good read.
- The **CloudAMQP blog** continuously publishes material about RabbitMQ: [blog](#)