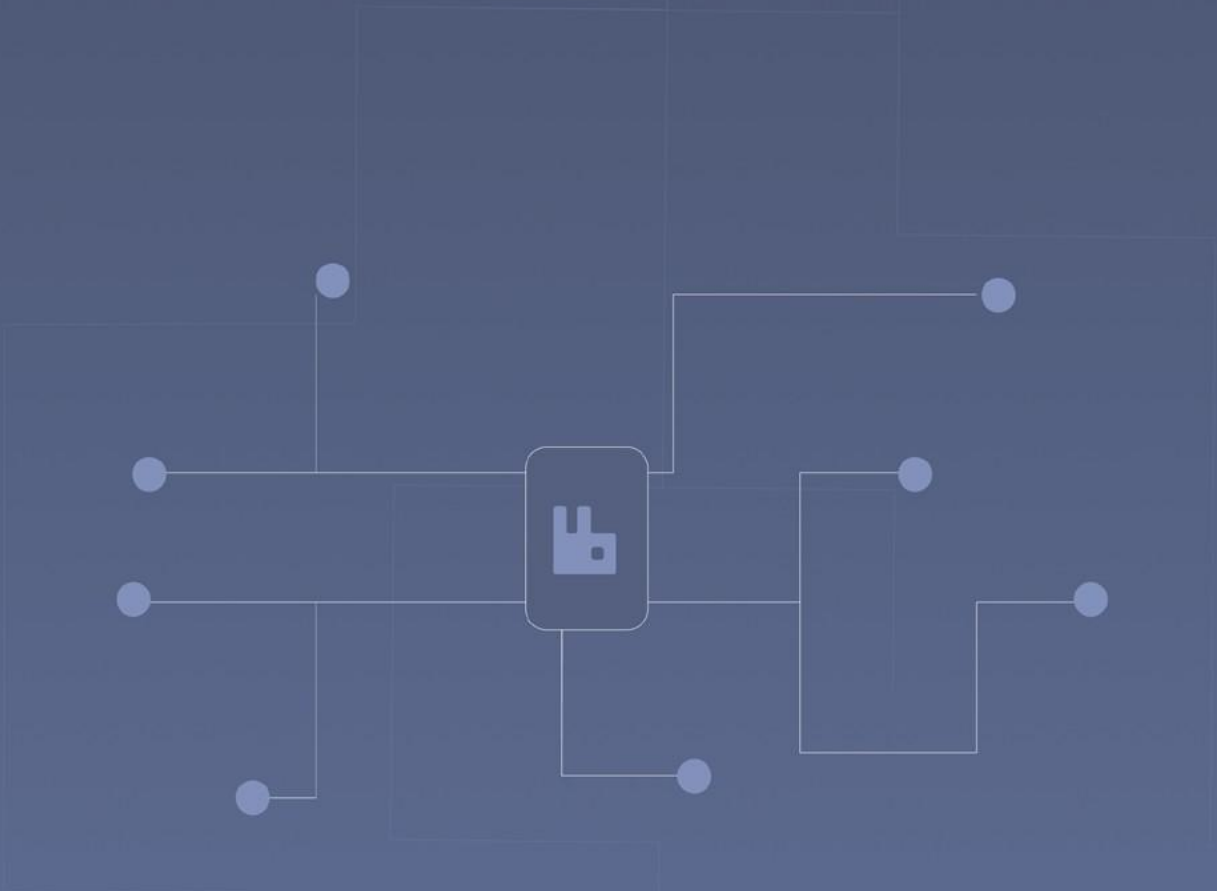


INTRODUCTION TO RABBITMQ

A MICROSERVICE ARCHITECTURE

Modules with various functionalities operate together to form a complete software application via properly articulated operations and interfaces.



Welcome to the wonderful world of message queueing! Many modern cloud architectures are a collection of loosely coupled microservices that communicate via a message queue. This microservice architecture is beneficial in that all the different parts of the system are easier to develop, deploy and maintain.

One of the advantages of a message queue is that it makes your data temporarily persistent, reducing the risk of errors that may occur when different parts of the system are offline. If one part of the system is unreachable, the other part continues to interact with the queue. This part of the book features basic information on microservices and message queueing, and the benefits of RabbitMQ.

MICROSERVICES AND RABBITMQ

Managing a complex, unified enterprise application can be a lot more difficult than it seems. Before making even a small change, hours of testing and analysis are required to examine the possible impacts your change could have on the overall system. New developers must spend days getting familiar with the system before they are considered ready to write a line of code that won't break anything. Microservice architecture makes everything a lot less complicated. In this chapter, you'll learn the benefits of a microservice architecture.

In a microservice architecture, components are decoupled from each other. These elements often offer various functionalities that operate together to form a complete software application via properly articulated operations and interfaces. Unlike a monolithic application, where all functionality is present within a single unit, a microservice application divides different features across components.

Microservices offer several advantages, one of them being effortless scalability. With a microservices architecture, you can effortlessly handle increased workloads and adapt to changing demands without any hassle. Additionally, this architecture enables you to deliver updates more frequently, allowing your products or services to stay up-to-date and meet the evolving needs of your customers.

BENEFITS OF A MICROSERVICE ARCHITECTURE

- **Easier development and maintenance**

Imagine building a huge, bulky billing application that will involve authentication, authorization, financial transactions and reporting. Dividing the application across multiple services (one for each functionality) separates the responsibilities and gives developers the freedom to write code for a specific service in any chosen language. Additionally, it also makes it easier to maintain written code and make changes to the system. For example, updating an authentication scheme will only require adding code to the authentication module and testing without having to worry about disrupting any other functionalities of the application.

- **Fault isolation**

Another obvious advantage offered by a microservice architecture is the ability to isolate the fault to a single module. For example, if a reporting service is down, authentication and billing services will still be running, ensuring that customers can perform important transactions even when they are not able to view reports.

- **Increased speed and productivity**

Microservice architecture is fundamentally about decoupling functions into manageable modules that are easy to maintain. Different developers can work on different modules at the same time. In addition to the development cycle, the testing

phase is also sped up by the use of microservices, as each microservice can be tested independently to determine the readiness of the overall system.

- **Improved scalability**

Microservices also allow effortless system scaling whenever required. Adding new functionality to just one service can be done without changing other services. Along the same lines, resource-intensive services can be deployed across multiple servers by using microservices.

- **Easy to understand**

One of the advantages offered by a microservice architecture is the ease of understandability. Because each service represents a single function, learning the relevant details becomes easier and faster. For example, a consultant who works on financial transactions service does not have to understand the whole system to perform maintenance or enhancements to their part of the system.

THE ROLE OF A MESSAGE QUEUE IN A MICROSERVICE ARCHITECTURE

Regarding communication in microservices, two popular approaches grace the stage: synchronous HTTPS and asynchronous, lightweight messaging. HTTP-based microservices might appear more straightforward to implement due to their familiar request-response nature. However, in reality, they have some major limitations. The tight coupling in HTTP-based microservices leads to issues such as fault isolation, inflexibility in introducing changes, and long response times. However, by adopting message queues, we can achieve a more decoupled architecture that enhances fault isolation, provides flexibility in introducing new changes, and reduces response times.

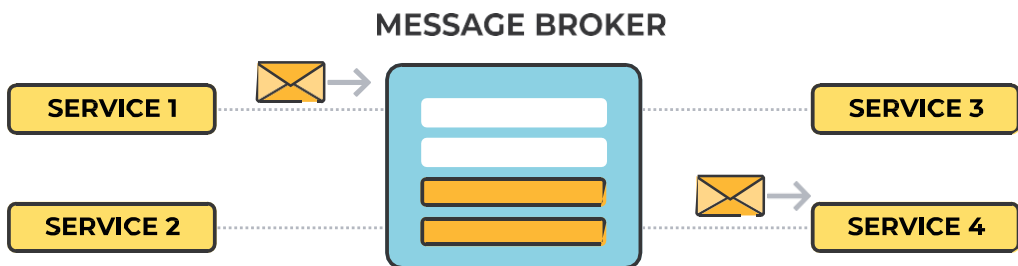


Figure 1 - Microservices exchanging messages via a message broker.

Think of a message queueing software, called a message broker, as a delivery person who takes mail from a sender and delivers it to the correct destination. In a microservice architecture, cross-dependencies typically mean no single service can perform without getting help from other services. This is why systems must have a mechanism in place to allow services to keep in touch with each other with no blocked responses. Message queuing fulfills this purpose by providing a means for services to push messages to a queue asynchronously and ensure they are delivered to the correct destination. To implement message queueing, a message broker is required.

WHAT IS RABBITMQ?

Message queueing is a way of exchanging data between processes, applications, and servers. With tens of thousands of users, RabbitMQ is one of the most popular open-source message brokers in the world. This chapter gives a brief understanding of message queueing and defines essential RabbitMQ concepts. The chapter also explains the steps to go through when setting up connections including how to publish as well as consume messages from a queue.



Figure 2 - Messages sent from a sender to a receiver.

RabbitMQ is a message broker. It is software used to define queues, connect applications, and accept messages. Message queues enable asynchronous communication, allowing other applications (endpoints) that are producing and consuming messages to interact with the queue instead of communicating directly with each other.

A message can include any type of information. For example, a message could contain information about a process or job that should start on another application, possibly even on another server, or it might be a simple text message.

The message broker stores the messages until a receiving application connects and consumes a message from the queue. The receiving application then processes the message appropriately. A message producer adds messages to a queue without having to wait for them to be processed.

RABBITMQ EXAMPLE

RabbitMQ acts as a middleman for various services. It can be used to reduce loads and delivery times on web application servers. For instance, offloading a time-consuming task to a third-party service with no other job.

This section will explore a case study of a PDF generator web application. The application enables users to upload information to a website, where it is processed to generate a PDF. Subsequently, the PDF is emailed back to the user.

When the user enters their information into the web interface, the web application puts a "PDF processing" job into a message and includes information such as name and email. The message is placed in a queue defined in RabbitMQ

The underlying architecture of a message broker is simple; client applications called producers create messages and deliver them to the broker. Other applications, known as consumers, connect to the broker, subscribe to messages from the broker, and process them. The software interacting with the broker can be a producer, a consumer, or both. Messages placed in the broker are stored until the consumer retrieves them.

The message queue safely holds the messages in case the PDF processing application crashes or if many requests are coming in simultaneously.



Figure 3 - A sketch of the RabbitMQ workflow.

WHY AND WHEN TO USE RABBITMQ

Message queueing allows web servers to respond to requests in their own time instead of being forced to perform resource-heavy procedures immediately. Message queueing is also useful for distributing a message to multiple recipients for consumption or balancing the load between workers.

The consumer application removes a message from the queue and processes the PDF while the producer pushes new messages to the queue. The consumer can be on the same or an entirely different server than the publisher, it makes no difference. Requests can be created in one programming language and handled in another programming language, as the two applications only communicate through the messages they are sending to each other. The two services have what is known as 'low coupling' between the sender and the receiver.

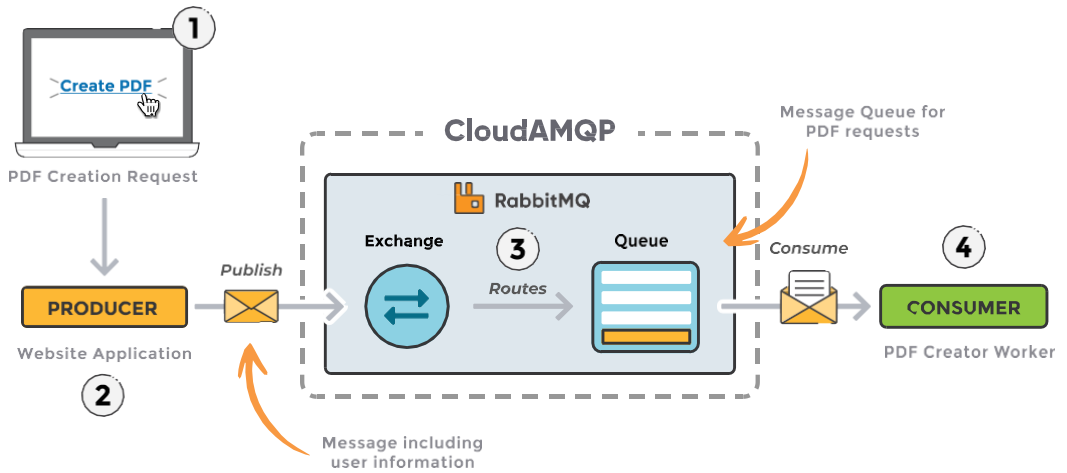


Figure 4 - Application architecture example with RabbitMQ.

Example

1. The customer sends a request to create a PDF to the web application.
2. The web application (the producer) sends a message to RabbitMQ that includes data from the request, such as name and email.
3. An exchange accepts the messages from a producer application and routes them to the correct message queues.
4. The PDF processor (the consumer) receives the job message from the queue and starts processing the PDF.

EXCHANGES

We just encountered a new concept: exchanges. Understanding exchanges is vital as they serve as intermediaries between producers and queues, enabling efficient message routing.

Rather than directly publishing messages to a queue, producers send them to an exchange. The exchange uses bindings and routing keys to determine the correct destination for the message. Bindings link the queue to an exchange. Routing keys act as an address for the message.

Message Flow in RabbitMQ

1. The producer publishes a message to an exchange. When creating an exchange, its type must be specified. The different types of exchanges are explained in detail later on in this book.
2. The exchange receives the message and is now responsible for the routing of the message. The exchange looks at different message attributes and keys depending on the exchange type.
3. In this case, we see two bindings to two different queues from the exchange. The exchange routes the message to the correct queue, depending on these attributes.
4. The messages stay in the queue until the consumer processes them.
5. The broker removes the message from the queue once the consumer confirms that the message has been received and processed.

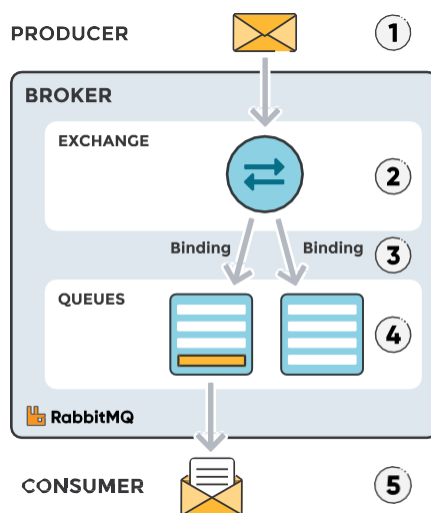


Figure 5 - Illustration of the message flow in RabbitMQ

Types of Exchanges

This is a short description of exchanges, an in-depth understanding of the different exchange types, binding keys, routing keys and how/when they should be used can be found in the chapter: Exchanges, routing keys and bindings.

- **Direct** - A direct exchange delivers messages to queues based on a message routing key. In a direct exchange, messages are routed to the queue with the binding key that exactly matches the routing key of the message.
- **Topic** - The topic exchange performs a wildcard match between the routing key and the routing pattern specified in the binding.
- **Fanout** - A fanout exchange routes messages to all of the queues with a binding tied to the exchange.
- **Headers** - A header exchange uses the message header attributes for routing purposes.

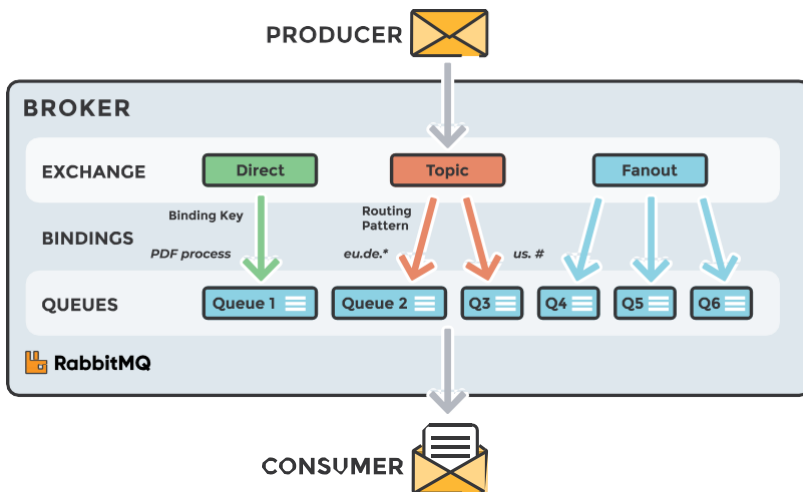


Figure 6 - Three different exchanges: direct, topic, and fanout.

RABBITMQ AND SERVER CONCEPTS

Below are some important concepts that are helpful to know before we dig deeper into RabbitMQ. The default virtual host, the default user, and the default permissions are used in the examples that follow.

- **Producer** - Application that sends the messages.
- **Consumer** - Application that receives the messages.
- **Queue** - The place where messages are stored until they are consumed by the consumer, or in other ways removed or dropped from the queue.
- **Message** - Data sent from producer to consumer through RabbitMQ.
- **Connection** - A link between the application (producer and consumer) and the broker, that performs underlying networking tasks including initial authentication, IP resolution, and networking.
- **Channel** - Connections can multiplex over a single TCP connection, meaning that an application can open "lightweight connections" on a single connection. This lightweight connection is called a channel. Each connection can maintain a set of underlying channels.
- **Exchange** - Theoretically, the exchange is the first entry point for a message entering the message broker. It receives messages from producers and pushes them to queues depending on rules defined by the exchange type. A queue needs to be bound to at least one exchange to be able to receive messages.
- **Binding** - An association, or relationship between a queue and an exchange. It describes which queue is interested in messages from a given exchange.
- **Routing Key** - The key that the exchange looks at to decide how to route the message to queues. Think of the routing key as the destination address of a message.
- **AMQP** - Advanced Message Queueing Protocol is the primary protocol used by RabbitMQ for messaging.
- **Users** - Connecting to RabbitMQ with a given username and password, that is assigned permissions such as rights to read, write and configure. Users can also have specific permissions to a specific virtual host.
- **Vhost** - Virtual host or vhost segregates applications that are using the same RabbitMQ instance. Different users can have different access privileges to different vhosts and queues, and exchanges can be created so that they only exist in one vhost.
- **Acknowledgments and Confirms** - Acknowledgments can be used in both directions, indicating that messages have been received or acted upon. For instance, a consumer can inform the broker that it has received or processed a message, and the broker can confirm to the producer that a message has been received.

SETTING UP A RABBITMQ INSTANCE

Install Erlang

Before starting, you will need to install Erlang on your server. First, install the required dependencies with the following command:

```
$sudo apt-get install software-properties-common curl gnupg2 apt-transport-https
```

Next, import the Erlang GPG key using the following command:

```
$wget -O- https://packages.erlang-solutions.com/ubuntu/erlang_solutions.asc | apt-key add -
```

Next, add the Erlang repository with the following command:

```
$ echo "deb https://packages.erlang-solutions.com/ubuntu focal contrib" | tee /etc/apt/sources.list.d/erlang.list
```

Finally, update the repository and install Erlang with the following command:

```
$sudo apt-get update -y  
$sudo apt-get install erlang -y
```

To verify the Erlang installation, run the following command:

```
$erl -v
```

You will get the Erlang shell in the following output:

```
Erlang/OTP 24 [erts-12.1.5] [64-bit] [smp:1:1] [ds:1:1:10] [async-threads:1] [jit]
```

```
Eshell V12.1.5 (abort with ^G)
```

```
1>
```

Press **CTRL+G** then type **q** to exit from the Erlang shell.

Install RabbitMQ on Ubuntu 20.04

By default, RabbitMQ is not available in the Ubuntu 20.04 default repository. So you will need to install the RabbitMQ repository to your system. Run the following command to install the RabbitMQ repository:

```
$curl -s https://packagecloud.io/install/repositories/rabbitmq/rabbitmq-server/script.deb.sh | bash
```

Now, update the repository and install the RabbitMQ with the following command:

```
$sudo apt-get update -y
$sudo apt-get install rabbitmq-server -y
```

You can check the status of the RabbitMQ using the following command:

```
$sudo systemctl status rabbitmq-server
```

You will get the following output:

```
● rabbitmq-server.service - RabbitMQ broker
   Loaded: loaded (/lib/systemd/system/rabbitmq-server.service; enabled; vendor
  preset: enabled)
   Active: active (running) since Thu 2021-12-02 09:17:12 UTC; 12s ago
 Main PID: 19117 (beam.smp)
    Tasks: 22 (limit: 2353)
   Memory: 87.0M
    CGroup: /system.slice/rabbitmq-server.service
            └─19117 /usr/lib/erlang/erts-12.1.5/bin/beam.smp -W w -MBas ageffcbf -
MHas ageffcbf -MBlmbcs 512 -Mhlmbcs 512 -MMmcs 30 -P 10485>
            └─19128 erl_child_setup 32768
            └─19168 /usr/lib/erlang/erts-12.1.5/bin/epmd -daemon
            └─19188 inet_gethost 4
            └─19189 inet_gethost 4
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: Doc guides:
```

```
https://rabbitmq.com/documentation.html
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: Support:
```

```
https://rabbitmq.com/contact.html
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: Tutorials:
```

```
https://rabbitmq.com/getstarted.html
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: Monitoring:
```

```
https://rabbitmq.com/monitoring.html
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: Logs:
```

```
/var/log/rabbitmq/rabbit@server.log
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]:
```

```
/var/log/rabbitmq/rabbit@server_upgrade.log
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: <stdout>
```

```
Dec 02 09:17:09 server rabbitmq-server[19117]: Config file(s): (none)
```

```
Dec 02 09:17:12 server rabbitmq-server[19117]: Starting broker... completed with 0
plugins.
```

```
Dec 02 09:17:12 server systemd[1]: Started RabbitMQ broker.
```

Create an Administrative User

By default, RabbitMQ can be connected without any username or password. So it is a good idea to set an admin user and password to connect to the RabbitMQ.

You can create an administrator user, set a password, set an administrator tag and set permission using the following commands:

```
$rabbitmqctl add_user admin securepassword
$rabbitmqctl set_user_tags admin administrator
$rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

You can list all users permission using the following command:

```
$rabbitmqctl list_permissions -p /
```

You will get the following output:

```
Listing permissions for vhost "/" ...
user  configure      write  read
guest  .*      .*      .*
admin  .*      .*      .*
```

Enable RabbitMQ Web UI

RabbitMQ provides a web based interface to manage the RabbitMQ instance. To access the RabbitMQ web interface, you will need to enable it first.

Run the following command to enable the RabbitMQ web interface:

```
$rabbitmq-plugins enable rabbitmq_management
```

You will get the following output:

```
Enabling plugins on node rabbit@server:
rabbitmq_management
```

The following plugins have been configured:

```
rabbitmq_management
rabbitmq_management_agent
rabbitmq_web_dispatch
```

Applying plugin configuration to rabbit@server...

The following plugins have been enabled:

```
rabbitmq_management
rabbitmq_management_agent
rabbitmq_web_dispatch
```

started 3 plugins.

By default, the RabbitMQ web interface listens on port **15672**. You can check it using the following command:

```
$ss -tunelp | grep 15672
```

You will get the following output:

```
tcp LISTEN 0 1024 0.0.0.0:15672 0.0.0.0:*  
users:([("beam.smp",pid=19117,fd=35)) uid:118 ino:57672 sk:19 <->
```

Access RabbitMQ Web Interface

Now, open your web browser and access the RabbitMQ web interface using the URL **http://your-server-ip:15672**. You should see the RabbitMQ login page:



Username: *

Password: *

Provide your admin username, password and click on the **Login** button. You will be redirected to the RabbitMQ dashboard:

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+/-
rabbit@server	42 32768 available	0 29401 available	357 1048576 available	1.19 MB 795 MB high watermark	43 GiB 44 MB low watermark	3m 13s	bas/c dbr l rsc	This node All nodes	

> Churn statistics

> Ports and contexts

> Export definitions

> Import definitions

The Management Interface

The management interface allows users to manage, create, delete, and list queues. It monitors queue length, is the place to go to check the message rate, change or add user permissions and much more. Detailed information about the management interface is provided in the *The Management Interface chapter*.

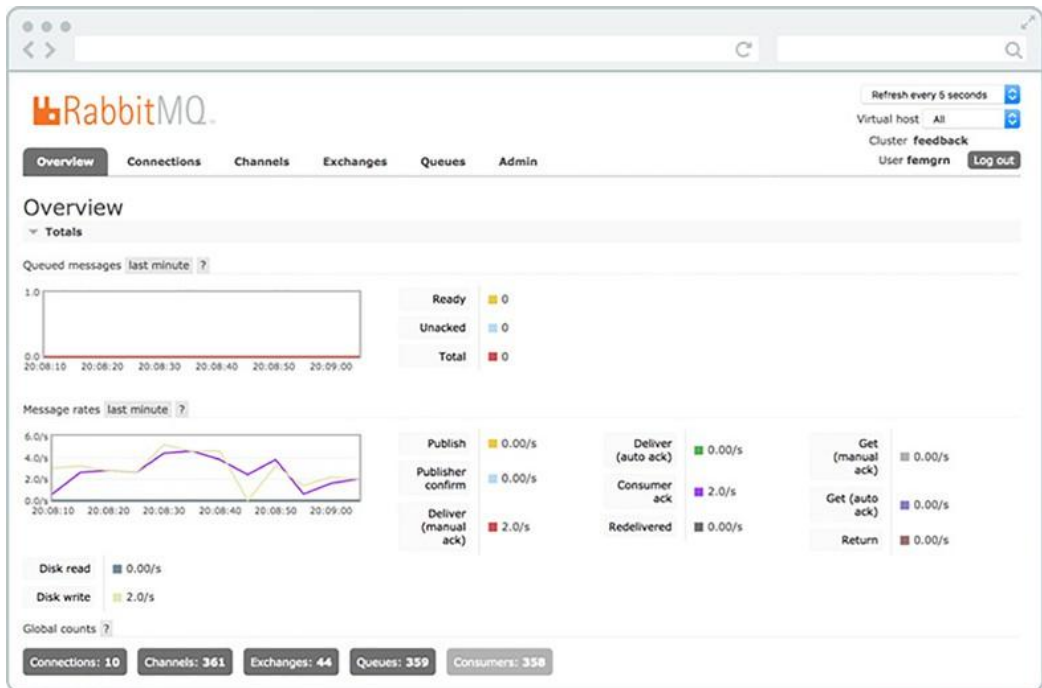


Figure 9 - The overview window in the RabbitMQ management interface.

Publish and Consume Messages

RabbitMQ speaks the AMQP 0.9.1 protocol by default, so to be able to communicate with RabbitMQ, a client library that understands the same protocol should be used. A RabbitMQ client library (sometimes called helper library) abstracts the complexity of the AMQP protocol into simple methods which, in this case, is communicating with RabbitMQ. These methods should be used when connecting to the RabbitMQ broker with parameters including, for example, hostname, port number, or when declaring a queue or an exchange. Libraries are available for all major programming languages.

The following steps represent the standard flow when setting up a connection and a channel in RabbitMQ via the client library, and how messages are published and consumed.

1. Set up a connection object. This is where the username, password, connection URL, port, etc., will be specified. A TCP connection will be set up between the application and RabbitMQ.
2. Open a channel. You are now ready 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. Set up exchanges and bind a queue to an exchange.
5. Publish a message to an exchange and consume a message from the queue.
6. Close the channel and the connection.

What are exchanges, bindings, and routing keys? In what way are exchanges and queues associated with each other? When should they be used and how? This chapter explains the different types of exchanges in RabbitMQ and gives examples of when to use them.

As mentioned in the previous chapter, messages are not published directly to a queue. Instead, the producer sends messages to an exchange. Exchanges are message routing agents, living in a virtual host (vhost) within RabbitMQ. Exchanges accept messages from the producer application and route them to message queues with the help of header attributes, bindings, and routing keys.

A binding is a *link* configured to make a connection between a queue and an exchange. The routing key is a message attribute. The exchange might look at the routing key, depending on exchange type, when deciding on how to route the message to the correct queue.

Exchanges, connections, and queues can be configured to include properties such as durable, temporary, and auto-delete. Durable exchanges survive server restarts and last until they are deleted. Temporary exchanges exist until RabbitMQ is shut down. Auto-deleted exchanges are removed once the last bound object is unbound from the exchange.

In RabbitMQ, four different types of exchanges route the message differently using different parameters and bindings setups. Clients can create their own unique exchanges or use the predefined default exchanges.

DIRECT EXCHANGE

A direct exchange delivers messages to queues based on a routing key. The routing key is a message attribute added to the message by the producer. Think of the routing key as an *address* that the exchange uses to decide on how to route the message. **A message goes to the queue(s) that exactly matches the binding key to the routing key of the message.** The direct exchange type is useful to distinguish messages published to the same exchange using a simple string identifier.

Queue A (`create_pdf_queue`) in Figure 10 is bound to a direct exchange (`pdf_events`) with the binding key (`pdf_create`). When a new message with the routing key (`pdf_create`) arrives at the direct exchange, the exchange routes it to the queue where the `binding_key = routing_key`, in this case to queue A (`create_pdf_queue`).

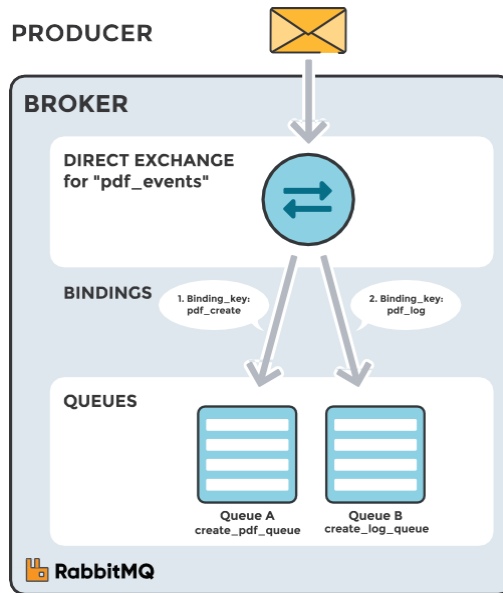


Figure 10 - A message is directed to the queue where the binding key is an exact match of the message's routing key.

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

A message with the routing key pdf_log is sent to the exchange pdf_events (Figure 10). The message is routed to create_log_queue because the routing key (pdf_log) matches the binding key (pdf_log).

Note: If the message routing key does not match any binding key, the message is discarded or forwarded to an alternate exchange if specified.

Default exchange

The default exchange is a pre-declared direct exchange with no name, usually referred to with the empty string, "". When using the default exchange, the message is 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 that matches the queue name.

TOPIC EXCHANGE

Topic exchanges route messages to a queue based on a wildcard match between the routing key and the routing pattern, which is specified by the queue binding. Messages can be routed to one or many queues depending on this wildcard match.

The routing key must be a list of words delimited by a period (.). Examples include `agreements.us` or `agreements.eu.stockholm`, which in this case identifies agreements that are set up for a company with offices in 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.*` only match routing keys where the first word is `agreements` and the fourth word is `"b"`). A pound symbol ("#") indicates a match on zero or more words (e.g., a routing pattern of `agreements.eu.berlin.#` matches any routing keys beginning with `agreements.eu.berlin`).

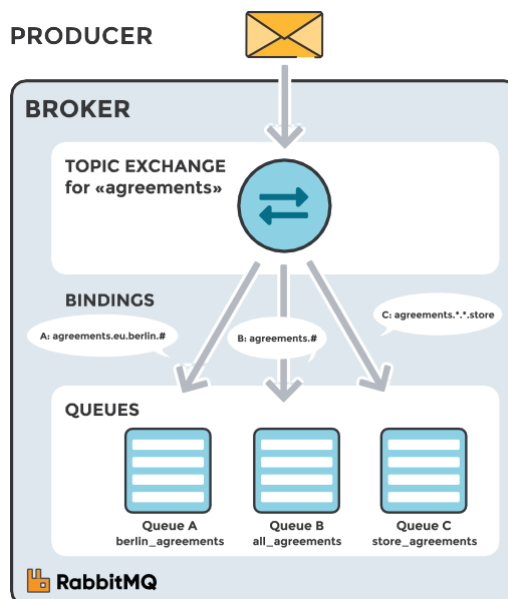


Figure 11 - Messages are routed to one or many queues based on a match between a message routing key and the routing patterns.

The consumers indicate which topics they are interested in (like subscribing to a feed of 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 are routed to the queue and stay there until the consumer handles the message.

Scenario 1

Figure 11 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 matches: *agreements.eu.berlin* and *agreements.eu.berlin.store*

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 matches: *agreements.eu.berlin* and *agreements.us*

Scenario 3

Consumer C is interested in all agreements for European stores

- Exchange: agreements
- Queue C: store_agreements

Routing pattern between exchange (agreements) and Queue C (store_agreements): *agreements.eu.*.store*

Example of message routing keys that will match: *agreements.eu.berlin.store* and *agreements.eu.stockholm.store*

A message with routing key *agreements.eu.berlin* is sent to the exchange agreements. The message is routed to the queue berlin_agreements because of 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` since the routing key (*agreements.eu.berlin*) also matches the routing pattern *agreements.#*.

FANOUT EXCHANGE

Fanout exchanges copy and route a received message to all queues that are bound to it regardless of routing keys or pattern matching, unlike direct and topic exchanges. If routing keys are provided, they will 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, like in distributed systems designed to broadcast various state and configuration updates.

Figure 12 shows an example where a message received by the exchange is copied and routed to all three queues 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.

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)

A message is sent to the exchange `sport_news` (Figure 12). The message is routed to all queues (Queue A, Queue B, Queue C) because all queues are bound to the exchange, and any provided routing keys are ignored.

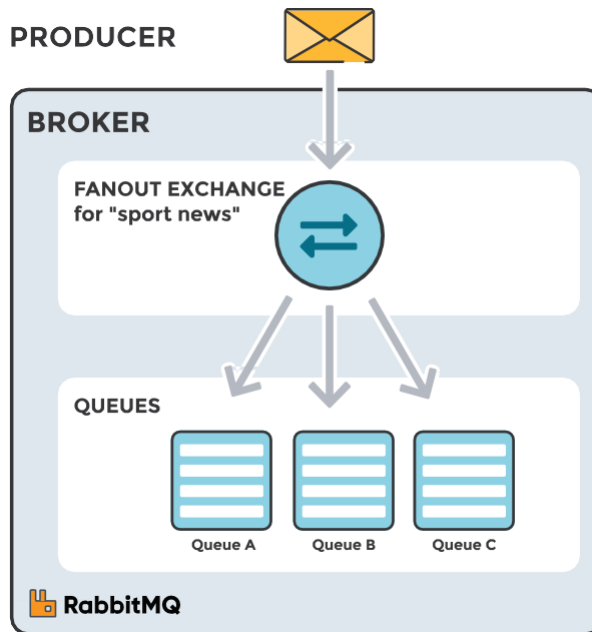


Figure 12 - Fanout Exchange: The received message is routed to all queues that are bound to the exchange.

HEADERS EXCHANGE

A headers exchange routes messages based on arguments contained in headers and optional values. Headers exchanges are very similar to topic exchanges, but route messages based on header values instead of routing keys. A message matches if the value of the header equals the value specified upon binding.

A special argument named "x-match", added in the binding between the exchange and the queue, specifies if all headers must match or just one. Either any common header between the message and the binding counts as a match or all the headers referenced in the binding need to be present in the message for it to match.

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, while value of "any" means at least one of the header pairs must match. Headers can be constructed using a wider range of data types, for example, integer or hash, instead of a string. The headers exchange type (used with the binding argument "any") is useful for directing messages which contain a subset of known (unordered) criteria.

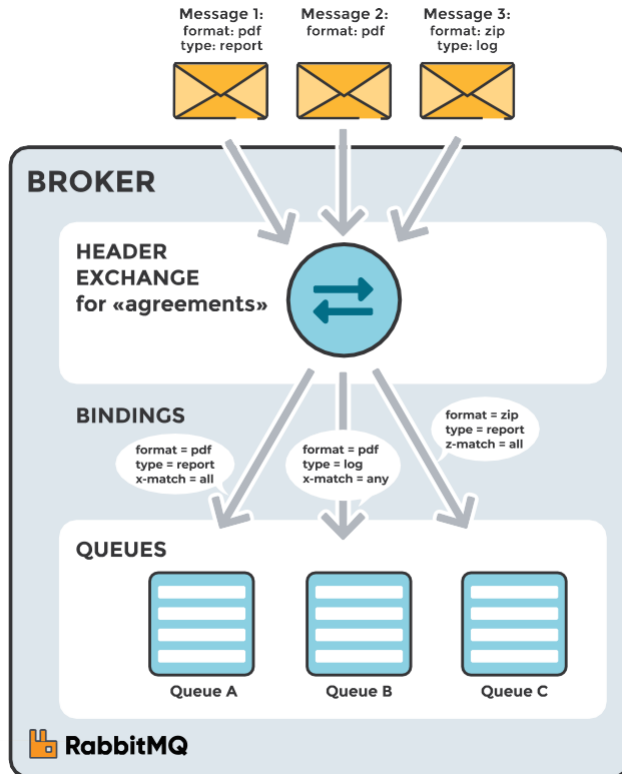


Figure 13 - Headers exchange routes messages to queues that are bound using arguments (key and value) containing headers and optional values.

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

Scenario 1

Message 1 is published to the exchange with header arguments (key = value): "format = pdf", "type = report".

Message 1 is delivered to Queue A because all key/value pairs match, and Queue B since "format = pdf" is a match (binding rule set to "x-match = any").

Scenario 2

Message 2 is published to the exchange with header arguments of (key = value): "format = pdf".

Message 2 is only delivered to Queue B. Because the binding of Queue A requires both "format = pdf" and "type = report" while Queue B is configured to match any key-value pair (x-match = any) as long as either "format = pdf" or "type = log" is present.

Scenario 3

Message 3 is published to the exchange with header arguments of (key = value): "format = zip", "type = log".

Message 3 is delivered to Queue B since its binding indicates that it accepts messages with the key-value pair "type = log", it doesn't mind that "format = zip" since "x-match = any".

Queue C doesn't receive any of the messages since its binding is configured to match all of the headers ("x-match = all") with "format = zip", "type = pdf". No message in this example lives up to these criterias.

DEAD LETTER EXCHANGE

RabbitMQ provides an AMQP extension known as the dead letter exchange. A message is considered dead when it has reached the end of its time-to-live, the queue exceeds the max length (messages or bytes) configured, or the message has been rejected by the queue or nacked by the consumer for some reason and is not marked for re-queueing. A dead-lettered message can be republished to an exchange called dead letter exchange. The message is routed to the dead letter exchange either with the routing key specified for the queue they were on or with the same routing keys with which they were originally published. The exchange then routes the message to a defined dead-letter queue.

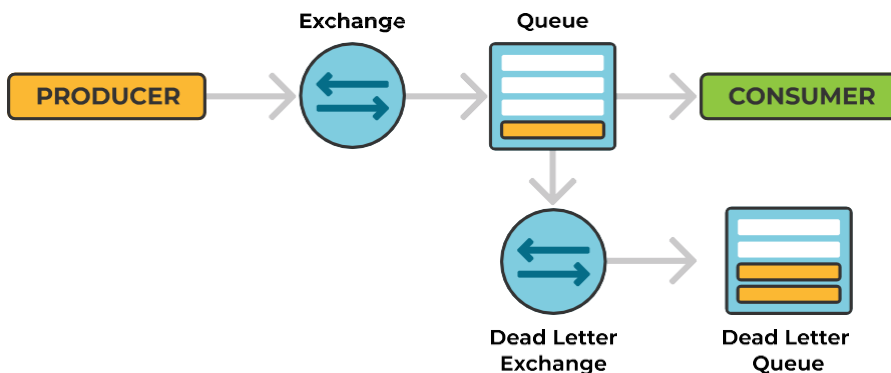


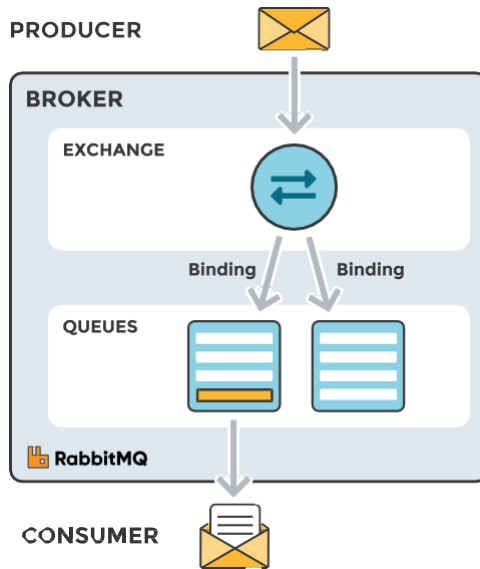
Figure 14 - RabbitMQ Dead Letter Exchange and Dead Letter Queue.

ALTERNATE EXCHANGE

A client may accidentally or maliciously route messages using non-existent routing keys. To avoid complications from lost information, collecting unroutable messages in a RabbitMQ alternate exchange is an easy, safe backup. RabbitMQ handles unroutable messages in two ways based on the mandatory flag setting within the message header. The server either returns the message when the flag is set to "true" or silently drops the message when set to "false". RabbitMQ let you define an alternate exchange to apply logic to unroutable messages.

RABBITMQ AND CLIENT LIBRARIES

A client library that understands the same protocol is needed to communicate with RabbitMQ. Fortunately, there are many options for AMQP client libraries in many different languages, and those client libraries have several methods to communicate with a RabbitMQ instance. This section of the book shows code examples for Ruby, Node.js and Python.



Setup Eclipse IDE

All the examples in this document have been written using Eclipse IDE. So we would suggest you should have the latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from www.eclipse.org/downloads/. Once you download the installation, unpack the binary distribution into a convenient location. For example, in C:\eclipse on Windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

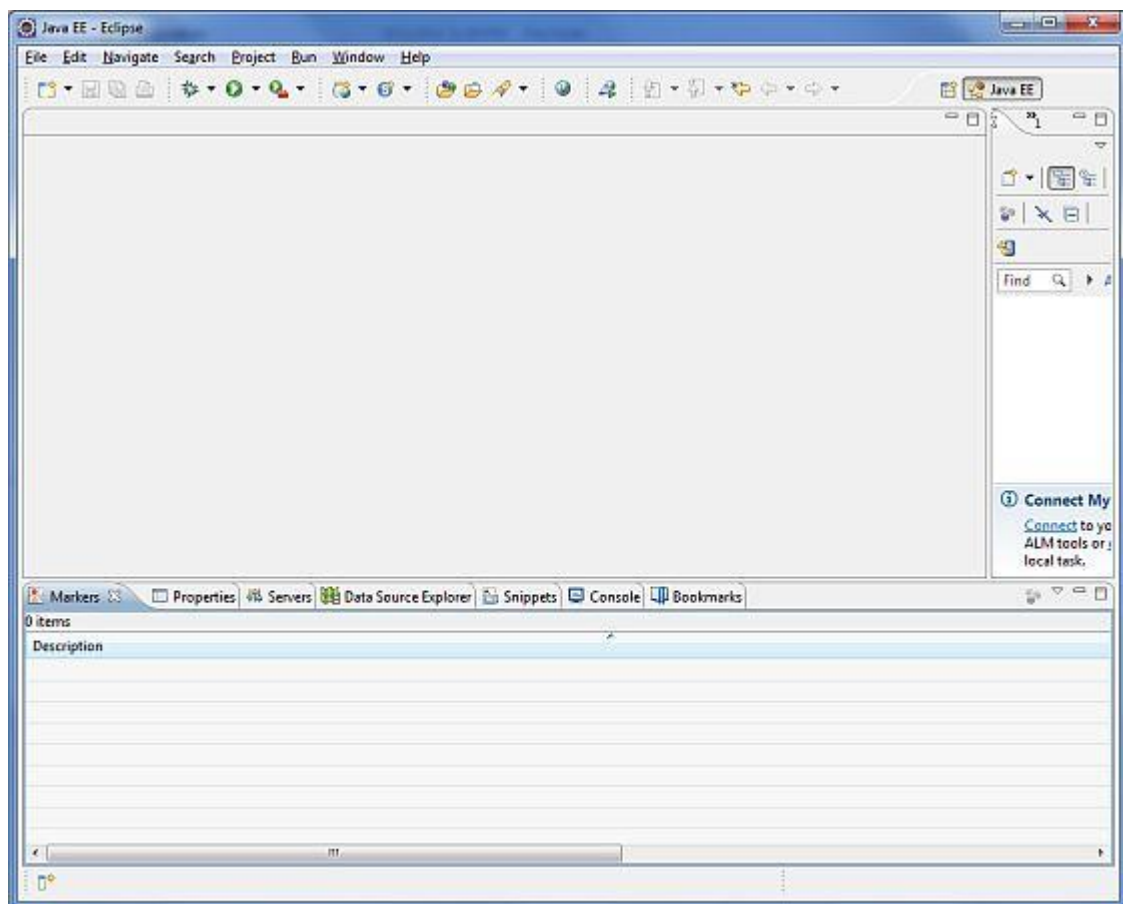
Eclipse can be started by executing the following commands on Windows machine, or you can simply double-click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine –

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display the following result
–



Set Maven

In this document, we are using maven to run and build the spring based examples to run ActiveMQ based applications. Follow the [Maven - Environment Setup](#) to install maven.

RabbitMQ - Producer Application

Now let's create a producer application which will send message to the RabbitMQ Queue.

Create Project

Using eclipse, select **File** → **New** → **Maven Project**. Tick the **Create a simple project(skip archetype selection)** and click Next.

Enter the details, as shown below –

- **groupId** – com.tutorialspoint
- **artifactId** – producer
- **version** – 0.0.1-SNAPSHOT
- **name** – RabbitMQ Producer

Click on Finish button and a new project will be created.

pom.xml

Now update the content of pom.xml to include dependencies for RabbitMQ.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tutorialspoint.activemq</groupId>
  <artifactId>producer</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>RabbitMQ Producer</name>
  <properties>
    <java.version>19</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.rabbitmq</groupId>
      <artifactId>amqp-client</artifactId>
      <version>5.14.2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
```

```

        <version>1.7.26</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.26</version>
    </dependency>
</dependencies>
</project>

```

Now create a Producer class which will send message to the RabbitMQ Queue.

```

package com.tutorialspoint.rabbitmq;

import java.io.IOException;
import java.util.Scanner;
import java.util.concurrent.TimeoutException;

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

public class Producer {
    private static String QUEUE = "MyFirstQueue";

    public static void main(String[] args) throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
            Channel channel = connection.createChannel()) {
            channel.queueDeclare(QUEUE, false, false, false, null);

            Scanner input = new Scanner(System.in);
            String message;
            do {
                System.out.println("Enter message: ");
                message = input.nextLine();
                channel.basicPublish("", QUEUE, null, message.getBytes());
            } while (!message.equalsIgnoreCase("Quit"));
        }
    }
}

```

Producer class creates a connection, creates a channel, connects to a queue. If user enters quit then application terminates else it will send the message to the queue using

basicPublish method.

We'll run this application later.

RabbitMQ - Consumer Application

Now let's create a consumer application which will receive message from the RabbitMQ Queue.

Create Project

Using eclipse, select **File** → **New** → **Maven Project**. Tick the **Create a simple project(skip archetype selection)** and click Next.

Enter the details, as shown below:

- **groupId** – com.tutorialspoint
- **artifactId** – consumer
- **version** – 0.0.1-SNAPSHOT
- **name** – RabbitMQ Consumer

Click on Finish button and a new project will be created.

pom.xml

Now update the content of pom.xml to include dependencies for ActiveMQ.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tutorialspoint.activemq</groupId>
  <artifactId>consumer</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>RabbitMQ Consumer</name>
  <properties>
    <java.version>19</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.rabbitmq</groupId>
      <artifactId>amqp-client</artifactId>
      <version>5.14.2</version>
```

```

</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.26</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.26</version>
</dependency>
</dependencies>
</project>

```

Now create a Consumer class which will receive message from the RabbitMQ Queue.
package com.tutorialspoint.rabbitmq;

```

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

public class Consumer {
    private static String QUEUE = "MyFirstQueue";

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

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

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
            System.out.println("Received '" + message + "'");
        };
        channel.basicConsume(QUEUE, true, deliverCallback, consumerTag -> { });
    }
}

```

Consumer class creates a connection, creates a channel, creates a queue if not existent then receives message from queue if there is any and it will keep polling queue for messages. Once a message is delivered, it is handled by `basicConsume()` method using `deliverCallback`.

We'll run this application later.

RabbitMQ - Test Application

Start the Producer Application

In eclipse, right click on the `Producer.java` source, and select `Run As → Java Application`. Producer application will start running and you'll see the output as follows –

Enter message:

Start the Consumer Application

In eclipse, right click on the `Consumer.java` source, and select `Run As → Java Application`. Consumer application will start running and you'll see the output as follows –

Waiting for messages. To exit press CTRL+C

Send Message

In Producer console window, type `Hi` and press enter button to send the message.

Enter message:

`Hi`

Receive Message

Verify in Consumer console window, the message is received.

Waiting for messages. To exit press CTRL+C

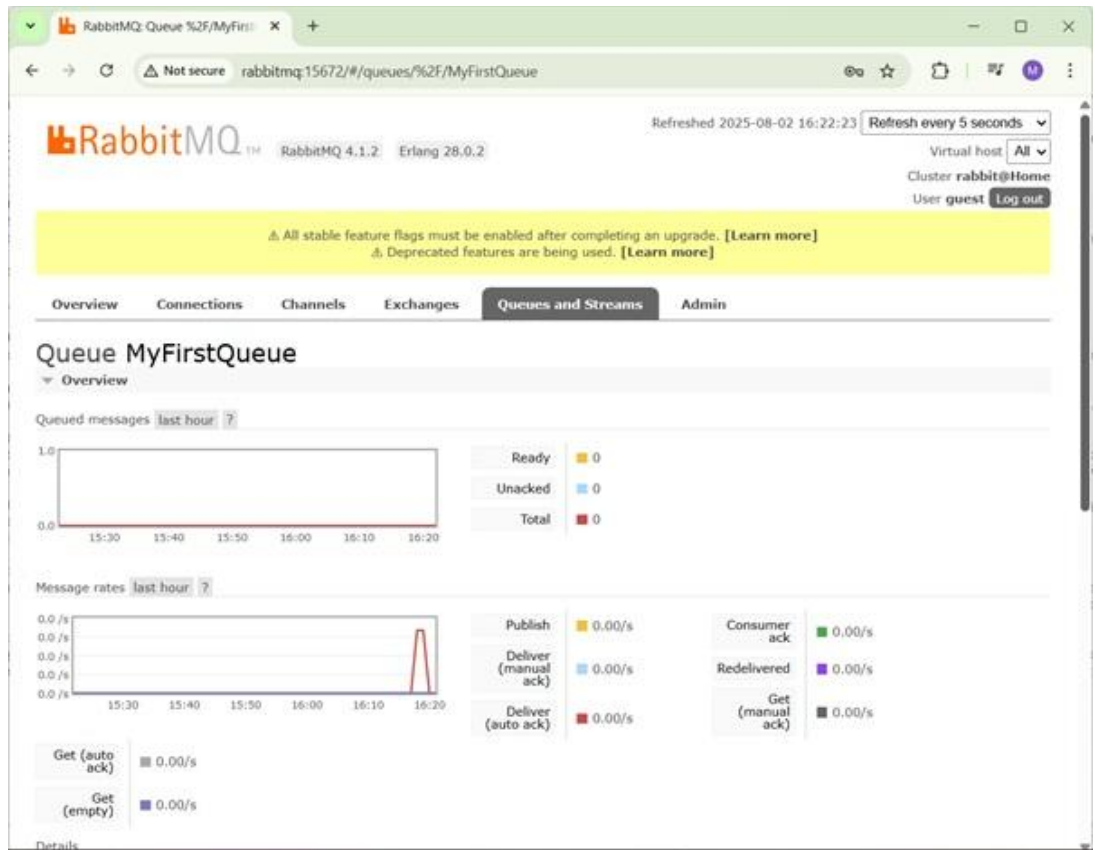
Received = `Hi`

Send `"Quit"` as message to terminate the producer window session and terminate client window session.

Verification

Now open **`http://rabbitmq:15672/`** in your browser. It will ask for credentials. Use

guest/guest as username/password and it will load the RabbitMQ admin console where you can check Queues to check the status. It will show messages enqueued and delivered.



RabbitMQ - Publisher Application

Now let's create a publisher application which will send message to the RabbitMQ Exchange. This exchange will deliver the message to the queue which is bound with the exchange.

Create Project

Using eclipse, select **File** → **New** → **Maven Project**. Tick the **Create a simple project(skip archetype selection)** and click Next.

Enter the details, as shown below –

- **groupId** – com.tutorialspoint

- **artifactId** – publisher
- **version** – 0.0.1-SNAPSHOT
- **name** – RabbitMQ Publisher

Click on Finish button and a new project will be created.

pom.xml

Now update the content of pom.xml to include dependencies for RabbitMQ.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tutorialspoint.activemq</groupId>
  <artifactId>publisher</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>RabbitMQ Publisher</name>
  <properties>
    <java.version>19</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.rabbitmq</groupId>
      <artifactId>amqp-client</artifactId>
      <version>5.14.2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.26</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.7.26</version>
    </dependency>
  </dependencies>
</project>
```

Now create a Publisher class which will send message to the RabbitMQ topic to broadcast it to all the subscribers.

```
package com.tutorialspoint.rabbitmq;

import java.io.IOException;
import java.util.Scanner;
import java.util.concurrent.TimeoutException;

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

public class Publisher {
    private static final String EXCHANGE = "MyExchange";
    public static void main(String[] args) throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
            Channel channel = connection.createChannel()) {
            channel.exchangeDeclare(EXCHANGE, "fanout");
            Scanner input = new Scanner(System.in);
            String message;
            do {
                System.out.println("Enter message: ");
                message = input.nextLine();
                channel.basicPublish(EXCHANGE, "", null, message.getBytes());
            } while (!message.equalsIgnoreCase("Quit"));
        }
    }
}
```

Producer class creates a connection, creates a channel, declare an exchange and then asks user to enter message. The message is sent to exchange and as queue name, we are not passing queue name thus all queues which are bound to this exchange will get the message. If user enters quit then application terminates else it will send the message to the topic.

We'll run this application later.

RabbitMQ - Subscriber Application

Now let's create a subscriber application which will receive message from the RabbitMQ Topic.

Create Project

Using eclipse, select **File** → **New** → **Maven Project**. Tick the **Create a simple project(skip archetype selection)** and click Next.

Enter the details, as shown below –

- **groupId** – com.tutorialspoint
- **artifactId** – subscriber
- **version** – 0.0.1-SNAPSHOT
- **name** – RabbitMQ Subscriber

Click on Finish button and a new project will be created.

pom.xml

Now update the content of pom.xml to include dependencies for RabbitMQ.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tutorialspoint.activemq</groupId>
  <artifactId>subscriber</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>RabbitMQ Subscriber</name>
  <dependencies>
    <dependency>
      <groupId>com.rabbitmq</groupId>
      <artifactId>amqp-client</artifactId>
      <version>5.14.2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.26</version>
    </dependency>
    <dependency>
```

```

    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.26</version>
  </dependency>
</dependencies>
</project>

```

Now create a Subscriber class which will receive message from the RabbitMQ Queue.
package com.tutorialspoint.rabbitmq;

```

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

public class Subscriber {
    private static String EXCHANGE = "MyExchange";
    public static void main(String[] args) throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.exchangeDeclare(EXCHANGE, "fanout");

        String queueName = channel.queueDeclare().getQueue();
        channel.queueBind(queueName, EXCHANGE, "");
        System.out.println("Waiting for messages. To exit press CTRL+C");

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
            System.out.println("Received '" + message + "'");
        };
        channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });
    }
}

```

Subscriber class creates a connection, creates a channel, declares the exchange, create a random queue and binds it with the exchange and then receives message from topic if there is any. Press Ctrl + C to terminate else it will keep polling queue for messages. We'll run this application multiple times to create multiple subscribers later.

RabbitMQ - Test Topic based Application

Start the Publisher Application

In eclipse, right click on the Publisher.java source, and select Run As &arr; Java Application. Publisher application will start running and you'll see the output as follows

–
Enter message:

Start the Subscriber Application

In eclipse, right click on the Subscriber.java source, and select Run As &arr; Java Application. Subscriber application will start running and you'll see the output as follows –

Waiting for messages. To exit press CTRL+C

Start another Subscriber Application

In eclipse, again right click on the Subscriber.java source, and select Run As &arr; Java Application. Another Subscriber application will start running and you'll see the output as follows –

Waiting for messages. To exit press CTRL+C

Send Message

In Publisher console window, type Hi and press enter button to send the message.

Enter message:

Hi

Receive Message

Verify in Subscriber console windows, the message is received in each window.

Received = Hi

Send Quit as message to terminate all publisher and subscriber console window sessions.


Verification

Now open **<http://rabbitmq:15672/>** in your browser. It will ask for credentials. Use guest/guest as username/password and it will load the RabbitMQ admin console where you can check Queues and Exchanges to check the status of messages delivered and bindings.

RabbitMQ Exchange %2F/MyE...

← → ↻ ⚠ Not secure rabbitmq:15672/#/exchanges/%2F/MyExchange

Ⓜ ☆ 📄 🗨 🚪

 RabbitMQ 4.1.2 Erlang 28.0.2

Refreshed 2025-08-02 16:36:43 Refresh every 5 seconds

Virtual host All

Cluster rabbit@Home

User guest Log out

⚠ All stable feature flags must be enabled after completing an upgrade. [\[Learn more\]](#)

⚠ Deprecated features are being used. [\[Learn more\]](#)

Overview

Connections

Channels

Exchanges


Queues and Streams

Admin

Exchange: MyExchange

Overview

Message rates last hour ?



Publish (Out)

0.00/s

Details

Type

fanout

Features

Policy

Bindings

Publish message

Delete this exchange

[HTTP API](#)

[Documentation](#)

[Tutorials](#)


[New releases](#)

[Commercial edition](#)

[Commercial support](#)

[Discussions](#)

[Discord](#)



THE MANAGEMENT INTERFACE

The RabbitMQ Management is a user-friendly way to monitor and handle a 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. It is possible to monitor message rates and send or receive messages manually.

RabbitMQ Management is a plugin that can be enabled for RabbitMQ, enabled by default in CloudAMQP. The management interface 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 debugging the application or when an overview of the whole system is needed. For example, if the number of unacked messages is getting high, it could mean that the consumers are getting slow.

A link to the RabbitMQ management interface can be found on the details page for your hosted RabbitMQ solution on your CloudAMQP instance.

Two other widely used options are available for extended monitoring of RabbitMQ clusters: Prometheus, a monitoring utility, and Grafana, a plugin for visualizing metrics. When combined, these two tools create a robust system for extended data collection and monitoring.

Concepts

- **Cluster** - a group of interconnected servers that work together as a single system.
- **Node** - a single server in the RabbitMQ cluster.

OVERVIEW

The overview gives a quick view of the cluster. It shows two graphs; one graph for queued messages and one with the message rate (Figure 13). The time interval shown in the graph can be changed by pressing the text *last minute* above the graph. Information about all different statuses for messages can be found by pressing the *question mark*.

Queued messages

This graph shows the total number of queued messages for all queues. The ready display shows the number of messages that are available to be delivered. Unacked shows the number of messages for which the server is waiting for acknowledgment.

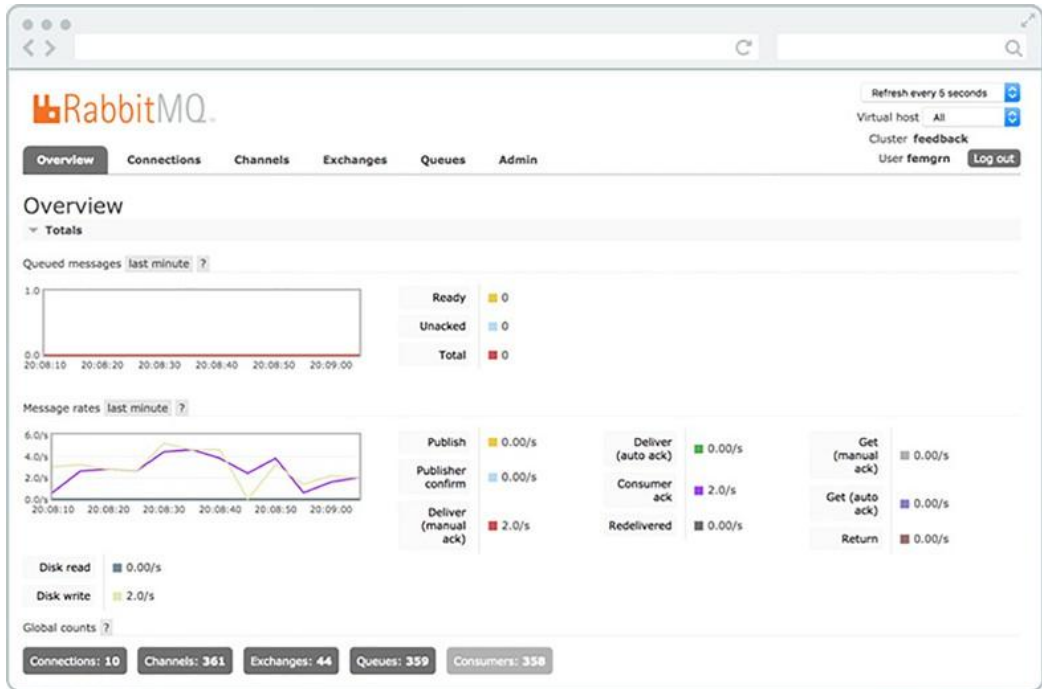


Figure 15 - The RabbitMQ management interface.

Message rate

Message rate show how fast the messages are being handled. *Publish* shows the rate at which messages are entering the server and *Confirm* shows the rate at which the server is confirming.

Global Count

Global count represents the total number of connections, channels, exchanges, queues and consumers for all virtual hosts to which the current user has access.

Nodes

The nodes display shows information about the different nodes in the RabbitMQ cluster. There is also information about server memory, the number of Erlang processes per node, and other node-specific information here. *Info* shows 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 16 - Node-specific information.

Import/export definitions

Configuration definitions can be imported or exported. When downloading the definitions, a JSON representation of the broker (the RabbitMQ settings) is given. This JSON can be used to restore exchanges, queues, vhosts, policies, and users. This feature can also be used as a backup solution.

CONNECTIONS AND CHANNELS

RabbitMQ connections and channels can be in different states including *starting*, *tuning*, *opening*, *running*, *flow*, *blocking*, *blocked*, *closing*, or *closed*. If a connection enters *flow-control* this often means the client is being rate-limited in some way.

Import / export definitions	
<p>Export</p> <p>Filename for download: er-01_2015-5-25.json</p> <p>Download broker definitions (?)</p>	<p>Import</p> <p>Definitions file: Bladdra... Ingen fil är vald.</p> <p>Upload broker definitions (?)</p>
<p>HTTP API Command Line</p> <p>Update every 5 seconds</p>	

Figure 17 - Import/export definitions as a JSON file.

Connections

The connections tab (Figure 18) shows the connections established to the RabbitMQ server. *Virtual hosts* shows in which vhost the connection operates and *User name* shows the user associated with the connection. *Channels* displays the number of channels using the connection. *SSL/TLS* indicate whether the connection is secured with SSL or not.

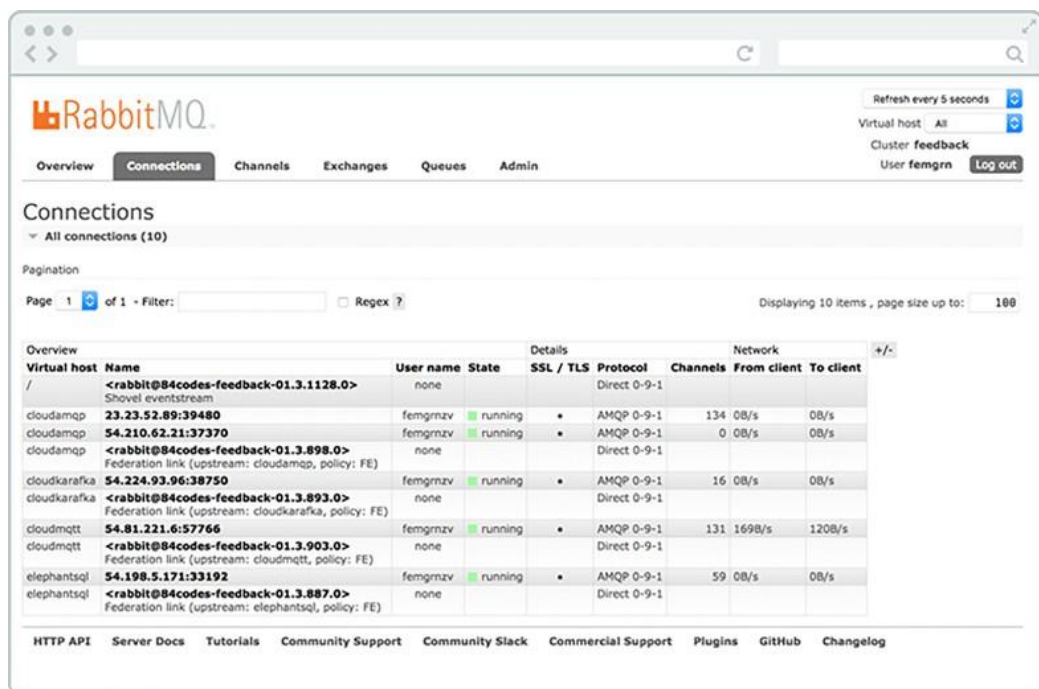


Figure 18 - The Connections tab in the RabbitMQ management interface.

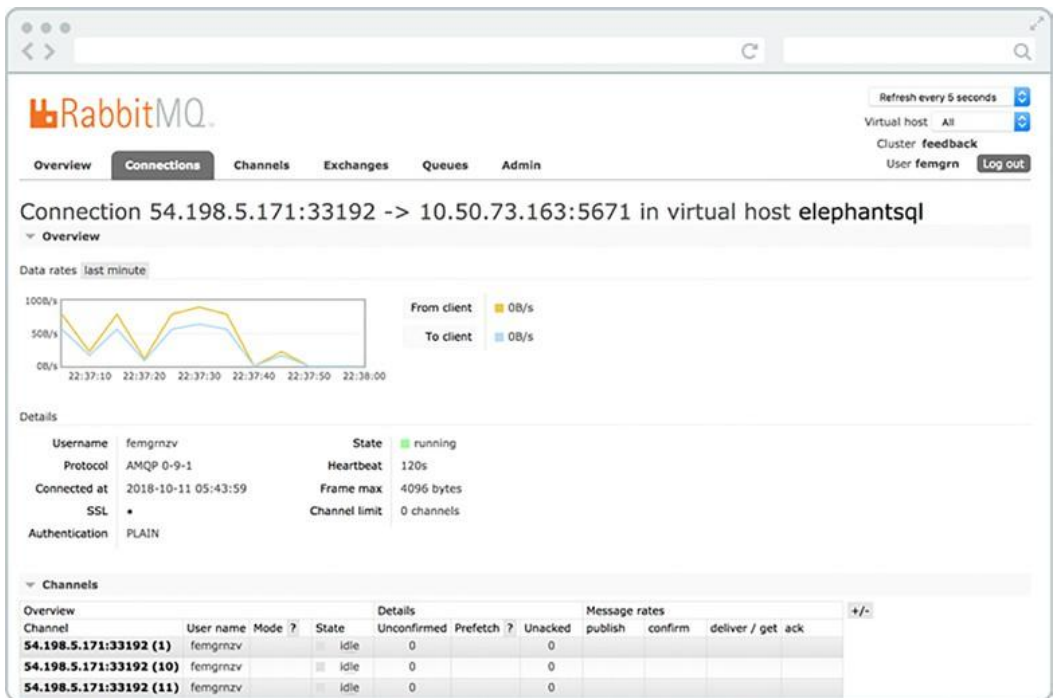


Figure 19 - Connection information for a specific connection.

Clicking on one of the connections gives an overview of that specific connection (Figure 19) including the channels within the connection and the data rates as well as client properties. The connection can be closed here as well.

More information about the attributes associated with connections can be found in the manual page for `rabbitmqctl`, in the command line tool for the broker.

Channels

The channel tab (Figure 20) shows information about all the current channels. The *Virtual host* shows in which vhost the channel operates and the *User name* shows the user associated with the channel. The *guarantee* mode can be in *confirm* or *transactional* mode. When a channel is in *confirm* mode, both the broker and the client count messages. The broker then confirms messages as it handles them by sending back a *basic.ack* on the channel. 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
<rabbit@84codes-feedback-01.3.1128.0> (1)	/	none		running	0	20	0			2.4/s	2.4/s
23.23.52.89:39480 (1)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (10)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (100)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (101)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (102)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (103)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (104)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (105)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (106)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (107)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (108)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (109)	cloudamqp	femgrmzv		idle	0		0				
23.23.52.89:39480 (11)	cloudamqp	femgrmzv		idle	0		0				

Figure 20 - The Channels tab.

Clicking on one of the channels provides a detailed overview of that specific channel (Figure 21). The message rate and the number of logical consumers retrieving messages from the channel are also displayed.

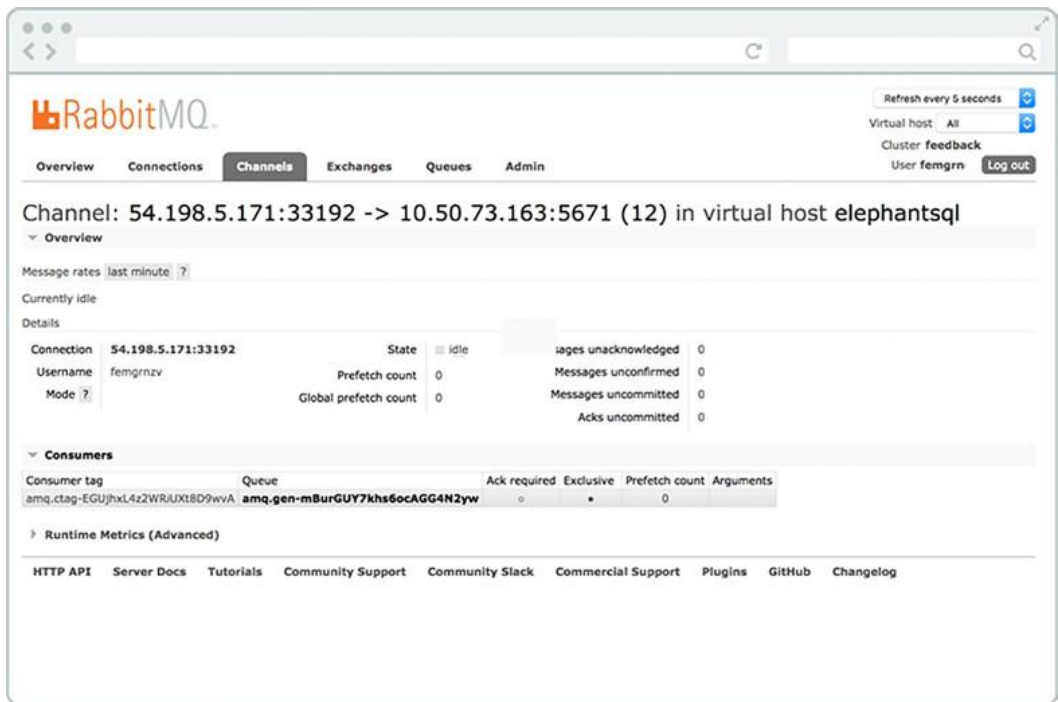


Figure 21 - Detailed information about a specific channel.

More information about the attributes associated with a channel can be found in the manual page for `rabbitmqctl`, which is in the command line tool the RabbitMQ broker.

EXCHANGES

All exchanges can be listed from the exchange tab (Figure 22). Virtual host shows the vhost for the exchange. Type is the exchange type such as direct, topic, headers and fanout. Features show the parameters for the exchange (*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, which are created by default.

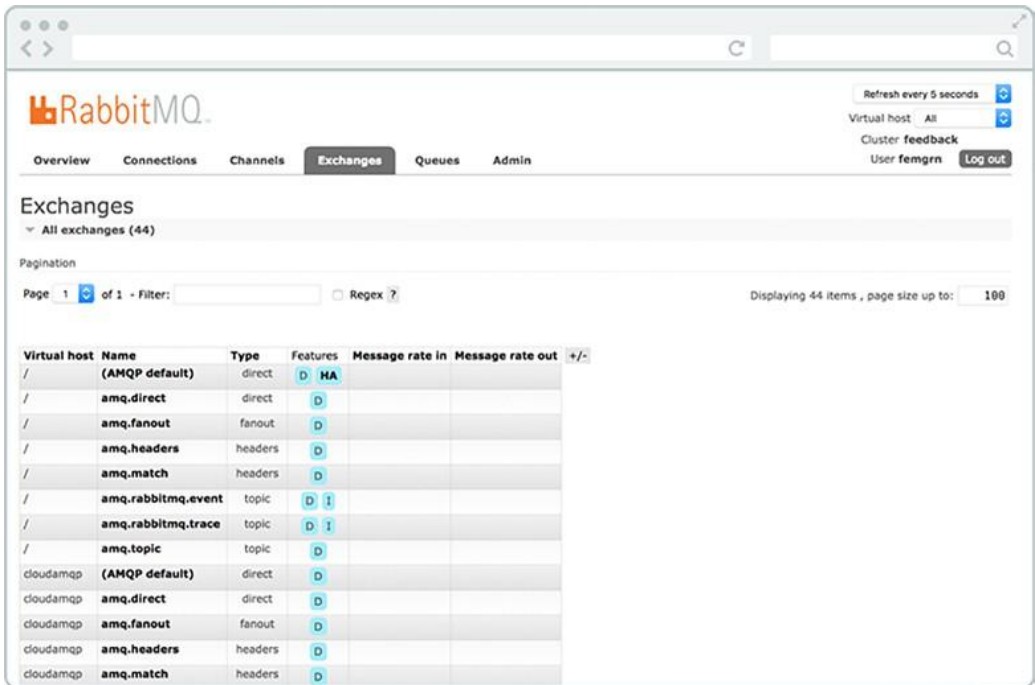


Figure 22 - The exchanges tab in the RabbitMQ management interface.



Figure 23 - Detailed view of an exchange.

Clicking on the exchange name displays a detailed page about the exchange (Figure 23). Adding bindings to the exchange and viewing already existing bindings can also be performed here as well as publishing a message to the exchange or deleting the exchange.

QUEUES

The Queues view shows the queues for all or one selected vhost (Figure 24). Queues may also be created from this area. Queues have different parameters and arguments depending on how they were created. The features column shows the parameters that belong to the queue, including:

- **Durable** - Ensures that RabbitMQ never loses the queue.
- **Message TTL** - The time a message published to a queue can live before being discarded.
- **Auto-expire** - The time a queue can be unused before it is automatically deleted.
- **Max length** - How many ready messages a queue can hold before it starts to drop them.
- **Max length bytes** - The total size of ready messages a queue can hold before it starts to drop them.

Clicking on any chosen queue from the list of queues will show all information about it (Figure 25).

The first two graphs include the same information as the overview but only the number and rates for this specific queue.

The screenshot shows the RabbitMQ management interface with the 'Queues' tab selected. The interface includes a navigation bar with tabs for Overview, Connections, Channels, Exchanges, Queues, and Admin. The Queues tab is active, displaying a list of queues. The list is filtered by 'server-info' and shows two items: 'server-info.overview' and 'server-info.version'. Each item has columns for Virtual host, Name, Features, State, Messages (Ready, Unacked, Total), and Message rates (Incoming, deliver, get, ack).

Overview		Features	State	Messages			Message rates				+/-
Virtual host	Name			Ready	Unacked	Total	Incoming	deliver	get	ack	
cloudkafka	server-info.overview	D	Idle	0	0	0					
cloudmqtt	server-info.version	D	Idle	0	0	0	0.00/s	0.00/s	0.00/s		

Figure 24 - The queues tab in the RabbitMQ management interface.

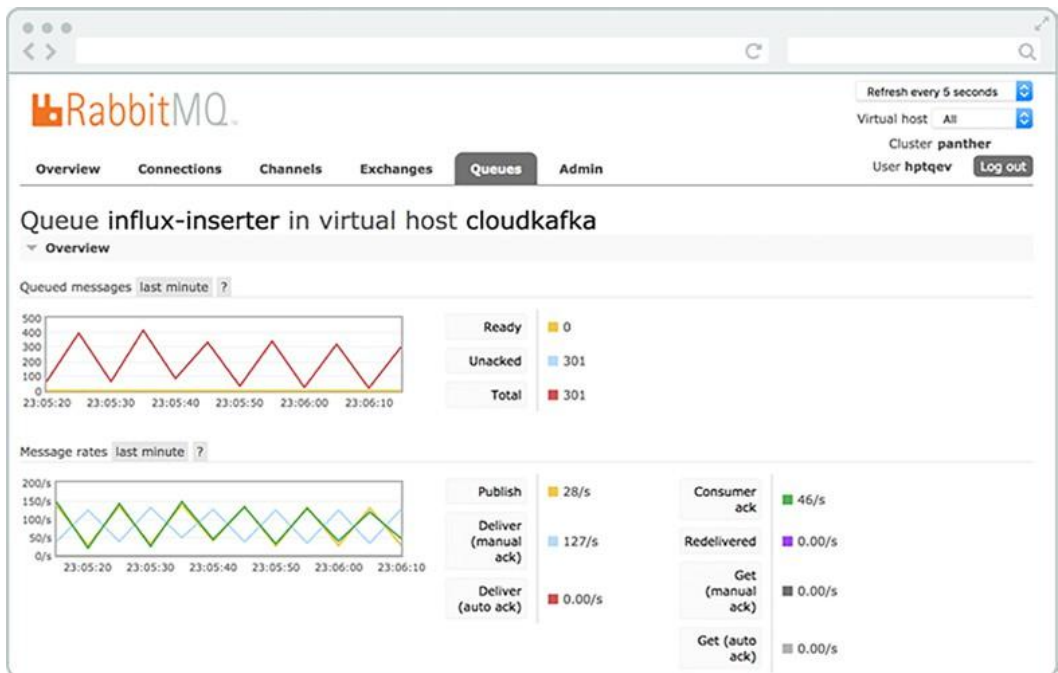


Figure 25 - Specific information about a single queue.

CONSUMERS

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

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

Figure 26 - Consumers connected to a queue.

Bindings

All active bindings to the queue are shown under bindings. New bindings to queues can be created from here or unbinding a queue from an exchange (Figure 27).

Bindings

From	Routing key	Arguments
(Default exchange binding)		
exchange	routingKey	

↓

This queue

Add binding to this queue

From exchange:

Routing key:

Arguments: = String

Bind

Figure 27 - The bindings interface.

Publish message

Publishing a message can be performed manually to the queue from this area. The message will be published to the default exchange with the queue name as its routing key, ensuring that the message will be sent to the proper 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 28 - Manually publishing a message to the queue.

Get message

Manually inspecting the message in the queue can be done in this area. *Get message* gets the first message in the queue. The *requeue* option will cause RabbitMQ to place it back in the queue in the same order.

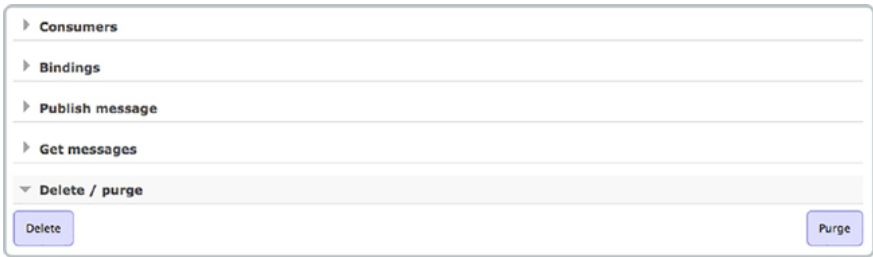


The screenshot shows the 'Get messages' panel in the RabbitMQ web interface. It features a warning message: 'Warning: getting messages from a queue is a destructive action. (?)'. Below the warning, there are three input fields: 'Requeue' with a dropdown menu set to 'Yes', 'Encoding' with a dropdown menu set to 'Auto string / base64' and a help icon '(?)', and 'Messages' with a text input field containing the number '1'. At the bottom of the panel is a button labeled 'Get Message(s)'.

Figure 29 - Manually inspect a message.

Delete or Purge queue

A queue can be deleted by pressing the *delete* button or the queue can be emptied with use of the *purge* function.



The screenshot shows the 'Delete / purge' panel in the RabbitMQ web interface. It has a sidebar on the left with a list of actions: 'Consumers', 'Bindings', 'Publish message', 'Get messages', and 'Delete / purge'. The 'Delete / purge' action is selected and expanded, showing two buttons: 'Delete' on the left and 'Purge' on the right.

Figure 30 - Delete or purge a queue from the web interface.

ADMIN

The Admin view (Figure 31) is where users are added and permissions for them are changed. This area is also used to set up vhosts (Figure 32), policies, federation, and shovels. Information about shovels can be found here: <https://www.rabbitmq.com/shovel.html> while information about federation will be given in part two of this book.

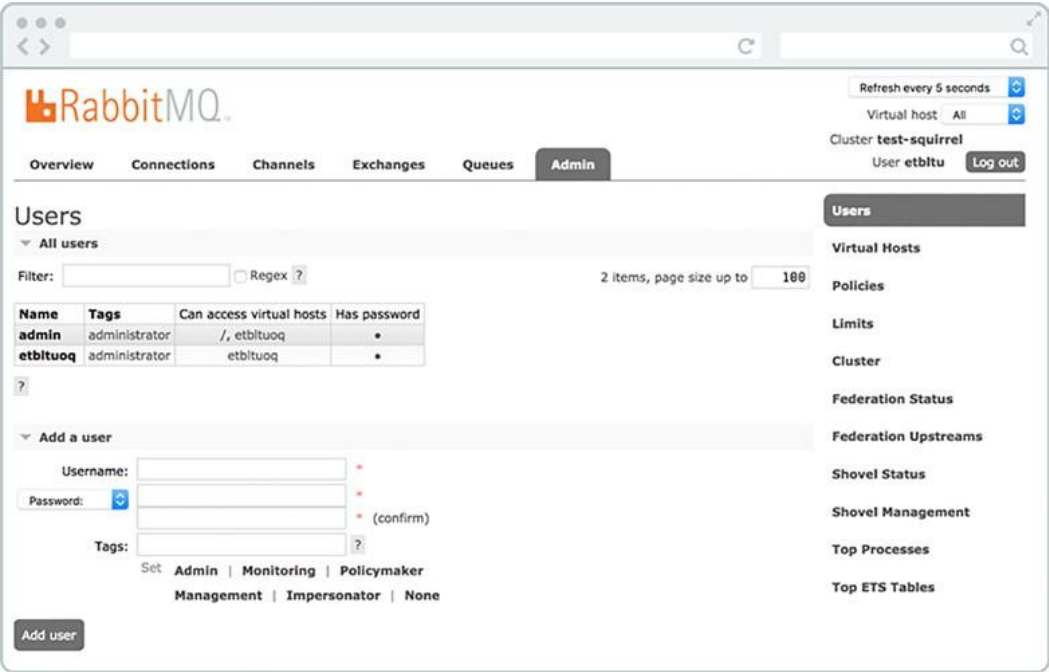


Figure 31 - The Admin interface where users can be added.

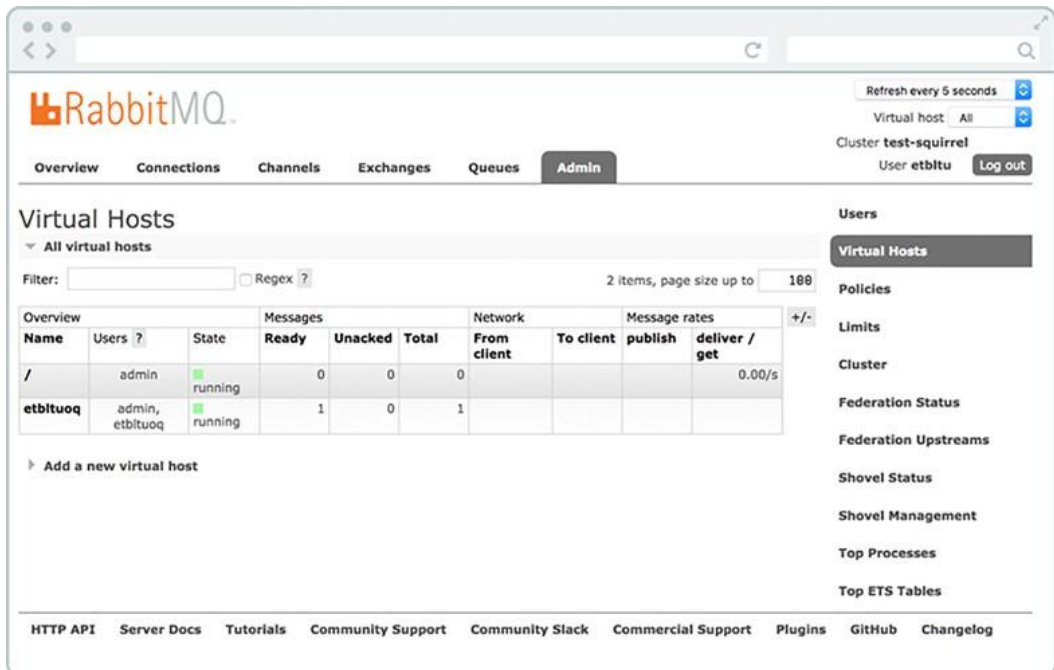


Figure 32 - Virtual Hosts can be added from the Admin tab.

The example in Figure 33, shows how to create an example queue and an exchange called *example.exchange* (Figure 34).

The exchange and the queue are connected by a binding called *pdfprocess* (Figure 35). Messages can be published (Figure 36) to the exchange with the routing key *pdfprocess*, and will end up in the queue (Figure 37).

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 33 - Queue view, add queue.

The management interface is extremely useful in handling many functions, and is a great tool to use as an overview of the system and the relationship between the functions of a message 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 34 - Exchange view, add exchange.

Add binding from this exchange

To queue

:

rabbitmq-example

*

Routing key:

pdfprocess

Arguments:

=

String

Bind

Figure 35 - Click on the exchange or on the queue, go to "Add binding from this exchange" or "Add binding to this queue".

Publish message

Routing key:

Delivery mode:

Headers: (?) =

Properties: (?) =

Payload:

Figure 36 - Publish a message to the exchange with the routing key "pdfprocess".

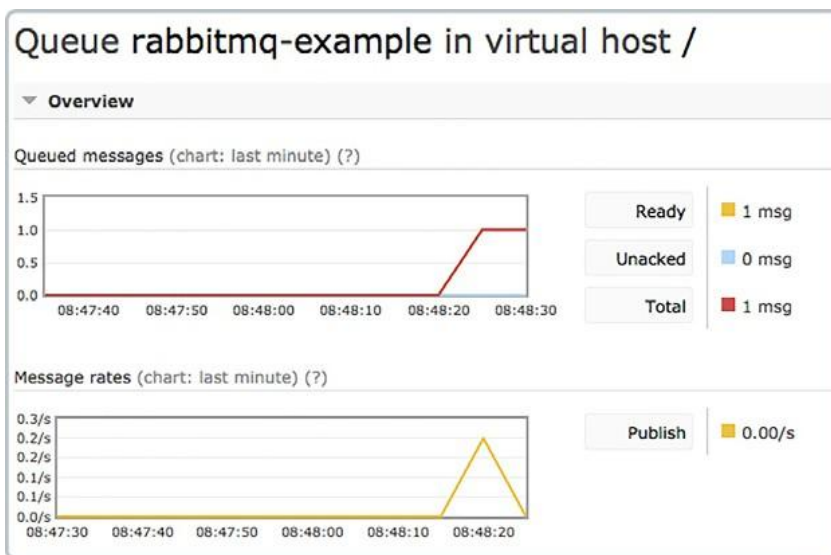


Figure 37 - Queue overview for example-queue when a message is published.

ARGUMENTS AND PROPERTIES

RabbitMQ has arguments and properties that can be used to define behaviors. Properties are defined by the AMQP protocol and included in RabbitMQ. Arguments can be any key-value pair and are used for feature extensions. Some properties are mandatory while others are optional, and all arguments are optional. Properties and Arguments can be defined for Queues, Exchanges, and Messages in RabbitMQ. Examples of a property for a queue is: *passive*, *durable* and *exclusive*. Properties are specified in the AMQP protocol 0.9.1.

An argument is an optional feature for defining behaviors, implemented by the RabbitMQ server. These arguments are also known as x-arguments and can sometimes be changed after queue declaration. An example of an argument for messages and queues is *TTL*, time to live. Defining properties and arguments can be beneficial, and in some cases crucial. Properties and Arguments do provide an easy and secure approach to keeping your RabbitMQ server tidy and healthy, minimizing unnecessary resource usage or letting a faulty client cause issues by publishing millions of messages to a single queue.

Arguments are additional, optional features to the mandatory properties. Some Arguments can be dynamically changed after the creation of the queue or exchange.

QUEUE PROPERTIES AND ARGUMENTS

Examples of Queue Properties include *passive*, which determines if the queue already exists, and *durable*, which tells if the queue remains when a server restarts.

An example of Queue Arguments includes *x-max-priority*, which sets a maximum number of priorities, or *x-message-ttl*, which sets queue TTL.

EXCHANGE PROPERTIES AND ARGUMENTS

An example of an Exchange Property is *durable*, which tells if the exchange remains when a server restarts, and *internal* which tells that the exchange can not be used directly by publishers.

Examples of Exchange Arguments include *x-dead-letter-exchange* and *x-dead-letter-routing-key*, which are used by the dead letter exchange.

SET ARGUMENTS AND PROPERTIES

A property can be set while creating the queue, via code, via the management interface, or via policies.

Code example

A queue's arguments are normally set on a per queue basis when the queue is declared by the client. How the arguments are set varies from client to client.

A queue can be marked as *durable*, which specifies if the queue should survive a RabbitMQ restart. Setting a queue property as *durable* only means that the queue definition will survive a restart, not the messages in it. Create a durable queue by specifying durable as *True* during the creation of the queue.

Code example of how to declare a durable queue:

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

Dead letter exchanges are no different than other exchanges except added arguments. Code example of how to apply arguments for an exchange:

```
channel.queue_declare("test_queue", arguments={"x-dead-letter-exchange": "dlx_exchange", "x-dead-letter-routing-key": "dlx_key"})
```

Via the RabbitMQ Management Interface

A queue or exchange can be created via the Management Interface. A message can also be sent through the management interface.

When manually creating a queue through the RabbitMQ Web Management Interface the arguments are set in a free text field and can be added to it with quick links. Properties can be set as *true* or *false*.

The screenshot shows the 'Add a new queue' form in the RabbitMQ Management Interface. The form includes the following fields and options:

- Type:** Classic (dropdown)
- Name:** QueueName (text input)
- Durability:** Durable (dropdown)
- Auto delete:** No (dropdown)
- Arguments:** A table with two rows:
 - Row 1: x-max-length = 50000 (Number dropdown)
 - Row 2: x-message-ttl = 6000 (Number dropdown)
 - Row 3: (empty) = (empty) (String dropdown)
- Quick links:** A row of links with question marks: Auto expire, Message TTL, Overflow behaviour, Single active consumer, Dead letter exchange, and Dead letter routing key.
- Additional links:** A row of links with question marks: Max length, Max length bytes, Maximum priority, Lazy mode, Version, and Master locator.
- Submit button:** Add queue (orange button)

Figure 38 - Queue created with the durability property, and two arguments, x-max-length and x-message-ttl.

▼ Publish message

Message will be published to the default exchange with routing key **Test**, routing it to this queue.

Delivery mode:

1 - Non-persistent ▼

Headers: ?

=

String ▼

Properties: ?

expiration

=

36000000

Payload:

Hello world!

Payload encoding:

String (default) ▼

Publish message

Figure 39 - A message published with TTL expiration set to 36000000 milliseconds.

ARGUMENTS AND POLICIES

To set arguments, the use of policies is recommended. Policies make it possible to configure arguments for one or many queues at once, and the queues will all be updated when you're updating the policy definition. To reduce the overhead work of configuring every single queue and exchange with arguments, the use of policies is perfect. Policies enable a way to configure multiple queues or exchanges in a consistent way, reducing the risk of sloppy mistakes in the configuration. A queue can only be applied by one policy simultaneously, but there is a priority system along with the regex recognition in order to manage several policies.

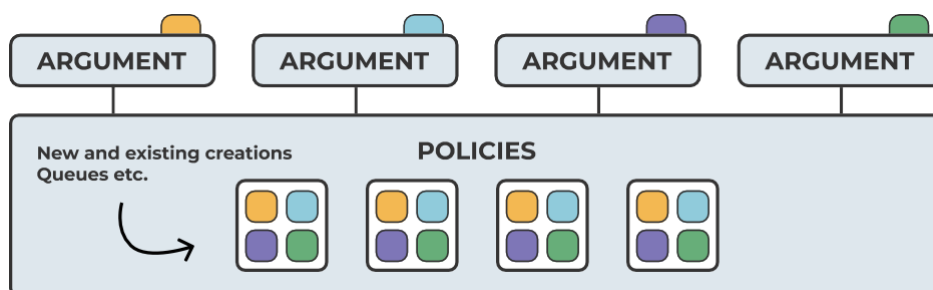


Figure 40 - RabbitMQ Queue arguments applied through policies.

POLICIES

In order to ensure uniformly configured queues and exchanges, RabbitMQ (AMQP 0.9.1) includes the ability to define Policies and Arguments. Policies can be advantageously used to apply queue or exchange arguments to more than one created queue/exchange. Policies are created per vhost, with a pattern that defines where it will be applied and a parameter that defines what the policy will do.

The specifications of the AMQP protocol (0.9.1) enable support for various features, called Arguments. Depending on which argument you implement, changes can be made to their settings after the queue declaration. Arguments define certain configurations, such as message and queue TTL, different consumer priorities, and queue length limit. Policies make it possible to configure arguments for one or many queues and exchanges at once, and the queues/exchanges will all be updated when the policy definition is updated. Policies can be changed at any time, and changes will affect all matching queues and exchanges.

When using policies, argument configurations and updates don't have to be done for every single queue or exchange. Policies simply ensure that all matching queues or exchanges come with the desirable preset arguments, suitable for their purpose, and also ensure that updates of one single argument are applied on all queues or exchanges bound to it.

A policy can be set when you want to apply a TTL on a set of queues. Policies could be used when you want to delete single or multiple queues at once, or when you want to delete all messages from a queue.

POLICIES IN RABBITMQ

A policy is applied when the pattern, a regular expression, matches a queue or exchange. As soon as a policy is created it will be applied to the matching queues and/or exchanges and its arguments will be amended to the definitions. As the match occurs continuously changes can easily be applied to multiple queues that are up and running. For example, if a TTL is to be set on a group of queues, or if multiple queues are to be deleted or purged at once. A policy is also applied every time an exchange or queue is created if a match exists. Only one policy can be matched to every queue or exchange at once, but one policy may be set to apply multiple arguments.

Policies are created per vhost, with a pattern that defines where it will be applied and a parameter that defines what the policy will do. The parameter is entered as a key (the parameter name) and a value (the parameter value), also called a key-value pair. Policies can be set from a terminal using `rabbitmqctl` or by using the HTTP API or Web Management Interface.

Create and view policies

To create a policy, define the following:

- Name of the policy
- The vhost where the policy lives (default is `/`)
- A pattern or exact match to determine to which queues or exchanges it applies
- A definition consisting of one or several key-value pairs
- A priority and how it will be applied in relation to other policies (default is 0)

Policies can be viewed and created from either of the following:

- The RabbitMQ management interface
- A terminal using `rabbitmqmqctl`
- The HTTP API

Policies in the RabbitMQ management interface

Policies are listed under *policies* in the RabbitMQ management interface. On the same page as the *Add/update* section, a new policy can be created.

The name, patterns, and definition fields are mandatory. Below the definitions box, a selection of keys is listed that can be added to the policy by clicking them. Values have to be added to the definitions box for every key added.

A priority should be used if multiple policies are used where the patterns overlap.

POLICIES IN RABBITMQCTL

rabbitmqctl is a command like tool for managing a RabbitMQ server where policies can be listed and created. Here are some examples:

List Policies

List policies by using the command:

```
rabbitmqctl list_policies -p vhostname
```

Create Policies

Create a policy by using the command:

```
rabbitmqctl set_policy <name> <pattern> <definition>
```

Example:

```
rabbitmqctl set_policy Policy2 '.*' '{"message-ttl": 60}'
```

To specify vhost, priority, and exchange/queue-application use the following syntax:

```
rabbitmqctl set_policy <name> <pattern> <definition> -p <vhost> --apply-o  
<queues|exchanges> --priority 8
```

Example:

```
rabbitmqctl set_policy Policy5 'queue_A' '{"message-ttl": 60}' -p zyxhvjpx  
--apply-to queues --priority 10
```

Delete Policies

Deleting policies is done with the command:

```
rabbitmqctl clear_policy <name>
```

Example:

```
sudo rabbitmqctl clear_policy 'Policy1'
```

POLICIES WITH THE HTTP API

The following commands are available for policies via RabbitMQ HTTP API.

List Policies

Policies can be listed with the following API call:

```
curl -u USERNAME:PASSWORD -X GET https://SERVERNAME.amq.cloudamqp.com/api/policies
```

Create Policies

Policies can be added with the following API call:

```
curl -XPUT -u USERNAME:PASSWORD --header "Content-Type: application/json"
--data '{"pattern":"[PATTERN]","definition":{"key":value},
"apply-to":["queues|exchanges"]}' https://host/api/policies/[VHOST]/[POLICYNAME]
```

Delete Policies

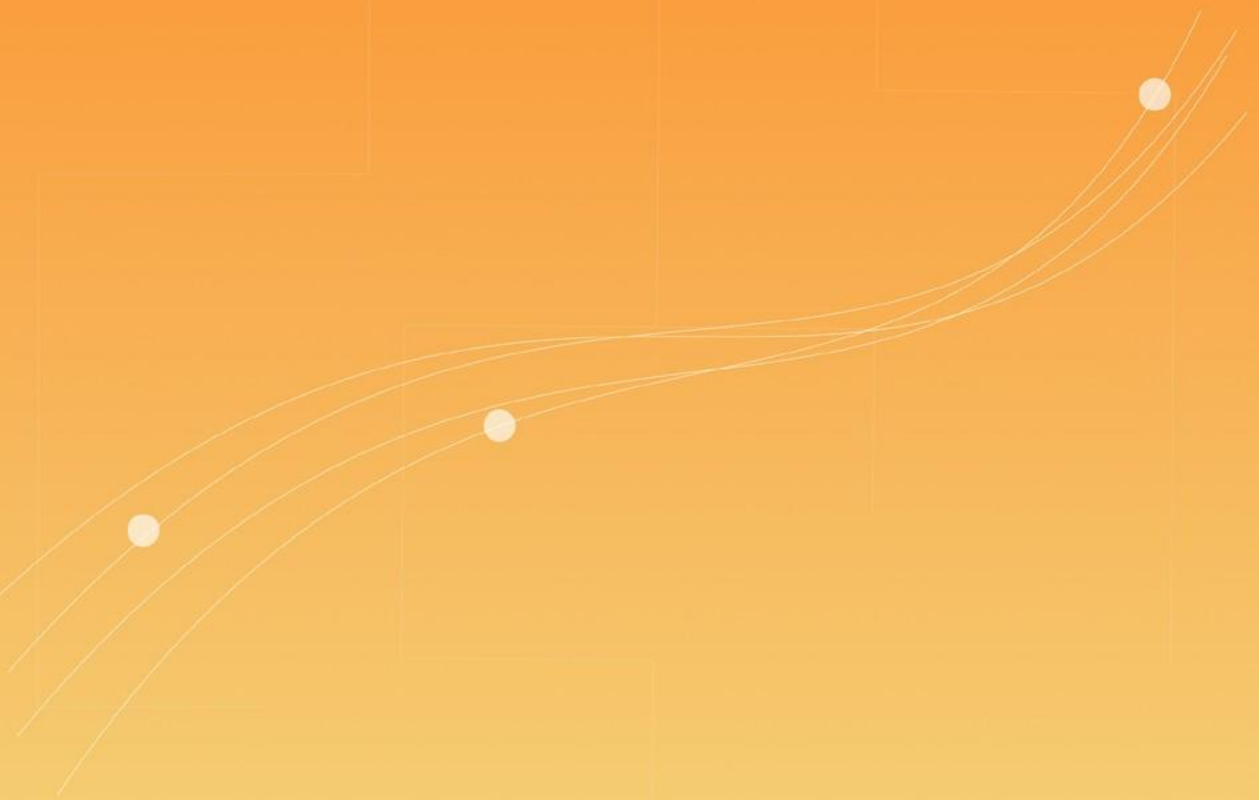
Policies can be deleted with the following API call:

```
curl -u USERNAME:PASSWORD -X DELETE https://host/api/policies/[VHOST]/[POLICYNAME]
```

ADVANCED MESSAGE QUEUEING

BEST PRACTICE

Various configurations affect your RabbitMQ cluster in different ways. Learn how to optimize performance through the setup.



Some applications require high throughput while other applications publish batch jobs that can be delayed. Tradeoffs must be accepted between performance and guaranteed message delivery. The goal when designing the system should be to maximize combinations of performance and availability that make sense for the specific application. Bad architectural design decisions and client-side bugs can damage the broker or affect throughput.

Part 2 of this book talks about the dos and don'ts along with best practices for two different usage categories, high availability and high performance (high throughput). But first, the attention turns to advanced message queueing features including how to migrate a cluster with queue federation, quorum queues, streams and prefetching of messages.

QUORUM QUEUES

Perhaps one of the most significant changes in RabbitMQ 3.8 was the new queue type called Quorum Queues. This is a replicated queue to provide high availability and data safety.

Quorum queues ensure that the cluster is up-to-date by agreeing on the contents of a queue. By doing so, quorum queues avoid losing data. Quorum queues are available as of RabbitMQ 3.8.0. All communication is routed to the queue leader, which means the queue leader locality has an effect on the latency and bandwidth requirement of the messages.

In quorum queues, the leader and replication are consensus-driven, which means they agree on the state of the queue and its contents. Quorum queues will only confirm when the majority of its nodes are available, which thereby avoids data loss.

Declare a quorum queue using the following command:

```
rabbitmqadmin declare queue name=<name> durable=true arguments="{\"x-queue-type\": \"quorum\"}"
```

This will declare a quorum queue with up to five replicas, which is the default. For example, a cluster of three nodes will have three replicas, one on each node. If you had a cluster of seven nodes, five out of the seven nodes would each host one replica while two particular nodes would not have any replicas.

After declaring a quorum queue, you can bind it to any exchange just as with other queue types. Queues must be durable and instantiated by setting the *x-queue-type* header to *quorum*. If the majority of nodes agree on the contents of a queue, the data is valid. Otherwise, the system attempts to bring all queues up to date.

Quorum queues have support for the handling of poison messages, which are messages that are never consumed completely or positively acknowledged. The number of unsuccessful delivery attempts can be tracked and displayed in the *x-delivery-count* header. A poison message can be dead-lettered when it has been returned more times than configured.

PREFETCH

Knowing how to tune your broker correctly brings the system up to speed without having to set up a larger cluster or doing a lot of updates in your client code. Understanding how to optimize the RabbitMQ prefetch count maximizes the speed of the system.

The RabbitMQ prefetch value is used to specify how many messages are being sent at the same time.

Messages in RabbitMQ are pushed from the broker to the consumers. The RabbitMQ default prefetch setting gives clients an unlimited buffer, meaning that RabbitMQ, by default, sends as many messages as it can to any consumer that appears ready to accept them. It is, therefore, possible to have more than one message "in flight" on a channel at any given moment.

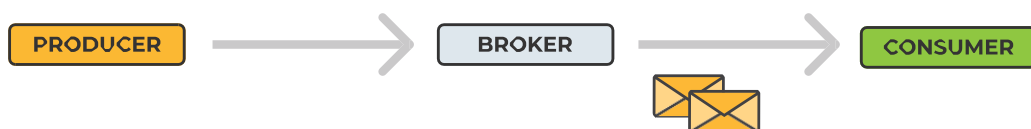


Figure 41 - Message flow in RabbitMQ.

Messages are cached by the RabbitMQ client library (in the consumer) until processed. All pre-fetched messages are invisible to other consumers and are listed as unacked messages in the RabbitMQ management interface.

An unlimited buffer of messages sent from the broker to the consumer could lead to a window of many unacknowledged messages. Prefetching in RabbitMQ simply allows you to set a limit of the number of unacked (not handled) messages.

There are two prefetch options available, channel prefetch count and consumer prefetch count.

CHANNEL PREFETCH COUNT AND CONSUMER PREFETCH COUNT

The channel prefetch value defines the max number of unacknowledged deliveries that are permitted on a channel. Setting a limit on this buffer caps the number of received messages before the broker waits for an acknowledgment.

Because a single channel may consume from multiple queues, coordination between them is required to ensure that they don't pass the limit. This can be a slow process especially when consuming across a cluster, and it is not the recommended approach.

The best practice is to set a consumer prefetch by setting a limit on the number of unacked messages at the client.

Please note that the prefetch value does not have an impact if you are using the *Basic.get* request.

HOW DO I SET THE PREFETCH COUNT?

RabbitMQ uses AMQP version 0.9.1 by default. The protocol includes the quality of service method *Basic.qos* for setting the prefetch count. RabbitMQ allows you to set either a channel or consumer count using this method.

Consider the following Pika example:

```
channel.basic_qos(10, global=False)
```

The *basic_qos* function contains the global flag. Setting the value to *false* applies the count to each new consumer. Setting the value to *true* applies a channel prefetch count to all consumers. Most APIs set the global flag to *false* by default.

Optimizing the prefetch count requires that you are considering the number of consumers and messages your broker handles. There is a negligible amount of additional overhead. The broker must understand how many messages to send to each consumer instead of each channel.

THE OPTIMUM CONSUMER PREFETCH COUNT

A larger prefetch count generally improves the rate of message delivery. The broker does not need to wait for acknowledgments as often and the communication between the broker and consumers decreases. Still, smaller prefetch values can be ideal for distributing messages across larger systems. Smaller values maintain the evenness of message consumption. A value of one helps ensure equal message distribution.

A prefetch count that is set too small may hurt performance since RabbitMQ might end up in a state, where the broker is waiting to get permission to send more messages. Figure 42 illustrates a long idling time with QoS prefetch setting of one (1). Then RabbitMQ won't send out the next message until after the round trip completes (deliver, process, acknowledge). Round-trip time in the illustration is in total 125ms with a processing time of only 5ms.

A large prefetch count, on the other hand, could take lots of messages off the queue and deliver all of them to one single consumer, keeping the other consumers in an idling state, as illustrated in Figure 43.

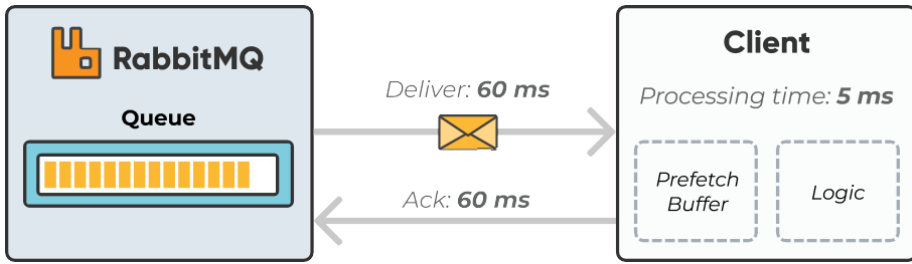


Figure 42 - RabbitMQ Prefetch round trip



Figure 43 - Idle consumer due to large prefetch count.

SET THE CORRECT PREFETCH VALUE

When a single consumer (or a few consumers) are processing messages quickly, the recommendation is prefetching many messages at once to keep your client as busy as possible. If you have about the same processing time all the time and network behavior remains the same, simply take the total round trip time and divide by the processing time on the client for each message to get an estimated prefetch value.

In a situation with many consumers and short processing time, we recommend a lower prefetch value. A value that is too low will keep the consumers idling a lot since they need to wait for messages to arrive. A value that is too high may keep one consumer busy while other consumers are being kept in an idling state.

If you have many consumers and/or long processing time, we recommend setting the prefetch count to one (1) so that messages are evenly distributed among all your workers.

Please note that if your client auto-acks messages, the prefetch value will have no effect.

Avoid the usual mistake of having an unlimited prefetch, where one client receives all messages and runs out of memory and crashes, causing all the messages to be re-delivered.

RABBITMQ STREAMS

INTRODUCTION

Queues in RabbitMQ are great! They anchor the communication between producers and consumers. Replicated queues (e.g. Quorum Queues) even orchestrate communication with reliability and data safety. However, there are scenarios where queues fall flat or crawl on their knees. What scenarios?

Queues are limited in the following scenarios:

- They deliver the same message to multiple consumers by binding a dedicated queue for each consumer. Clearly, this could create a scalability problem.
- They erase read messages making it impossible to re-read(replay) them or grab a specific message in the queue.
- They perform poorly when dealing with millions of messages because they are optimized to gravitate toward an empty state.

The RabbitMQ team introduced Streams in RabbitMQ 3.9 to mitigate the above-listed challenges. But what are RabbitMQ Streams?

RABBITMQ STREAMS INTRODUCTION

RabbitMQ Streams perform the same tasks as queues in that they buffer messages from producers that consumers read. However, Streams differ from queues in two ways:

- How producers write messages to them
- And how consumers read messages from them

Under the hood, Streams model an append-only log that's immutable. This means messages written to a Stream can't be erased; they can only be read. A more scholarly description would be to call this behavior of Streams "non-destructive consumer semantics."

To read messages from a Stream in RabbitMQ, one or more consumers subscribe to it and read the same message as many times as they want. Additionally, Streams are always persistent and replicated.

Like queues, consumers talk to a Stream via AMQP-based clients and, by extension, use the AMQP protocol. Alternatively, consumers can connect to a Stream via the binary stream protocol. The stream protocol fosters faster message flow when working with RabbitMQ Streams.

All these unique sets of characteristics compound to make Streams in RabbitMQ a dramatic shift from queues. The Stream wasn't created to replace queues but to complement them. Streams open up endless possibilities for new RabbitMQ use cases like the scenarios identified earlier.

Let's explore these use cases a little bit deeper.

WHEN TO USE RABBITMQ STREAMS

The use cases where streams shine include:

- **Fanout architectures:** Where many consumers need to read the same message
- **Replay & time-travel:** Where consumers need to read and reread the same message or start reading from any point in the stream.
- **Large Volumes of Messages:** Streams are great for use cases where large volumes of messages need to be persisted.
- **High Throughput:** RabbitMQ Streams process relatively higher volumes of messages per second.

Fanout Architectures

A fanout architecture is where multiple consumers read the same message. As mentioned earlier, implementing this sort of architecture with queues isn't optimal. Having to add queues for every added consumer is resource intensive, which gets worse when dealing with queues that need to persist data.

Streams in RabbitMQ make implementing fanout architectures a breeze. Because consumers read messages from a Stream in a non-destructive manner, a message will always be there for the next consumer to access it. In essence, to implement a fanout architecture, declare a RabbitMQ Stream and bind as many consumers as needed.

Figure 44 below depicts what implementing a fanout with a Stream would look like.

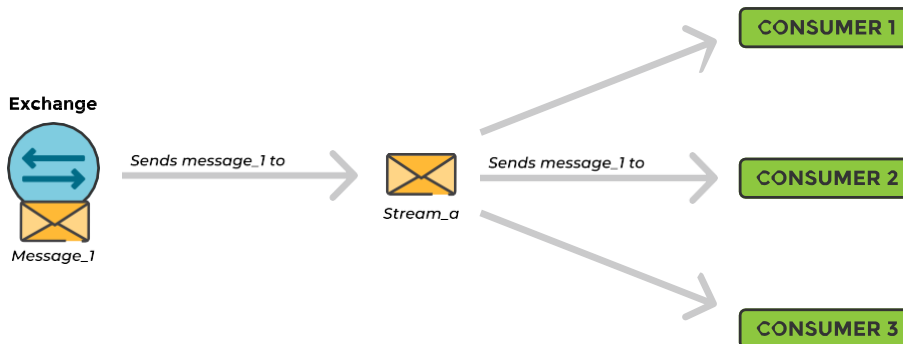
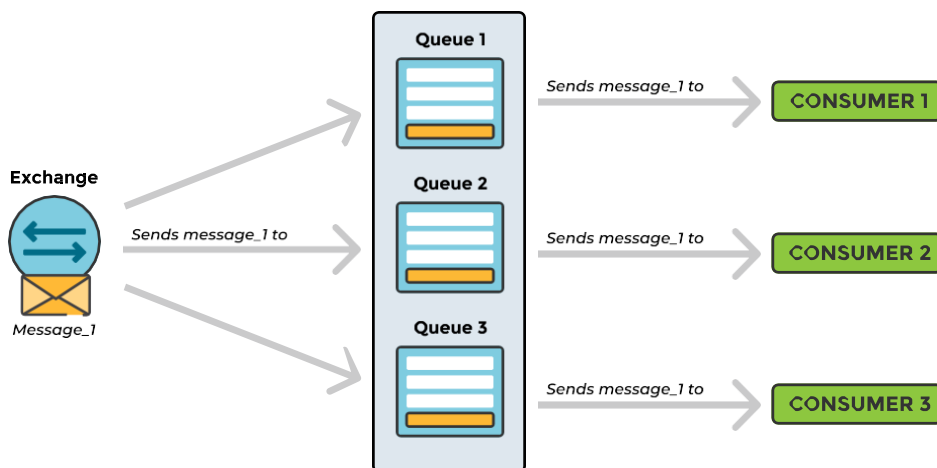


Figure 44 - Example of fanout in RabbitMQ Streams.

On the other hand, trying to achieve the same thing with queues would look like what's shown in **Figure 45**.

Figure 45 - Fanout implemented with Queues.



Replay & Time Travel

RabbitMQ Streams are also fit for use cases where a consumer needs to re-read the same message, which isn't possible with queues. Aside from re-reading messages, it is also possible to start reading messages from any point in the Stream.

This easy replay and time-travel feature of Streams is made possible with offsets. Offsets are to Streams what indexes are to arrays or keys to hash maps. To start consuming messages from a specific point/index in a Stream, just specify an offset in the consumer query. Essentially, every message in a Stream is associated with an offset.

For example, **Figure 46** shows messages and what their corresponding offsets would look like in a given Stream.

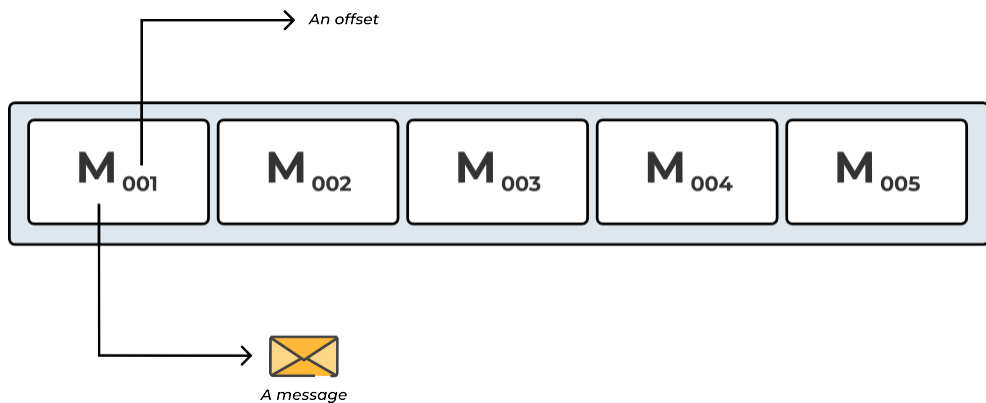


Figure 46 - Queue offset in RabbitMQ Streams.

Large Volumes of Messages

RabbitMQ Streams are perfect when persisting large volumes of messages. Streams shine in this area because they store messages on the file system. As a result, a Stream in RabbitMQ could grow indefinitely until the host disk space runs out.

As running out of disk might not be desirable, RabbitMQ Streams allow setting a maximum log data size. When the upper limit is reached, the oldest messages are discarded, preventing the Stream from consuming the entire disk space.

High Throughput

If a RabbitMQ use case requires processing high volumes of messages per second, then using a Stream is the best option.

In comparison, Quorum queues handled about 40,000 messages per second. Streams, on the other hand, handled around 64,000 messages per second when used with AMQP protocol and over 1 million messages per second when used with native Stream protocol.

This high throughput is a testament to the simplicity of the Stream data structure and the Stream protocol itself. For example, since the Stream protocol doesn't handle things like routing messages, de-queueing messages, etc., it technically does less work, and this translates to higher performance.

*RABBITMQ
STREAMS LIMITS &
CONFIGURATIONS*

Beyond the default configurations that are bundled with a Stream, you might want to customize these default settings in some scenarios. This chapter looks at some of these configurations as well as the limitations of RabbitMQ Streams.

You can configure a Stream in RabbitMQ using queue arguments specified with a policy key or at the time of queue declaration.

DATA RETENTION CONFIGURATIONS

Since RabbitMQ Streams are immutable, they inherently tend to grow infinitely. As this is an undesirable behavior, a Stream can be configured to discard old messages through a retention policy. A retention policy configures a Stream to truncate its messages once it reaches a given size or a specified age. Truncating messages entails deleting an entire segment file. But what is a segment file?

RabbitMQ Streams do not persist messages in a big, single file. Instead, a Stream is broken down into smaller files known as segment files. A Stream truncates its size by deleting a segment file and all its messages. To configure a Stream's retention strategy, you can adopt size or time-based retention strategies.

- Size-based retention strategy - the Stream is configured to truncate its size once the total size of the stream reaches a given value.
- Time-based retention strategy - the Stream is configured to truncate a segment file once that segment reaches a given age.

Setting up the sized-based retention strategy requires providing the following arguments when declaring the Stream. This can also be done through a policy:

- `x-max-length-bytes`
- `x-stream-max-segment-size-bytes`

On the other hand, setting up the time-based retention strategy requires providing the following arguments when declaring the Stream:

- `x-max-age`
- `x-stream-max-segment-size-bytes`

In both strategies, the `x-stream-max-segment-size-bytes` argument is required. Let's dive into the significance of these arguments and understand their purpose.

x-max-length-bytes

This argument will control the maximum size of the RabbitMQ Stream. When this is set, RabbitMQ will delete segment files from the beginning of the Stream. The deletion happens when the Stream's total size reaches the value of `x-max-length-bytes`.

For example, if the maximum size of a Stream is set to `"x-max-length-bytes":100000000`, the Stream will discard the oldest messages when the Stream's disk usage hits 100000000 bytes. RabbitMQ does not provide a default value for this.

The unit could be in KB, MB, GB, or TB, however, when you just provide a value for this argument without a unit, it will default to bytes.

max-age

This argument will control how long a message survives in a RabbitMQ Stream. The unit of this configuration could either be in years (Y), months (M), days (D), hours (H), minutes (M), or seconds (S).

For example, if the `max-age` of a Stream is set to, `"x-max-age":"30D"`, the Stream will discard segment files that have been there for 30 days or more. RabbitMQ does not provide a default value for this.

x-stream-max-segment-size-bytes

As mentioned earlier, RabbitMQ Streams encompass one or more segment files on disk, and this argument controls the size of each segment. For example, if the maximum size of the segment file of a Stream is set to `"x-stream-max-segment-size-bytes":50000`, each segment file will have a maximum size of 50000 bytes. RabbitMQ provides a default value for this: 500000000 bytes

Returning to the topic of the `x-stream-max-segment-size-bytes` argument, it is required in both retention strategies. Let's explore the reasons for its necessity in both cases.

The `max-age` and `x-max-length-bytes` arguments are important for the retention of messages in RabbitMQ Streams, but the retention is evaluated on a per-segment basis. Essentially, Streams only apply the retention policies whenever an existing segment file has reached its maximum size and is closed in favor of a new one.

As a result, if the `x-stream-max-segment-size-bytes` argument is not provided, the Stream will never know when to close the current segment file and create a new one. And, by extension, invoke the retention policy. This is why this argument is required in the size and time-based retention strategies.

Note: The `x-max-length-bytes`, and the `x-max-age`, arguments can be combined. And, of course, always provide the third required argument. In that case, the Stream will only discard messages when both conditions are true. For example, if the `x-max-length-bytes`, is 100 (not ideal) and the `x-max-age` is 30D, the Stream will only discard segment files that have been in the Stream for more than 30 days only when the Stream's disk usage reaches 100. In essence, even if there are segment files whose `max-age` has exceeded the limit, the Stream won't discard them until the max length is exceeded and vice versa.

Controlling the Initial Replication Factor

Remember Streams are persistent and replicated. When a Stream is initialized, RabbitMQ will create a replica of the Stream on some randomly selected nodes in the cluster. However, the number of replicas can be controlled in two ways:

- with the `x-initial-cluster-size` queue argument when declaring the Stream via an AMQP client.
- with the `initial-cluster-size` queue argument when declaring the Stream via the stream plugin.

x-initial-cluster-size

This argument controls the number of nodes in the cluster on which the Stream will be replicated. Like quorum queues and replicated classic queues, streams are affected by cluster sizes. The more replicas a stream has, the more data needs to be replicated, lowering the throughput. It is recommended to use an uneven cluster size to constitute a quorum, such as 1, 3, or 5.

For example, `"x-initial-cluster-size": 3`

RabbitMQ Stream Leader Election Configuration

Even though a Stream would always have replicas across nodes, there is always the leader replica or node. All Stream operations go through the leader replica first and then replicated on the other nodes. Which node becomes the replica is controlled in three ways:

- By passing the `x-queue-leader-locator` argument when declaring the Stream
- By setting the `queue-leader-locator` policy key
- By defining the `queue_leader_locator` in the configuration file

The supported values for leader election configuration are:

- `client-local` - This is the default value. The client that declares the Stream is usually connected to some node. The client-local value elects this node to be the leader.
- `Balanced` - If there are less than 1000 queues, make the node hosting the minimum number of Stream leaders the leader. Else, make a random node the leader.
-

RABBITMQ STREAMS LIMITATIONS

Message Encoding

Streams store messages as AMQP 1.0 encoded data. When publishing using AMQP 0.9.1 a conversion is done under the hood. While this conversion will often play out well, sometimes it doesn't. For example, if the header of an AMQP 0.9.1 message contains complex values like arrays/lists, the header will not be converted. That is because headers in an AMQP 1.0 message can only contain values of simple types, such as strings and numbers.

UI Metric Accuracy

When working with Streams, sometimes the Management UI does not reflect the precise message count. In streams, Offset Tracking information also counts as messages, making the message count artificially larger than it is. This should make no practical difference in most systems.

We have now covered some optional configurations that make it easier to tweak a Stream for a specific use case. Overall, Streams weren't created to replace queues but to complement them. Streams open up new possibilities for RabbitMQ use cases.

At CloudAMQP, you can activate the Streams plugin with a click of a button.

QUEUE FEDERATION

RabbitMQ supports federated queues, which have several uses, including when collecting messages from multiple clusters to a central cluster, when distributing the load of one queue to multiple other clusters, and/or when migrating to another cluster without stopping all producers/consumers.

Queue federation connects an upstream queue to transfer messages to the downstream queue provided there are consumers that have capacity. It is perfect to use when migrating between two clusters. Consumers and publishers can be moved in any order, and the messages will not be duplicated (unlike exchange federation). The federated queue will only retrieve messages when it has run out of messages locally, when it has consumers that need messages, and when the upstream queue has "spare" messages that are not being consumed.

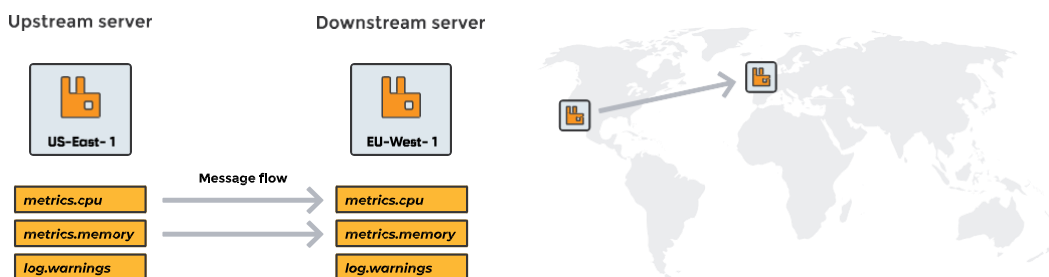


Figure 47 - Upstream and downstream servers.

See Figure 47 for an illustration of the concept of upstream and downstream servers. The upstream server is where messages are initially published, while the downstream server is where the messages are forwarded.

Queue Federation example setup

In this example, there is already one cluster set up in Amazon US-East-1 (named dupdjffe, as seen in Figure 48). This cluster is going to be migrated to a cluster in EU-West-1 (in the pictures named aidajcdt). In this case, the server in US-East-1 will be defined as the upstream server of EU-West-1.

Some queues in the cluster in US-East-1 are going to be migrated via federation, the metric-queues (metric.cpu and metric.memory).

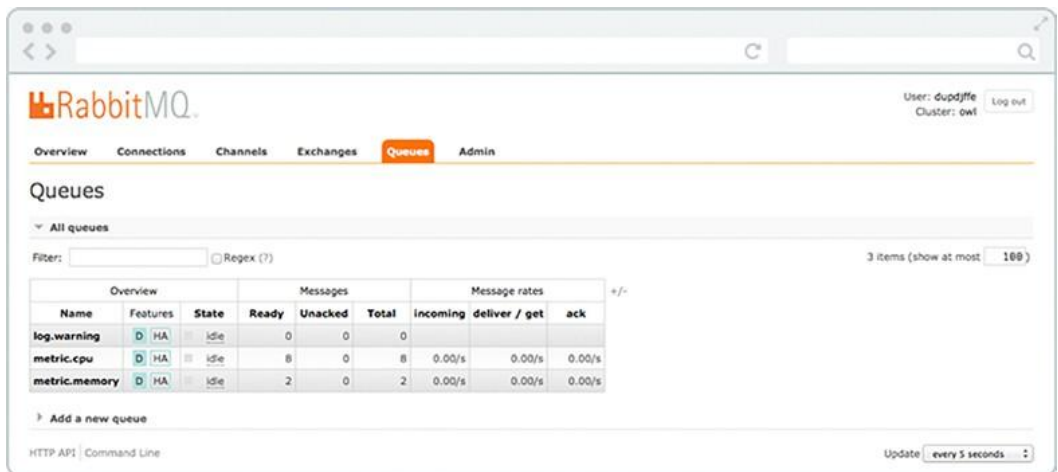


Figure 48 - Migration of *metric.cpu* and *metric.memory*.

- Set up a new cluster.**

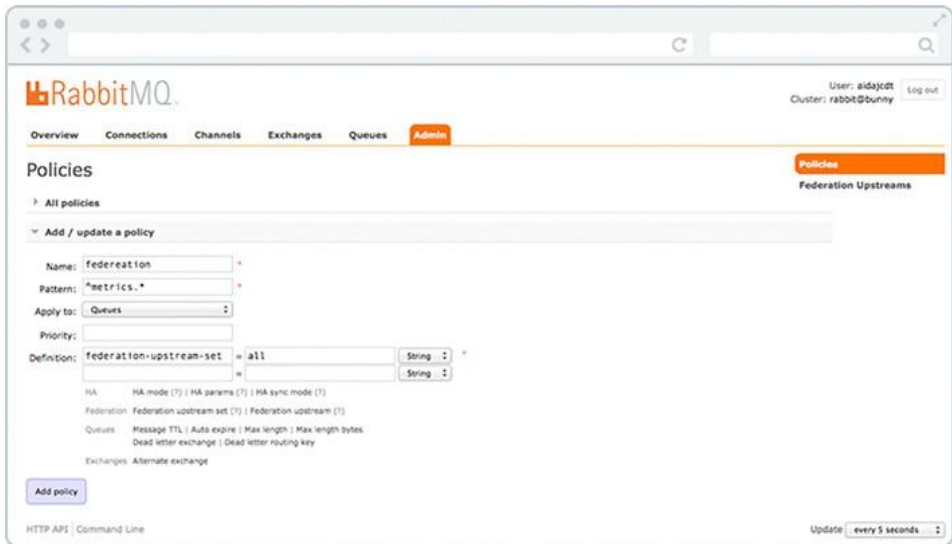
Start by setting up the new cluster; in this case EU-West-1.

- Create a policy that matches the queues to federate.**

A policy is a pattern against which queue names are matched. The *pattern* argument is a regular expression used to match queue (or exchange) names. In this case, we tell the policy to federate all queues whose names begin with *metric*.

Navigate to *Admin* -> *Policies* and press *Add/update* to create the policy (Figure 49). A policy can apply to an upstream set or a single upstream of exchanges and/or queues. In this example, it is applied to all upstream queues with *federation-upstream-set* set to all.

Note: Policies are matched every time an exchange or queue is created.



The screenshot shows the RabbitMQ Admin UI with the 'Policies' tab selected. The 'Add / update a policy' form is visible. The 'Name' field is 'federation', the 'Pattern' is '^metrics.*', and 'Apply to' is set to 'Queues'. The 'Definition' is 'federation-upstream-set' with two parameters, both set to 'all' and 'String'. A 'Log out' button is in the top right corner.

Policies

Policy: federation-upstream-set

Definition: federation-upstream-set

HA: HA mode (?:) | HA params (?:) | HA sync mode (?:)

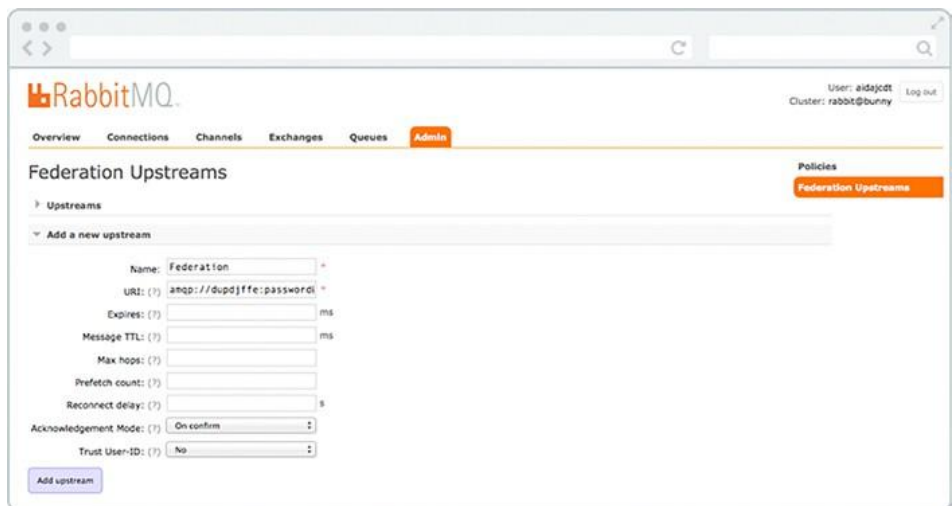
Federation: Federation upstream set (?:) | Federation upstream (?:)

Queues: Message TTL (?:) | Auto expire (?:) | Max length (?:) | Max length bytes (?:)

Exchanges: Alternate exchange (?:)

Update: every 5 seconds

Figure 49 - Set up the federation to the upstream.



The screenshot shows the RabbitMQ Admin UI with the 'Federation Upstreams' tab selected. The 'Add a new upstream' form is visible. The 'Name' field is 'Federation', the 'URL' is 'amqp://dupdiffe:password@', and 'Expires' is set to 'ms'. The 'Add upstream' button is at the bottom left.

Federation Upstreams

Upstreams

Add a new upstream

Name: Federation

URL: amqp://dupdiffe:password@

Expires: ms

Message TTL: ms

Max hops: ms

Prefetch count: ms

Reconnect delay: ms

Acknowledgement Mode: On confirm

Trust User-ID: No

Add upstream

Figure 50 - Set up the federation to the upstream.

3. Start by setting up the new cluster, in this case the cluster in EU-West-1

Open the management interface for the downstream server EU-West-1, go to the Admin -> Federation Upstreams screen, and press Add (Figure 50). Fill in all information; the URI should be the URI of the upstream server.

Leave expiry time and TTL blank, which means that the message will stay forever.

4. Start to move messages.

Connect or move the publisher or consumer to the new cluster. If the cluster is migrating, moving can be done in any order. The federated queue will only retrieve messages when it has run out of messages locally, when it has consumers that need messages, or when the upstream queue has "spare" messages that are not being consumed.

5. Verify that messages are federated

Verify that the downstream server consumes the messages published to the queue at the upstream server when there are consumers available at the downstream server.

RABBITMQ BEST PRACTICE

These are the RabbitMQ Best Practice recommendations based on the experience CloudAMQP have gained while working with RabbitMQ. This includes information on how to configure a RabbitMQ cluster for optimal performance and how to configure it to get the most stable cluster. Queue size, common mistakes, pre-fetch values, connections and channels, and number of nodes in a cluster are some of the things discussed.

QUEUES

Performance optimizations of RabbitMQ queues.

Use quorum queues.

Quorum queues aim to resolve both the performance and the synchronization failings of mirrored queues. Using a variant of the Raft protocol, which has become the industry defacto distributed consensus algorithm, quorum queues are both safer and achieve higher throughput than mirrored queues.

Use Quorum Queues

Use Quorum Queues in favour of classic mirrored queues.

Keep queues short, if possible

Keep queues as short as possible. A message published to an empty queue will go straight to the consumer as soon as the queue receives it (a persistent message in a durable queue will go to disk). The recommendation is to keep fewer than 10,000 messages in one queue.

Many messages in a queue can put a heavy load on RAM usage. In order to free up RAM, RabbitMQ starts flushing (page out) messages to disk. The page out process usually takes time and blocks the queue from processing messages when there are many messages to page out. A large amount of messages might have a negative impact on the broker since the process deteriorates queuing speed.

Keep queues short

Short queues are the fastest. A message in an empty queue will go straight out to the consumer as soon as the queue receives the message.

It is time-consuming to restart a cluster with many messages because the index must be rebuilt. It takes time to sync messages between nodes in the cluster after a restart.

Enable lazy queues to get predictable performance (for RabbitMQ version < 3.12)

Queues can become long for various reasons; consumers might crash, or be offline due to maintenance, or they might simply be working slower than usual. Lazy queues are able to support long queues with millions of messages. Lazy queues, introduced in RabbitMQ version 3.6, write messages to disk immediately, thus spreading the work out over time instead of taking the risk of a performance hit somewhere down the line. It provides a more predictable, smooth performance without any sudden drops, but at a cost. Messages are only loaded into memory when needed, thereby minimizing the RAM usage, but increasing the throughput time.

Starting with RabbitMQ version 3.12, all classic queues behave similarly to lazy queues.

Limit queue size with TTL or max-length

Another recommendation for applications that often get hit by spikes of messages, and where throughput is more important than anything else, is to set a max-length on the queue. This keeps the queue short by discarding messages from the head of the queue so that it never becomes larger than the max-length setting.

Number of queues

Queues are single-threaded in RabbitMQ. Better throughput is achieved on a multi-core system if multiple queues and multiple consumers are used. Optimal throughput is achieved with as many queues as cores on the underlying node(s).

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster. This might slow down the server if there are thousands upon thousands of active queues and consumers. The CPU and RAM usage may also be affected negatively with the use of too many queues.

Split queues over different cores

Queue performance is limited to one CPU core or hardware thread because a queue is single threaded. Better performance is achieved if queues are split over different cores and into different nodes when using a RabbitMQ cluster. Routing messages to multiple queues results in a much higher overall performance.

RabbitMQ queues are bound to the node where they are first declared. All messages routed to a specific queue will end up on the node where that queue resides. It is possible to manually split queues evenly between nodes, but the downside is having to keep track of where each queue is located.

Two plugins that help if there are multiple nodes or a single node cluster with multiple cores are the consistent hash exchange plugin and RabbitMQ sharding.

Use multiple queues and consumers

You achieve optimal throughput if you have as many queues as cores on the underlying node(s).

The consistent hash exchange plugin

The consistent hash exchange plugin allows for use of an exchange to load balance messages between queues. Messages sent to the exchange are distributed consistently and equally across many queues based on the routing key of the message. The plugin creates a hash of the routing key and spreads the messages out between queues that have a binding to that exchange. Performing this manually would be almost impossible.

The consistent hash exchange plugin is used to get maximum use of many cores in the cluster. Note that it is important to consume from all queues. Read more about the [consistent hash exchange plugin](#).

RabbitMQ sharding

The RabbitMQ sharding plugin partitions queues automatically after an exchange is defined as sharded. The supporting queues are then automatically created on every cluster node and messages are sharded across them. RabbitMQ sharding shows one queue to the consumer, but many queues could be running behind it in the background. The RabbitMQ sharding plugin is a centralized place to send messages with a goal of load balancing through the addition of queues to the other nodes in the cluster. Read more about [RabbitMQ sharding](#).

Don't set names on temporary queues

Setting a queue name is important when sharing the queue between producers and consumers, but it is not when using temporary queues. Instead, allow the server to choose a random queue name, or modify the RabbitMQ policies.

A queue name starting with `amq.` is reserved for internal use by the broker.

Auto-delete unused queues

Client connections can fail and potentially leave unused queues behind. Leaving too many queues behind might affect the performance of the system. There are ways to have queues deleted automatically.

The first option is to set a **TTL (time-to-live) policy** on the queue. For example, a TTL policy of 28 days will delete queues that have not had messages consumed from them in the last 28 days.

Another option is an **auto-delete** queue, which is a queue that gets deleted when its last consumer has canceled or when the channel/connection is closed (or when it has lost the TCP connection with the server).

Finally, an **exclusive queue** which is only used (consumed from, purged, deleted, etc.) by its declaring connection. Exclusive queues are deleted when their declaring connection is closed or gone due to underlying TCP connection loss or other circumstances.

Set limited use on priority queues

Queues can have zero or more priority levels. Keep in mind that each priority level uses an internal queue on the Erlang VM, which means that it takes up resources. In most use cases, it is sufficient to have no more than five priority levels, which keeps resource use manageable.

Payload - RabbitMQ Message Size and Type

How to handle the payload size of messages sent to RabbitMQ is a common question asked by developers. The answer is, of course, to avoid sending very large files in messages, but also keep in mind that the rate of messages per second can be a larger bottleneck than the message size itself. Sending multiple small messages might be a bad alternative. The better approach could be to bundle the small messages into one larger message and let the consumer split it up. However, bundling multiple messages might affect the processing time. If one of the bundled messages fails, will all of them need to be reprocessed? Bandwidth and architecture will dictate the best way to set up messages queues with consideration to payload.

CONNECTIONS AND CHANNELS

Each connection uses about 100 KB of RAM (and even more, if TLS is used). Thousands of connections can be a heavy burden on a RabbitMQ server. In a worst case scenario, the server can crash due to running out of memory. Try to keep connection/channel count low, and avoid connection and channel leaks.

Don't use too many connections or channels

Try to keep connection/channel count low.

Don't share channels between threads

Most clients don't make channels thread-safe because it would have serious negative impact on performance, which is why sharing channels between threads is not recommended.

Don't share channels between threads

You should make sure that you don't share channels between threads as most clients don't make channels thread-safe.

Don't open and close connections or channels repeatedly

Don't open and close connections or channels repeatedly, as doing so will create a higher latency because more TCP packages have to be sent and received.

The handshake process for an AMQP connection is actually quite involved and requires at least seven TCP packets (more if TLS is used). The AMQP protocol has a mechanism called channels that "multiplexes" a single TCP connection. It is recommended that each process only create one TCP connection with multiple channels in that connection for different threads. Connections should be long-lived so that channels can be opened and closed more frequently, if required. Even channels should be long-lived if possible. Do not open a channel every time a message is published. Best practice includes the re-use of connections and multiplexing a connection between threads with channels, when possible.

- AMQP connections: 7 TCP packages
 - AMQP channel: 2 TCP packages
 - AMQP publish: 1 TCP package (more for larger messages)
 - AMQP close channel: 2 TCP packages
 - AMQP close connection: 2 TCP packages
- » Total 14-19 packages (+ Acks)

Don't open and close connections or channels repeatedly

Repeatedly opening and closing channels means higher latency,
as more TCP packages have to be sent and received.

Separate connections for publisher and consumer

Unless the connections are separated between publisher and consumer, messages may not be consumed. This is especially true if the connection is in flow control, which will constrict the message flow even more.

Another thing to keep in mind is that RabbitMQ may cause back pressure on the TCP connection when the publisher is sending too many messages to the server. When consuming on the same TCP connection, the server might not receive the message acknowledgments from the client, affecting the performance of message consumption and the overall server speed.

Separate connections for publisher and consumer

For the highest throughput, separate the connections for publisher and consumer.

RABBITMQ MANAGEMENT INTERFACE PLUGIN

Another effect of having a large number of connections and channels is that the performance of the RabbitMQ management interface will slow down. Metrics have to be collected, analyzed and displayed for every connection and channel, which consumes server resources.

ACKNOWLEDGMENTS AND CONFIRMS

Messages in transit might get lost in an event of a connection failure and may need to be retransmitted. Acknowledgments let the server and clients know when to retransmit messages. The client can either `acknowledge` the message when it receives it, or when the client has completely processed the message. Acknowledgment has a performance impact, so for the fastest possible throughput, manual acks should be disabled.

A consuming application that receives essential messages should not acknowledge messages until it has finished whatever it needs to do with them so that unprocessed messages (worker crashes, exceptions, etc.) do not go missing.

Publish confirm is the same but for the publisher; the broker acks when it has received a message from a publisher. Publish confirm also has a performance impact, but keep in mind that it's required if the publisher needs messages to be processed at least once.

UNACKNOWLEDGED MESSAGES

All unacknowledged messages must reside in RAM on the servers. Too many unacknowledged messages will eventually use all system memory. An efficient way to limit unacknowledged messages is to limit how many messages the clients pre-fetch. Read more about this in the pre-fetch section.

PERSISTENT MESSAGES AND DURABLE QUEUES

To prepare for broker restarts, broker hardware failure, or broker crashes, use persistent messages and durable queues to ensure that they are on disk. Messages, exchanges and queues that are not durable and persistent are lost during a broker restart.

Queues should be declared as *durable* and messages should be sent with delivery mode *persistent*.

Persistent messages are heavier as they have to be written to disk. Similarly, lazy queues have the same effect on performance even though they are transient messages. For high performance, use transient messages and for high throughput, use temporary or non-durable queues.

Use persistent messages and durable queues

If you cannot afford to lose any messages, make sure that your queue is declared as "durable", and your messages are sent with delivery mode "persistent" (delivery_mode=2).

TLS AND AMQPS

Connecting to RabbitMQ over AMQPS is the AMQP protocol wrapped in TLS. TLS has a performance impact since all traffic must be encrypted and decrypted. For maximum performance, use VPC peering instead, which encrypts the traffic without involving the AMQP client/server.

CloudAMQP configures the RabbitMQ servers so that they accept and prioritize fast but secure encryption ciphers.

PRE-FETCH

The pre-fetch value is used to specify how many messages are consumed at the same time. It is used to get as much out of the consumers as possible.

The RabbitMQ default pre-fetch setting gives clients an unlimited buffer, meaning that RabbitMQ by default sends as many messages as it can to any consumer that looks ready to accept them. Messages that are sent are cached by the RabbitMQ client library in the consumer until processed. A typical mistake is to have an unlimited pre-fetch, where one client receives all messages and runs out of memory and crashes, and then all messages are re-delivered.

Don't use an unlimited pre-fetch value

One client could receive all messages and then run out of memory.

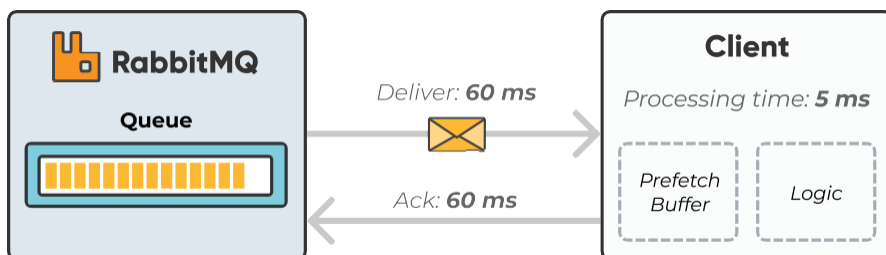


Figure 51 - Too small prefetch count may hurt performance.

A pre-fetch count that is too small may hurt performance, as RabbitMQ is typically waiting to get permission to send more messages. Figure 37 illustrates long idling time. In the example, we have a QoS pre-fetch setting of one (1). This means that RabbitMQ will not send out the next message until after the round trip completes (deliver, process, acknowledge). Round trip time in this picture is in total 125ms with a processing time of only 5ms.

A too large pre-fetch count, on the other hand, could deliver many messages to one single consumer, and keep other consumers in an idling state, as illustrated in Figure 52.

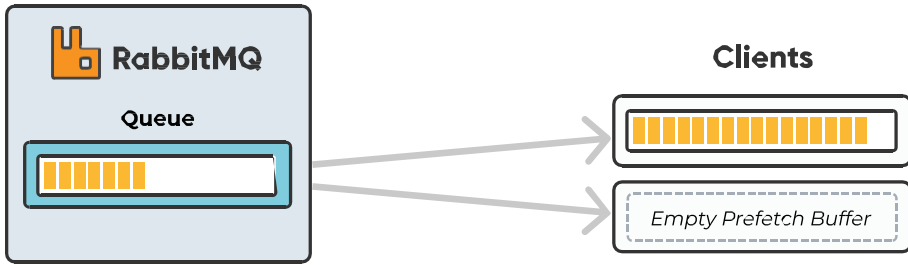


Figure 52 - A large prefetch count could deliver lots of messages to one single consumer.

Please note that if a client is set up to auto-ack messages, the pre-fetch value has no effect.

The pre-fetch value will have no effect if the client is set to auto-ack messages.

HIPE

Enabling HiPE increases server throughput at the cost of increased startup time. Enabling HiPE means that RabbitMQ is compiled at startup. The throughput increases from 20 to 80 percent according to benchmark tests. The drawback of HiPE is the startup time increases by about 1-3 minutes. HiPE is still marked as experimental in RabbitMQ's documentation.

CLUSTERING AND HIGH AVAILABILITY

Single node setup

Creating a CloudAMQP instance with one node results in one single, high-performance node, as the messages are not between multiple nodes. All data that is written to disk is safe, but if you use transient messages or non-durable queues you might lose messages if there's a hardware failure and the node has to be restarted. In order to avoid losing messages in a single node broker, you should be prepared for broker restarts, broker hardware failure, or broker crashes. To ensure that messages and broker definitions survive restarts, ensure that they are on the disk. Messages, exchanges, and queues that are not durable and persistent will be lost during a broker restart.

If you cannot afford to lose any messages, make sure that your queue is declared as

durable and that messages are sent with delivery mode *persistent*. The messages will be saved to disk and everything will be intact when the node comes back up again.

Cluster setup with at least 3 nodes

A CloudAMQP cluster with three nodes gives you three RabbitMQ servers. These servers are placed in different zones (availability zones in AWS) in all data centers with support for zones. The default quorum queue cluster size is set to the same value as the number of nodes in the cluster. This argument can be overridden with the queue argument *quorum_cluster_size*. Each quorum queue is replicated; it has a leader and multiple followers.

A quorum queue with a replication factor of five will consist of five replicated queues: the leader and four followers. Each replicated queue will be hosted on a different node (broker).

REMEMBER TO ENABLE HA ON NEW VHOSTS

A common mistake on CloudAMQP clusters is that users create a new vhost but forget to enable an HA-policy for it. Messages will therefore not be synced between nodes.

ROUTING (EXCHANGES SETUP)

Direct exchanges are the fastest to use because multiple bindings mean that RabbitMQ must calculate where to send the message.

PLUGINS

Some plugins might consume a lot of CPU or use a large amount of RAM, which makes them less than ideal on a production server. Disable plugins that are not being used via the control panel in CloudAMQP.

Disable plugins that you are not using.

STATISTICS RATE MODE

The RabbitMQ management interface collects and stores stats for all queues, connections, channels, and more, which might affect the broker in a negative way if, for example, there are too many queues. Avoid setting the RabbitMQ management statistics rate to 'detailed' as it could affect performance.

Don't set management statistics rate mode as detailed.

RABBITMQ, ERLANG AND CLIENT LIBRARIES

Stay up-to-date with the latest stable versions of RabbitMQ and Erlang. CloudAMQP extensively tests new major versions before release. Therefore, the most recommended version is the default in the dropdown menu for a new cluster.

Use the latest recommended version of client libraries and check the documentation or inquire directly with any questions regarding which library to use.

Don't use old RabbitMQ versions or RabbitMQ clients

Stay up-to-date with the latest stable versions of RabbitMQ and Erlang. Make sure that you are using the latest recommended version of client libraries.

DEAD LETTERING

A queue that is declared with the `x-dead-letter-exchange` property sends messages that are either rejected, nacked (negative acknowledged), or expired (with TTL) to the specified dead letter exchange. Remember that the specified `x-dead-letter-routing-key` in the routing key of the message will change when the message becomes dead lettered.

TTL

Declaring a queue with the `x-message-ttl` property means that messages will be discarded from the queue if they haven't been consumed within the time specified.

BEST PRACTICES FOR HIGH PERFORMANCE

This section is a summary of recommended configurations for high-performance to maximize the message passing throughput in RabbitMQ.

Keep queues short

For optimal performance, keep queues as short as possible all the time. Longer queues impose more processing overhead. Queues should always stay around zero (0) for optimal performance.

Set max-length if needed

A feature recommended for applications that often receive message spikes is setting a max-length. This keeps the queue short by discarding messages from the head of the queues to keep it no larger than the max-length setting.

Use transit messages

Use transit messages to avoid lowered throughput caused by persistent messages, which are written to disk as soon as they reach the queue.

Use multiple queues and consumers

Queues are single-threaded in RabbitMQ, and one queue can handle up to about 50k messages. Better throughput on a multi-core system is achieved through use of multiple queues and consumers.

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster, which may slow down the server if thousands upon thousands of active queues and consumers are on the system.

Split queues over different cores

Better performance is achieved by splitting queues into different cores and into different nodes if possible, as queue performance is limited to one CPU core.

Two plugins are recommended that will help systems cope with multiple nodes or a single node cluster with multiple cores; the consistent hash exchange plugin and the RabbitMQ sharding plugin.

Disable manual acks and publish confirms

Acknowledgment and publish confirms both have an impact on performance. For the fastest possible throughput, manual acks should be disabled, which will speed up the broker by allowing it to "fire and forget" the message.

Avoid multiple nodes (HA)

One node gives the highest throughput when compared to an HA cluster setup. Messages and queues are not mirrored to other nodes.

Disable unused plugins

Some plugins might be great, but they also consume a lot of CPU or use a high amount of RAM. Because of this, they are not recommended for a production server. Disable unused plugins.

BEST PRACTICES FOR HIGH AVAILABILITY

This section is a summary of recommended configurations for high availability or up-time for the RabbitMQ cluster.

Quorum Queues

Use of Quorum Queues instead of classic mirrored queues.

Keep queues short

For optimal performance, keep queues short whenever possible. Longer queues impose more processing overhead, so keep queues around zero (0) for optimal performance.

Enable lazy queues

Lazy queues write messages to disk immediately, spreading the work out over time instead of risking a performance hit somewhere down the line. The result of using a lazy queue is a more predictable, smooth performance curve without sudden drops.

RabbitMQ HA – Two (2) nodes

Enhancing the availability of data by using replicas means that clients can find messages even through system failures. Two (2) nodes are optimal for high availability, and CloudAMQP locates each node in a cluster in different availability zones (AWS). Additionally, queues are automatically mirrored and replicated (HA) between availability zones. Message queues are by default located on one single node but they are visible and reachable from all nodes.

When a node fails, the auto-failover mechanism distributes tasks to other nodes in the cluster. RabbitMQ instances include a load balancer, which makes broker distribution transparent from the message publishers. Maximum failover time in CloudAMQP is 60s (the endpoint health is measured every 30s, and the DNS TTL is set to 30s).

Optional federation use between clouds

Clustering is not recommended between clouds or regions, and therefore there is no plan at CloudAMQP to spread nodes across regions or datacenters. If one cloud region goes down, the CloudAMQP cluster also goes down, but this scenario would be extremely rare. Instead, cluster nodes are spread across availability zones within the same region.

To protect the setup against a region-wide outage, set up two clusters in different regions and use federation between them. Federation is a method in which a software system can benefit from having multiple RabbitMQ brokers distributed on different machines.

Send persistent messages to durable queues

In order to avoid losing messages in the broker, make sure they are on disk by declaring the queue as "durable" and sending messages with delivery mode as "persistent". Messages, exchanges, and queues that are not durable and persistent are lost during a broker restart, hardware failure, and crashes.

Management statistics rate mode

Setting the rate mode of RabbitMQ management statistics to 'detailed' has a serious impact on performance. The detailed setting is not recommended in production.

Limited use of priority queues

Each priority level uses an internal queue on the Erlang VM, which consumes resources. In most use cases, it is sufficient to have no more than five (5) priority levels.

RABBITMQ PROTOCOLS

RabbitMQ is an open source multi-protocol messaging broker. This means that RabbitMQ supports several messaging protocols over a range of different open and standardized protocols such as AMQP, HTTP, STOMP, MQTT, and WebSockets/Web-Stomp.

Message queueing protocols have features in common, so choosing the right one comes down to the use case or scenario. In the simplest case, a message queue uses an asynchronous protocol in which the sender and the receiver do not operate on the message at the same time.

The protocol defines the communication between the client and the server and has no impact on the message itself. One protocol can be used when publishing while another can be used to consume. The MQTT protocol, with its minimal design, is perfect for built-in systems, mobile phones, and other memory and bandwidth sensitive applications. While using AMQP for the same task will work, MQTT is a more appropriate choice of protocol for this specific type of scenario.

When creating a CloudAMQP instance, all the common protocols are available by default (exception: WebSockets/Web-Stomp is only enabled on dedicated plans).

AMQP

RabbitMQ was originally developed to support AMQP which is the "core" protocol supported by the RabbitMQ broker. AMQP stands for Advanced Message Queueing Protocol and it is an open standard application layer protocol. RabbitMQ implements version 0.9.1 of the specification today, with legacy support for version 0.8 and 0.9. AMQP was designed for efficient support of a wide variety of messaging applications and communication patterns. AMQP is a more advanced protocol than MQTT, is more reliable, and has better security support. AMQP also has features such as flexible routing, durable and persistent queues, clustering, federation, and high availability queues. The downside is that it is a more verbose protocol depending on solution implementation.

As with other message queueing protocols, the defining features of AMQP are message orientation and queueing. Routing is another feature, which is the process by which an exchange decides which queues to place messages on. Messages in RabbitMQ are routed from the exchange to the queue depending on exchange types and keys. Reliability and security are other important features of AMQP. RabbitMQ can be configured to ensure that messages are always delivered. Read more in the [Reliability Guide](#).

For more information about AMQP, check out the AMQP Working Group's overview page.

CloudAMQP AMQP assigned port number is 5672 or 5671 for AMQPS (TLS/SSL encrypted AMQP).

MQTT

MQ Telemetry Transport is a publish-subscribe, pattern-based "lightweight" messaging protocol. The protocol is often used in the IoT (Internet of Things) world of connected devices. It is designed for built-in systems, mobile phones, and other memory and bandwidth-sensitive applications.

MQTT benefits for IoT make a difference for extremely low-power devices. MQTT is very code-footprint efficient and strongly focuses on using minimal bandwidth. Implementing MQTT on a client requires less resources than AMQP because of its simplicity.

Native support for MQTT did not exist before version 3.12. To support MQTT in earlier versions, RabbitMQ instead accepted messages from the MQTT plugin. It then forwards these messages to its core protocol, AMQP 0.9.1. In other words, it only proxied MQTT messages via its core protocol. While this approach provided a simple way to extend RabbitMQ beyond AMQP, it has some performance limitations.

Fortunately, this has changed drastically in RabbitMQ 3.12. In later versions of RabbitMQ, the MQTT and Web MQTT plugins had been rewritten to add native support for the MQTT protocol in RabbitMQ.

With this improvement, the MQTT and Web MQTT plugins parse MQTT messages and send them directly to the appropriate queues instead of forwarding them to queues via the AMQP 0.9.1 protocol. This, in turn, had led to some substantial performance improvements.

CloudAMQP MQTT assigned port number is 1883 (8883 for TLS wrapped MQTT). Use the same default username and password as for AMQP.

STOMP

STOMP, Simple (or Streaming) Text Oriented Message Protocol, is a simple text-based protocol used for transmitting data across applications. It is a less complex protocol than AMQP, with more similarities to HTTP. STOMP clients can communicate with almost every available STOMP message broker, which provides easy and widespread messaging interoperability among many languages, platforms, and brokers. It is, for example, possible to connect to a STOMP broker using a telnet client.

STOMP does not deal with queues and topics. Instead, it uses a SEND semantic with a destination string. RabbitMQ maps the message to topics, queues or exchanges, and consumers then SUBSCRIBE to those destinations. Other brokers might map onto something else understood internally.

STOMP is recommended when implementing a simple message queueing application without complex demands on a combination of exchanges and queues. RabbitMQ supports all current versions of STOMP via the STOMP plugin.

CloudAMQP STOMP assigned port number is 1883, 61613 (61614 for TLS wrapped STOMP).

HTTP

HTTP stands for Hypertext Transfer Protocol, an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP is not a messaging protocol. However, RabbitMQ can transmit messages over HTTP.

The RabbitMQ-management plugin provides an HTTP-based API for management and monitoring of the RabbitMQ server.

CloudAMQP HTTP assigned port number is 443.

PUBLISH WITH HTTP

Example of how to publish a message to the default exchange with the routing key "my_key":

```
curl -XPOST -d'{"properties":{}, "routing_key": "my_key", "payload": "my body", "payload_encoding": "string"}' https://username:password@hostname/api/exchanges/vhost/amq.default/publish
```

```
Response: {"routed": true}
```

Get message with HTTP

Example of how to get one message from the queue "your_queue" (not an HTTP GET as it will alter the state of the queue):

```
curl -XPOST -d'{"count": 1, "requeue": true, "encoding": "auto"}' https://user:pass@host/api/queues/your_vhost/your_queue/get
```

```
Response: Json message with payload and properties.
```

Get queue information

```
curl -XGET https://user:pass@host/api/queues/your_vhost/your_queue
```

```
Response: Json message with queue information.
```

Autoscale by polling queue length

When jobs are arriving faster in the queue than they are processed, and when the queue starts growing in length, it's a good idea to spin up more workers. By polling the HTTP API queue length, you can spin up or take down workers depending on the length.

WEB-STOMP

Web-Stomp is a plugin to RabbitMQ which exposes a WebSockets server (with fallback) so that web browsers can communicate with your RabbitMQ server/cluster directly.

To use Web-Stomp you first need to create at least one user, with limited permissions, or a new vhost which you can expose publicly because the username/password must be included in your javascript, and a non-limited user can subscribe and publish to any queue or exchange.

The Web-Stomp plugin is only enabled on dedicated plans on CloudAMQP.

Next includesocks.min.js and stomp.min.js in your HTML from for example CDNJS:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js"></script>
```

```
// Replace with your hostname
```

```
var wss = new WebSocket("wss://blue-horse.rm.cloudamqp.com/ws/");
```

```
var client = Stomp.over(wss);
```

```
// RabbitMQ SockJS does not support heartbeats so disable them
```

```
client.heartbeat.outgoing = 0;
```

```
client.heartbeat.incoming = 0;
```

```
client.debug = onDebug;
```

```
// Make sure the user has limited access rights
```

```
client.connect("webstomp-user", "webstomp-password", onConnect, onError, "vhost");
```

```
function onConnect() {
```

```
    var id = client.subscribe("/exchange/web/chat", function(d) {
```

```
        var node = document.createTextNode(d.body + '\n');
```

```
        document.getElementById("chat").appendChild(node);
```

```
    });
```

```
}
```

```
function sendMsg() {
```

```
    var msg = document.getElementById("msg").value;
```

```
    client.send("/exchange/web/chat", { "content-type": "text/plain" }, msg);
```

```
}
```

```
function onError(e) {  
  console.log("STOMP ERROR", e);  
}
```

```
function onDebug(m) {  
  console.log("STOMP DEBUG", m);  
}
```