FOURTH EDITION

THE OPTIMAL RABBITMQ GUIDE

FROM BEGINNER TO ADVANCED

BY LOVISA JOHANSSON

OVER 65,000
READERS & COUNTING

THE OPTIMAL RABBITMQ GUIDE

From Beginner to Advanced

I want to say a big thank you to everyone who has helped me, from the earliest draft of this book, up to this edition. A special thanks go out to my lovely colleagues at 84codes and my tech friends.

Finally, a big thank you to all CloudAMQP users for your feedback and continued support. It has been a great motivator to see our customers' projects succeed!

- Lovisa Johansson, CloudAMQP

The Optimal RabbitMQ Guide

Book version: 4.0

Author: Lovisa Johansson

Email: lovisa@cloudamqp.com

Co-author: Elin Vinka, Sofie Abrahamsson, Nyior Clement

Graphics: Daniel Marklund, Elin Vinka, Magnus Lindberg

Published: 2023-09-15

ISBN: 978-91-987951-2-7

I'd love to hear from you!

Please e-mail me with any comments that you might have about the book and let me know what you think should or shouldn't be included in the next edition. If you have an application using RabbitMQ or a user story that you would like to share, please send me an e-mail at lovisa@84codes.com.

PART ONE

PART TWO

PART THREE

This book is dedicated to all the current and future users of RabbitMQ, and to the Swedish mentality that makes us "queue lovers" at heart.

INTRODUCTION

Today's demand for a better, faster techology means that reliability and scalability are more important than ever. Companies are being forced to rethink their architecture. Monoliths are evolving into microservices and servers are moving into the cloud. Message Queues and RabbitMQ in particular, as one of the most widely deployed open-source message brokers, have come to play a significant role in the growing world of microservices. This book is divided into three parts:

The first part is an introduction to microservices and RabbitMQ, including a section on how to use RabbitMQ on CloudAMQP. This section includes the most important RabbitMQ concepts and how it allows users to create products that meet current and future industry demands.

The second part is for more advanced users, who will learn how to take full advantage of RabbitMQ. This section explores Best Practice recommendations for High Performance and High Availability and common RabbitMQ errors and mistakes. It is also a deep dive into some RabbitMQ concepts and features.

The third part gives some real-world user stories from our own experiences with RabbitMQ as well from a CloudAMQP customer's point of view.

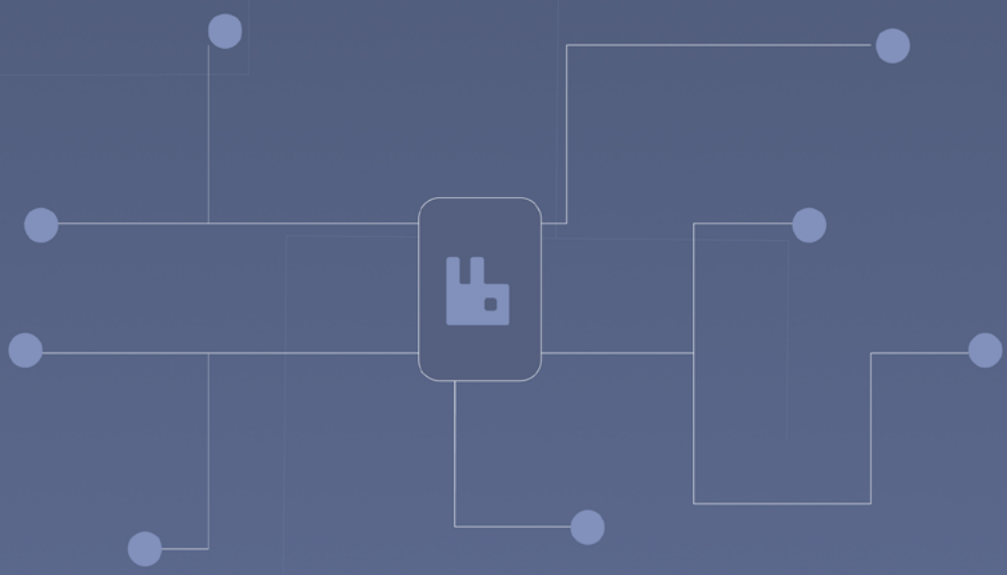
We hope that this book takes you far on your message queueing journey with RabbitMQ.

PART ONE

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.

PART ONE

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.

PART ONE

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.

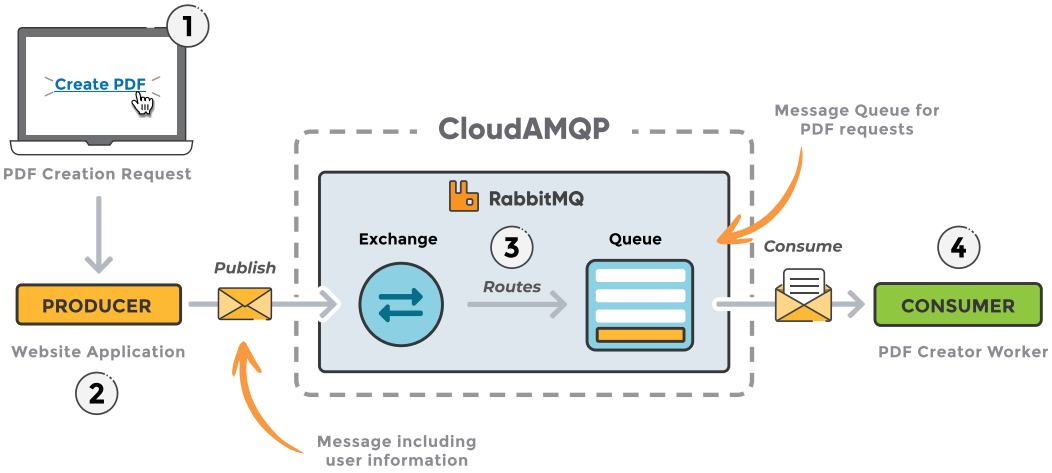


***Figure 4** - Application architecture example with RabbitMQ.*

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.

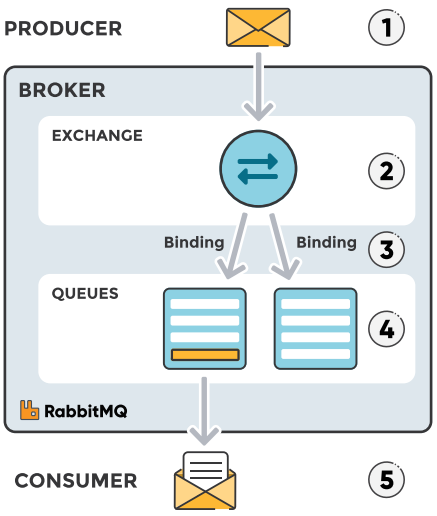


Figure 5 *- Illustration of the message flow in RabbitMQ*

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.

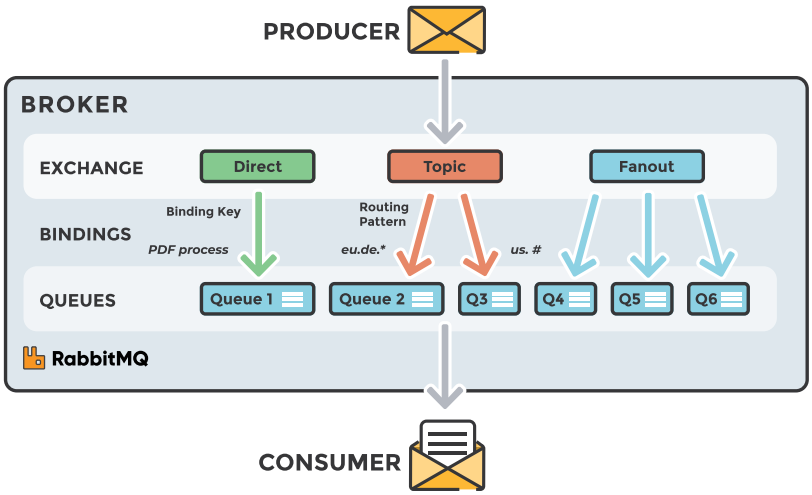


Figure 6 - Three different exchanges: direct, topic, and fanout.

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

To be able to follow this guide, set up a CloudAMQP instance or set up RabbitMQ on a local machine. CloudAMQP is a hosted RabbitMQ solution (RabbitMQ as a Service), meaning that you can sign up for an account and create an instance of RabbitMQ.

There is no need to set up and install RabbitMQ or care about cluster handling, as CloudAMQP handles everything for you. A RabbitMQ instance is available in many data- centers and regions, and is available for free with the plan Little Lemur. Go to the CloudAMQP plan page to sign up for an appropriate plan. Click on details in the cloud-hosted RabbitMQ instance to find the username, password, and con- nection URL (Figure 8).

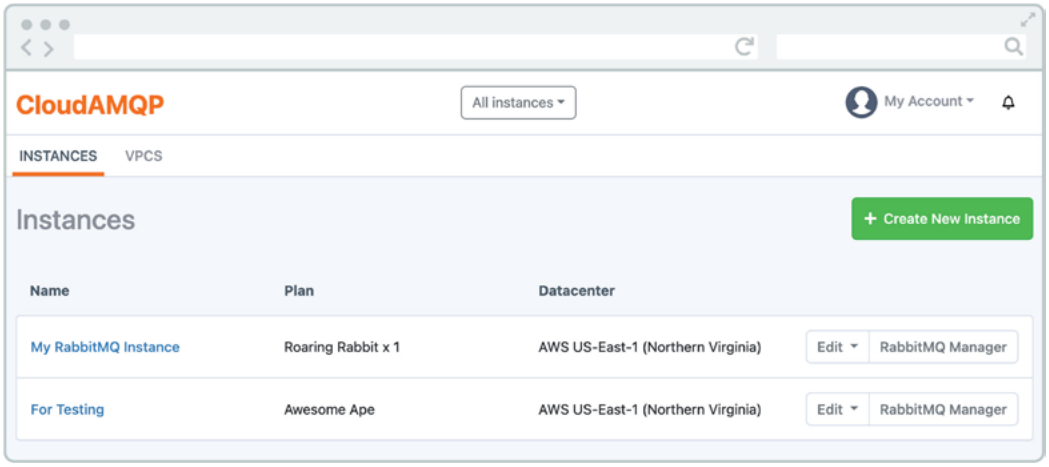


***Figure 7** - Instances in the CloudAMQP web interface.*

Getting Started with RabbitMQ

With a running RabbitMQ instance, you can easily send messages across various languages, platforms, and operating systems. Start by accessing the user-friendly management interface which gives you a comprehensive overview of your RabbitMQ server.

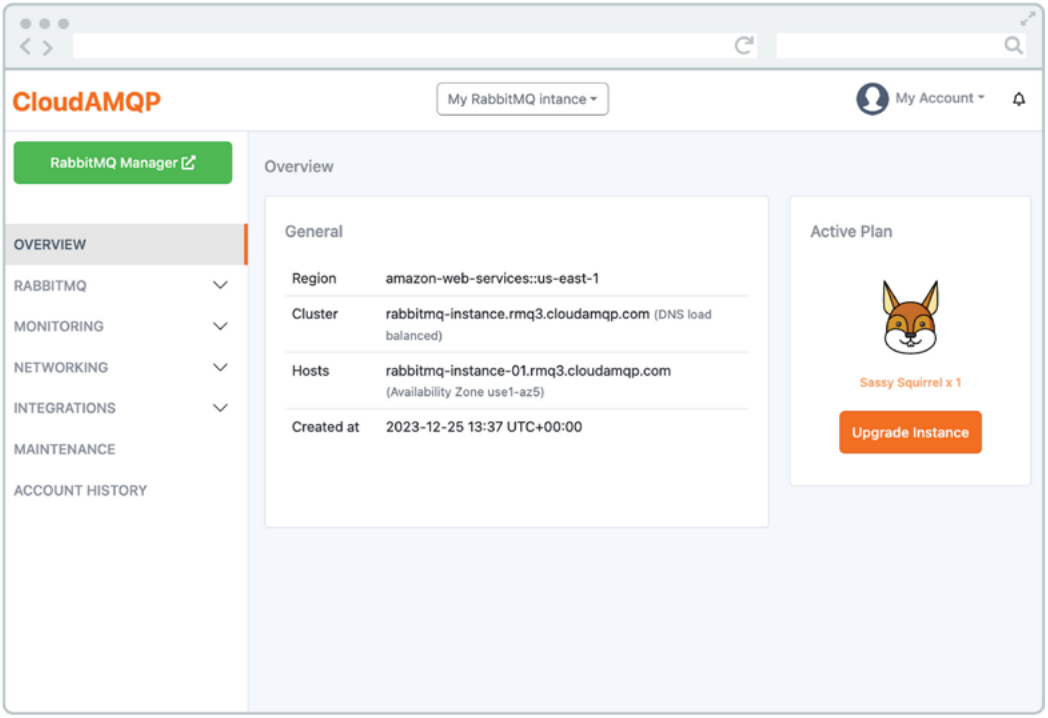


***Figure 8** - Detailed information of an instance in the CloudAMQP Console.*

The Management Interface

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

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*.

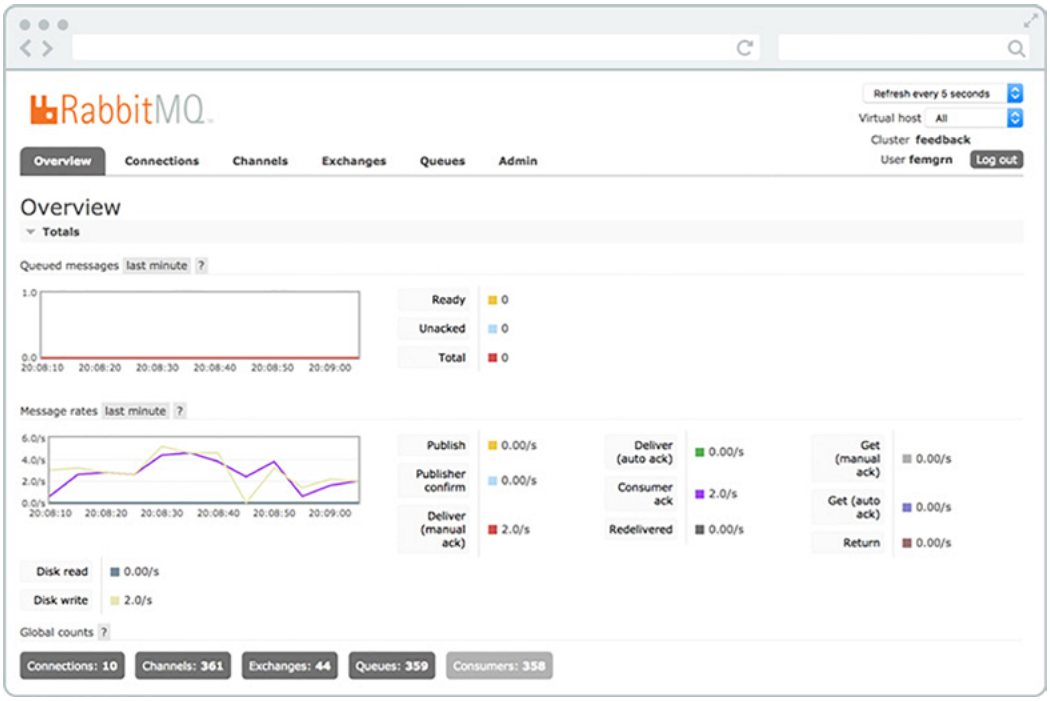


***Figure 9** - The overview window in the RabbitMQ management interface.*

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.

Sample Code

Sample code for Ruby, Node.js, and Python can be found in upcoming chapters. Remember, different programming languages can be used in different parts of the system. The publisher could be written in Node.js while the subscriber is written in Python, for example.

Hint: Separate Projects and Environments Using Vhosts

Just as it's possible to create different databases within a PostgreSQL (database) server for different projects, vhost makes it possible to separate applications on one single broker. Isolate users, exchanges, queues, etc. to one specific vhost or separate environments. For example, organize production to one vhost and staging to another vhost within the same broker instead of setting up multiple brokers. The downside of using a single RabbitMQ server is that there is no resource isolation between vhosts. Shared plans on CloudAMQP are located on isolated vhosts.

PART ONE

EXCHANGES, ROUTING KEYS AND BINDINGS

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).

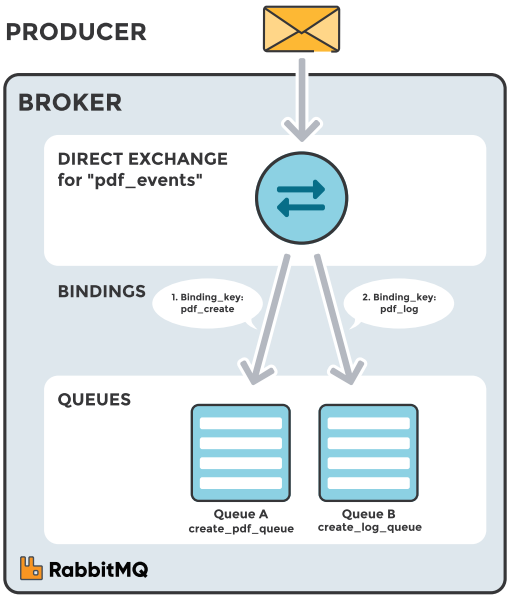


***Figure 10** – A message is directed to the queue where the binding key is an exact match of the message's routing key.*

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*).

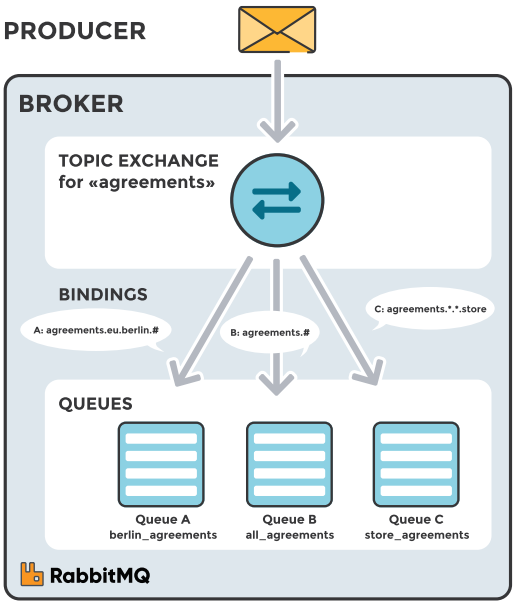


Figure 11 *- Messages are routed to one or many queues based on a match between a message routing key and the routing patterns.*

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.

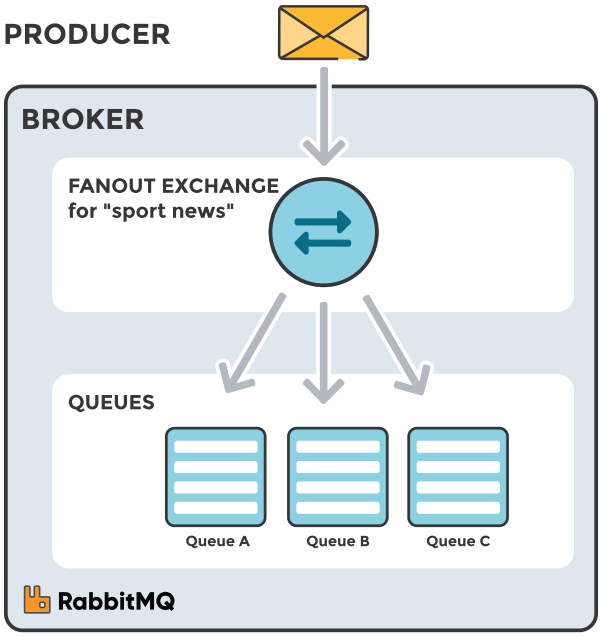


Figure 12 *- Fanout Exchange: The received message is routed to all queues that are bound to the exchange.*

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.

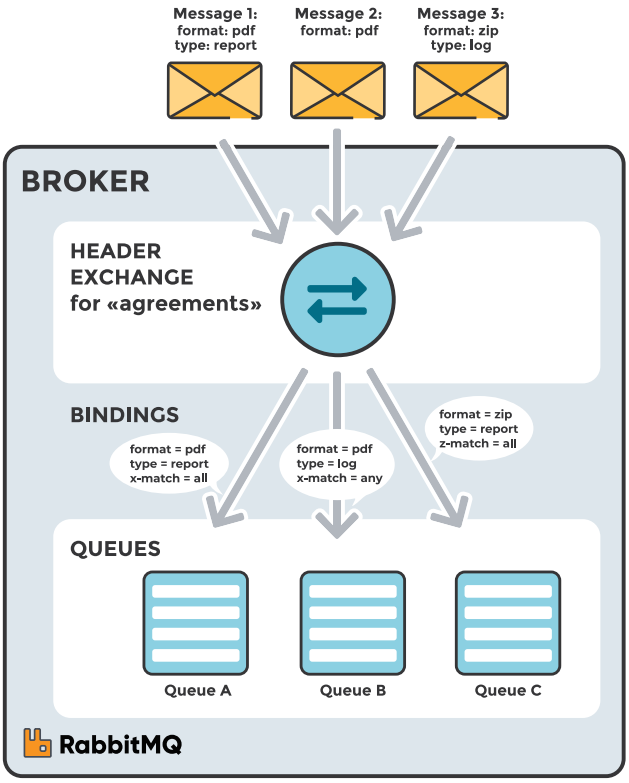


Figure 13 *- Headers exchange routes messages to queues that are bound using arguments (key and value) containing headers and optional values.*

- 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 it's 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.

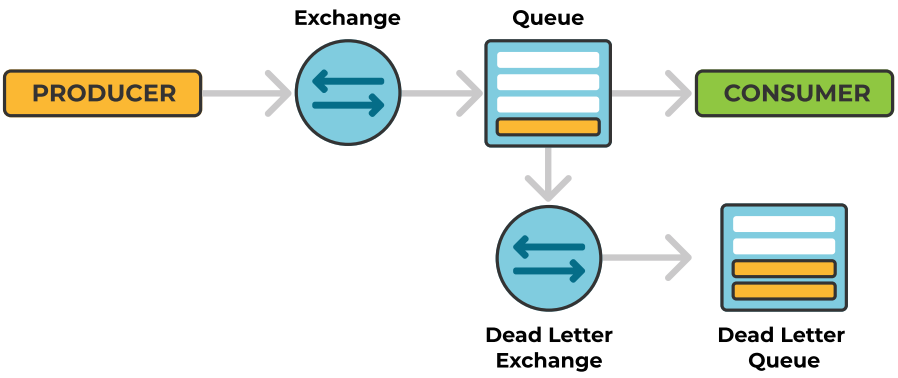


Figure 14 *- RabbitMQ Dead Letter Exchange and Dead Letter Queue.*

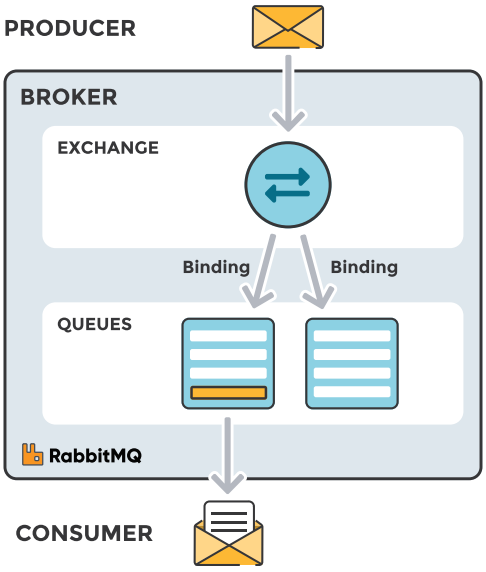
ALTERNATE EXCHANGE

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.

PART ONE

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.



PART ONE

RABBITMQ WITH RUBY AND AMQP::CLIENT

Ruby developers have some options for AMQP client libraries. In this example, AMQP::Client is used as an asynchronous client for publishing and consuming messages.

You need a RabbitMQ instance to get started. A free RabbitMQ instance can be set up for testing over at CloudAMQP, as described in Chapter 1. Once you have an instance, we add the environment variable CLOUDAMQP_URL to a .env file. You can find your CloudAMQP URL under Details in the CloudAMQP Console.

This code sample can be copied from https://training.cloudamqp.com

.ENV FILE

```
#Update with your own credentials

CLOUDAMQP_URL="amqps://xxxyyyyzzzz:xxx@server.rmq.cloudamqp.com/xxxyyyyzzzz"
```

Then, we add the following gems to our Gemfile (if they are not yet added).

GEMFILE

```
source 'https://rubygems.org'

gem 'amqp-client'

gem 'dotenv'
```

Now let's use bundler to install these libraries, open a terminal and execute the following command:

```
bundle install
```

(If you don't have bundler installed you can go ahead and install it by running `gem install bundler`. Or you may install each gem separately like so: `gem install amqp-client dotenv`)

We will now create one script for the **publisher** that publishes messages and a **consumer** script that consumes messages from the queue.

PUBLISHER.RB

```
require "amqp-client"

require "dotenv/load"

require "json"

# Opens and establishes a connection

connection = AMQP::Client.new(ENV['CLOUDAMQP_URL']).connect

# Open a channel

channel = connection.channel

puts "[✅] Connection over channel established"

# Create an exchange called "emails"

channel.exchange_declare("emails", "direct")

# Create a queue

channel.queue_declare("email.notifications")

# Bind the queue to the exchange: queue_bind(name, exchange, binding_key)

channel.queue_bind("email.notifications", "emails", "notification")

# Define a method for publishing messages to the queue

def send_to_queue(channel, routing_key, email, name, body)

  msg = "{#{email}, #{name}, #{body}}".to_json

  # Publish function expects: publish(body, exchange, routing_key)

  channel.basic_publish(msg, "emails", routing_key)

  puts "[📤] Message sent to queue #{msg}"

end

# Now lets send some messages to the queue over our channel

send_to_queue channel, "notification", "example@example.com", "John Doe", "Your order
has been received"

send_to_queue channel, "notification", "example@example.com", "Jane Doe", "The product
is back in stock"

send_to_queue channel, "resetpassword", "example@example.com", "Willem Dafoe", "Here
is your new password"

begin

  connection.close

  puts "[❎] Connection closed"

rescue => exception

  puts "Error: #{exception}"

end
```

Let's review the code snippet above. `AMQP::Client.new().connect` starts a connection. The connection is established once the channel is created `connection.channel`.

connection will hold the connection and channels will be set up in the connection, which is declared as channel for easy access. The method `send_to_queue()` will publish a message to the exchange named emails with a given routing key set in the call.

Next, let's add the logic that would handle the consumer side of things, which we add to a separate file.

CONSUMER.RB

```
require "amqp-client"
require "dotenv/load"
# Opens and establishes a connection, channel is created automatically
client = AMQP::Client.new(ENV['CLOUDAMQP_URL']).start
# Declare a queue
queue = client.queue "email.notifications"
counter = 0
# Subscribe to the queue
queue.subscribe() do |msg|
  counter += 1
  # Add logic to handle the message here...
  puts "[📥] Message received [#{counter}]: #{msg.body}"
  # Acknowledge the message
  msg.ack
rescue => e
  puts e.full_message
  msg.reject(requeue: false)
end
# Close the connection when the script exits
at_exit do
  client.stop
  puts "[❎] Connection closed"
end
# Keep the consumer running
sleep
```

Let's review the code snippet above. Here we use `AMQP::Client.new().start` which also sets up a channel for us.

`client.queue "email.notifications"` asserts the queue of interest. `queue.subscribe()` sets up a consumer that receives messages from the queue.

The `queue.subscribe() do |msg|` block is now receiving messages from the queue, in an attempt to process the information. This is where the logic would be placed for sending the actual email based on the content inside the message, which is not something we will go through in this example.

Finally, the `msg.ack` function sends an "ack" to the queue for the received message. Which tells the Queue that the message has been handled successfully and can be removed.

To run each script you can simply open up two terminals and navigate to the project folder. In the first terminal you can run:

```
ruby publisher.rb
```

Then in the other terminal you can start the consumer:

```
ruby consumer.rb
```

This is the expected output:

```
~/Sites/rabbitmq-ruby ruby publisher.rb
[✅] Connection over channel established
[📩] Message sent to queue "{example@example.com, John Doe, Your order has been received}"
[📩] Message sent to queue "{example@example.com, Jane Doe, The product is back in stock}"
[📩] Message sent to queue "{example@example.com, Willem Dafoe, Here is your new password}"
[❌] Connection closed
~/Sites/rabbitmq-ruby
```

```
~/Sites/rabbitmq-ruby ruby consumer.rb
[📩] Message received [1]: "{example@example.com, John Doe, Your order has been received}"
[📩] Message received [2]: "{example@example.com, Jane Doe, The product is back in stock}"
```

A FEW THINGS TO NOTE

The consumer.rb script keeps running until you close it, unlike publisher.rb. This means it can instantly consume any messages published to the queue. (To quit consumer.rb you can press CTRL+c in your terminal).

In reality it's best to keep the connection alive in both the publisher and consumer scripts, rather than opening and closing it repeatedly. This is because opening and closing connections is considered "expensive".

You might notice that the consumer only received two out of the three messages we sent. This happened because the last message was sent with a routing key that didn't match any of the consumer's bindings (`resetpassword`), causing it to be lost. To avoid losing messages, make sure that all messages are routed to a queue.

Finally, if you want to send an email for resetting a password, the consumer would need to perform additional logic such as connecting to the database and generating a reset key. A good way to handle these messages separately would be to add another queue that bind to the corresponding routing key.

ADDITIONAL TRAINING SUGGESTIONS

1. Set up an additional queue that binds to the `resetpassword` routing key and then create another consumer that handles these messages. Rerun the producer to make sure all messages are delivered this time.

2. One of the many advantages of RabbitMQ is decoupling. As an example you may have producers and consumers developed by different teams and in different pro- gramming languages, all communicating successfully via the message broker. Try sending a message from your existing producer and write code in another programming language that successfully consumes the message.

PART ONE

RABBITMQ AND NODE.JS WITH AMQP-CLIENT

Node.js developers have a number of options for AMQP client libraries. In this example, amqp-client.js is used.

You need a RabbitMQ instance to get started. A free RabbitMQ instance can be set up for testing over at CloudAMQP, as described in Chapter 1. Once you have an instance, we add the environment variable CLOUDAMQP_URL to a .env file. You can find your CloudAMQP URL under *Details* in the CloudAMQP Console.

This code sample can be copied from https://training.cloudamqp.com

.ENV

```
#Update with your own credentials

CLOUDAMQP_URL="amqps://xxxyyyyzzzz:xxx@server.rmq.cloudamqp.com/xxxyyyyzzzz"
```

First let's start a new NodeJS project by opening a terminal and run the command.

```
npm init
```

When asked, name your project (e.g. `nodejs-rabbitmq-demo`). You can leave the other questions unanswered/empty by pressing enter all the way.

Next add `@cloudamqp/amqp-client` and `dotenv` as a dependency to your `package.json` file. This can be done by running the following command in the terminal:

```
npm install --save @cloudamqp/amqp-client dotenv
```

In order for `amqp-client` to work you need to add `"type": "module"` to your package.json like so:

```
{
  "name": "nodejs-rabbitmq-demo",
  "type": "module",
  ...
}
```

We will now create one script for the publisher that publishes messages and a consumer script that consumes messages from the queue.

The publisher first establishes a connection between the RabbitMQ instance and the application, through a channel within the connection. A Queue and an Exchange are declared and created (if they do not already exist), a binding is created between the two, and, finally, a few messages are published.

The consumer also sets up a connection+channel and subscribes to the queue. The

messages are then handled one by one.

First, the full code for the publisher and the consumer is given. If you are following along, create a new js file and copy paste the code as we proceed.

PUBLISHER.JS

```
//Dependencies
import { AMQPClient } from '@cloudamqp/amqp-client'
import {} from 'dotenv/config'

async function startPublisher() {
  try {
    //Setup a connection to the RabbitMQ server
    const cloudAMQPURL = process.env.CLOUDAMQP_URL
    const connection = new AMQPClient(cloudAMQPURL)
    await connection.connect()
    const channel = await connection.channel()

    console.log("[✅] Connection over channel established")

    //Declare the exchange and queue, and create a binding between them
    await channel.exchangeDeclare('emails', 'direct')
    const q = await channel.queue('email.notifications')
    await channel.queueBind('email.notifications', 'emails', 'notification')

    //Publish a message to the exchange
    async function sendToQueue(routingKey, email, name, body) {
      const message = { email, name, body }
      const jsonMessage = JSON.stringify(message);

        //amqp-client function expects: publish(exchange, routingKey, message, options)
        await q.publish('emails', { routingKey }, jsonMessage)
        console.log("[📤] Message sent to queue", message)
    }
```

```
    //Send some messages to the queue

    sendToQueue("notification", "example@example.com", "John Doe", "Your order has
been received");

    sendToQueue("notification", "example@example.com", "Jane Doe", "The product is
back in stock");

    sendToQueue("resetpassword", "example@example.com", "Willem Dafoe", "Here is your
new password");


   setTimeout(() => {

      //Close the connection

      connection.close()

      console.log("[❎] Connection closed")

      process.exit(0)

    }, 500);

  } catch (error) {

    console.error(error)


    //Retry after 3 second

    setTimeout(() => {

      startPublisher()

    }, 3000)

  }

}

//Last but not least, we have to start the publisher and catch any errors

startPublisher()
```

Let's review the code snippet above. `new AMQPClient(...)` returns a connection instance. The connection is established once the channel is created `connection.channel()`.

The `startPublisher()` function call at the bottom will be the initiator of connecting to your RabbitMQ server. If the connection is closed or fails to be established, it will try to reconnect.

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

The `sendToQueue()` function will publish a message to the exchange named `emails` with a given routing key.

Next, let's add the logic that would handle the consumer side of things, which we add to a separate file.

CONSUMER.JS

```
import { AMQPClient } from '@cloudamqp/amqp-client'
import {} from 'dotenv/config'

async function startConsumer() {
  //Setup a connection to the RabbitMQ server
  const cloudAMQPURL = process.env.CLOUDAMQP_URL
  const connection = new AMQPClient(cloudAMQPURL)
  await connection.connect()
  const channel = await connection.channel()
  console.log("[✅] Connection over channel established")
  const q = await channel.queue('email.notifications')
  let counter = 0;

  const consumer = await q.subscribe({noAck: false}, async (msg) => {
    try {
      console.log(`[📩] Message received (${++counter})`, msg.bodyToString())
      msg.ack()
    } catch (error) {
      console.error(error)
    }
  })
  //When the process is terminated, close the connection
  process.on('SIGINT', () => {
    channel.close()
    connection.close()
    console.log("[❎] Connection closed")
    process.exit(0)
  });
}

startConsumer().catch(console.error);
```

Let's review the code snippet above.

`channel.queue('email.notifications')` asserts the queue of interest. `q.subscribe()` sets up a consumer that receives messages from the queue.

The `try {}` block is now receiving messages from the queue, in an attempt to process the information. This is where the logic would be placed for sending the actual email based on the content inside the message, which is not something we will go through in this example.

Finally, the `msg.ack()` function sends an "ack" to the queue for the received message. Which tells the Queue that the message has been handled successfully and can be removed.

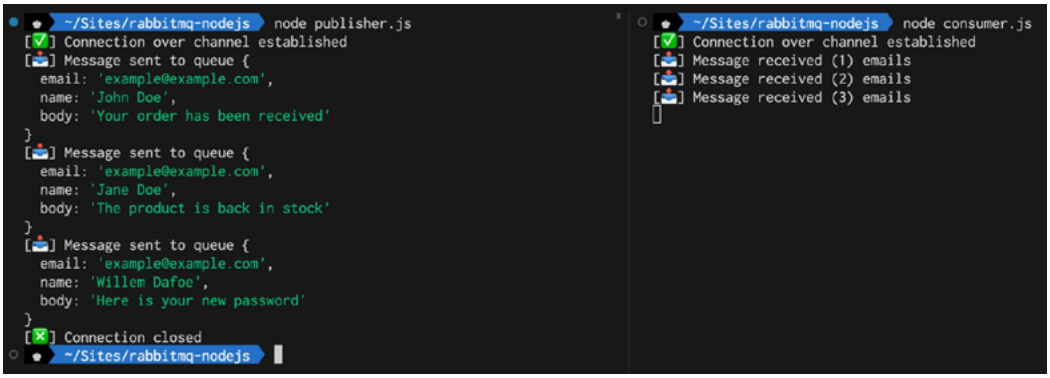
To run each script you can simply open up two terminals and navigate to the project folder. In the first terminal you can run:

```
node publisher.js
```

Then in the other terminal you can start the consumer:

```
node consumer.js
```

This is the expected output:



A FEW THINGS TO NOTE

The consumer.js script keeps running until you close it, unlike publisher.js. This means it can instantly consume any messages published to the queue. (To quit consumer.js you can press CTRL+c in your terminal)

In reality it's best to keep the connection alive in both the publisher and consumer scripts, rather than opening and closing it repeatedly. This is because opening and closing connections is considered "expensive".

You might notice that the consumer only received two out of the three messages we sent. This happened because the last message was sent with a routing key that didn't match any of the consumer's bindings (`resetpassword`), causing it to be lost. To avoid losing messages, make sure that all messages are routed to a queue.

Finally, if you want to send an email for resetting a password, the consumer would need to perform additional logic such as connecting to the database and generating a reset key. A good way to handle these messages separately would be to add another queue that bind to the corresponding routing key.

ADDITIONAL TRAINING SUGGESTIONS

1. Set up an additional queue that binds to the `resetpassword` routing key and then create another consumer that handles these messages. Rerun the producer to make sure all messages are delivered this time.

2. One of the many advantages of RabbitMQ is decoupling. As an example you may have producers and consumers developed by different teams and in different programming languages, all communicating successfully via the message broker. Try sending a message from your existing producer and write code in another programming language that successfully consumes the message.

PART ONE

RABBITMQ AND PYTHON WITH PIKA

Python developers have some options for AMQP client libraries. In this example, pika is used as an asynchronous client for publishing and consuming messages.

You need a RabbitMQ instance to get started. A free RabbitMQ instance can be set up for testing over at CloudAMQP, as described in Chapter 1. Once you have an instance, we add the environment variable CLOUDAMQP_URL to a .env file. You can find your CloudAMQP URL under Details in the CloudAMQP Console.

This code sample can be copied from https://training.cloudamqp.com

.ENV

```
#Update with your own credentials

CLOUDAMQP_URL="amqps://xxxyyyyzzzz:xxx@server.rmq.cloudamqp.com/xxxyyyyzzzz"
```

Next we need to install pika and dotenv. This can be done by running the following command in the terminal:

```
pip3 install pika load_dotenv
```

We will now create one script for the publisher that publishes messages and a consumer script that consumes messages from the queue.

The publisher first establishes a connection between the RabbitMQ instance and the application, through a channel within the connection. A Queue and an Exchange are declared and created (if they do not already exist), a binding is created between the two, and, finally, a few messages are published.

The consumer also sets up a connection+channel and subscribes to the queue. The messages are then handled one by one.

First, the full code for the publisher and the consumer is given. If you are following along, create a new py file and copy paste the code as we proceed.

PUBLISHER.PY

```
import pika, os

from dotenv import load_dotenv

load_dotenv()

# Access the CLOUDAMQP_URL environment variable and parse it (fallback to localhost)

url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f')

# Create a connection and open a channel

params = pika.URLParameters(url)

connection = pika.BlockingConnection(params)

channel = connection.channel()

print("[✅] Connection over channel established")

# Create an exchange called "emails"

channel.exchange_declare("emails", "direct")

# Declare a queue

channel.queue_declare(queue="email.notifications")

# Bind the queue to the exchange: queue_bind(name, exchange, binding_key)

channel.queue_bind("email.notifications", "emails", "notification")

def send_to_queue(channel, routing_key, email, name, body):

  msg = f"{email}, {name}, {body}"

  channel.basic_publish(

        exchange='emails',

        routing_key=routing_key,

        body=msg

  )

  print(f"[📩] Message sent to queue {msg}")


send_to_queue(

  channel, "notification", "example@example.com", "John Doe", "Your order has been
received"

)
```

```
send_to_queue(
  channel, "notification", "example@example.com", "Jane Doe", "The product is back in stock"
)
send_to_queue(
  channel, "resetpassword", "example@example.com", "Willem Dafoe", "Here is your new password"
)


try:
  connection.close()
  print("[❎] Connection closed")
except Exception as e:
  print(f"Error: #{e}")
```

Let's review the code snippet above. `pika.BlockingConnection(...)` returns a connection instance. The connection is established once the channel is created `connection.channel()`.

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

The `send_to_queue()` function will publish a message to the exchange named `emails` with a given routing key.

Next, let's add the logic that would handle the consumer side of things, which we add to a separate file.

CONSUMER.PY

```
import pika, os, sys
from dotenv import load_dotenv
load_dotenv()
# Access the CLOUDAMQP_URL environment variable and parse it (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f')
# Create a connection and open a channel
params = pika.URLParameters(url)
connection = pika.BlockingConnection(params)
channel = connection.channel()
print("[✅] Connection over channel established")
# Declare a Queue
channel.queue_declare(queue="email.notifications")
def callback(ch, method, properties, msg):
    print(f"[📩] Message received: {msg}")
    ch.basic_ack(delivery_tag=method.delivery_tag)
channel.basic_consume(
    "email.notifications",
    callback,
    auto_ack=False,
)
try:
  print("\n[❎] Waiting for messages. To exit press CTRL+C \n")
  channel.start_consuming()
except Exception as e:
  print(f"Error: #{e}")
  try:
    sys.exit(0)
  except SystemExit:
    os._exit(0)
```

Let's review the code snippet above.

`channel.queue_declare(queue="email.notifications")` asserts the queue of interest. `channel.basic_consume()` sets up a consumer that receives messages from the queue.

The `callback` block is now receiving messages from the queue, in an attempt to process the information. This is where the logic would be placed for sending the actual email based on the content inside the message, which is not something we will go through in this example.

Finally, the `ch.basic_ack` function sends an "ack" to the queue for the received message. Which tells the Queue that the message has been handled successfully and can be removed.

To run each script you can simply open up two terminals and navigate to the project folder. In the first terminal you can run:

```
python3 publisher.py
```

Then in the other terminal you can start the consumer:

```
python3 consumer.py
```

This is the expected output:

A FEW THINGS TO NOTE

The consumer.py script keeps running until you close it, unlike publisher.py. This means it can instantly consume any messages published to the queue. (To quit consumer.js you can press CTRL+c in your terminal)

In reality it's best to keep the connection alive in both the publisher and consumer scripts, rather than opening and closing it repeatedly. This is because opening and closing connections is considered "expensive".

You might notice that the consumer only received two out of the three messages we sent. This happened because the last message was sent with a routing key that didn't match any of the consumer's bindings (`resetpassword`), causing it to be lost. To avoid losing messages, make sure that all messages are routed to a queue.

Finally, if you want to send an email for resetting a password, the consumer would need to perform additional logic such as connecting to the database and generating a reset key. A good way to handle these messages separately would be to add another queue that bind to the corresponding routing key.

ADDITIONAL TRAINING SUGGESTIONS

1. Set up an additional queue that binds to the `resetpassword` routing key and then create another consumer that handles these messages. Rerun the producer to make sure all messages are delivered this time.

2. One of the many advantages of RabbitMQ is decoupling. As an example you may have producers and consumers developed by different teams and in different programming languages, all communicating successfully via the message broker. Try sending a message from your existing producer and write code in another programming language that successfully consumes the message.

PART ONE

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.

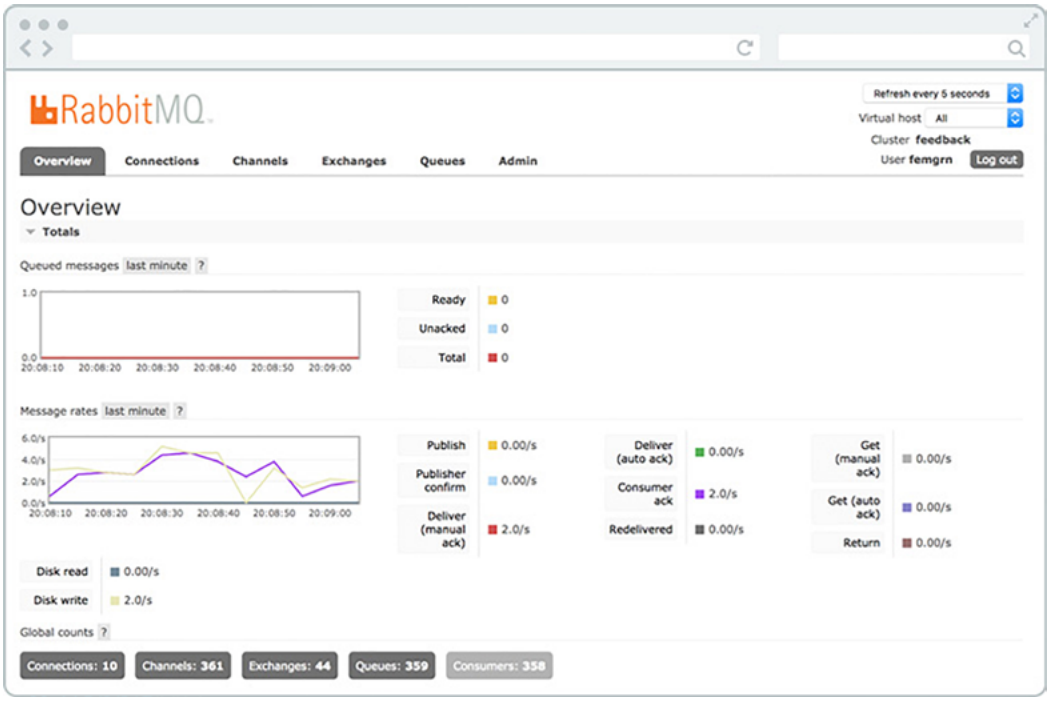


Figure 15 *- The RabbitMQ management interface.*

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.

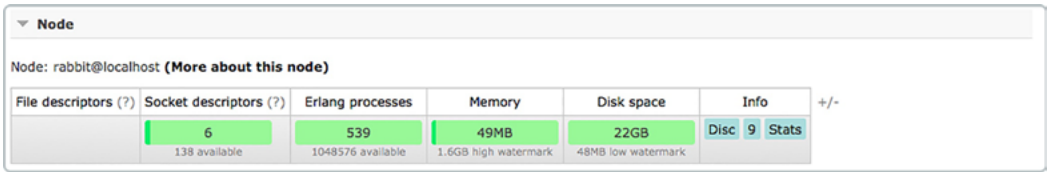


Figure 16 *- Node-specific information.*

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.

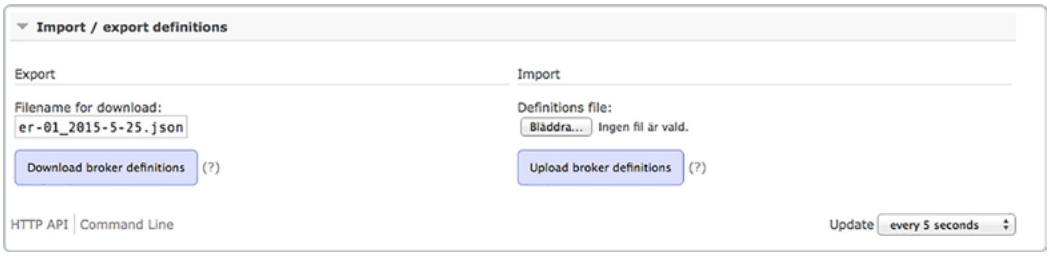


Figure 17 *- Import/export definitions as a JSON file.*

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.

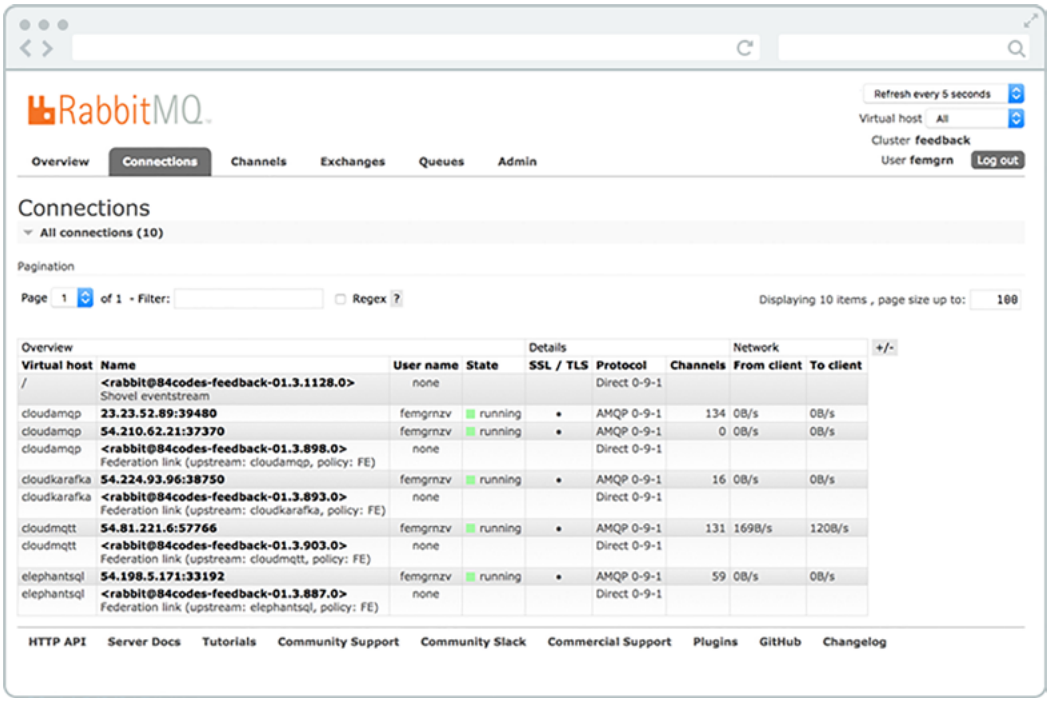


Figure 18 *– The Connections tab in the RabbitMQ management interface.*

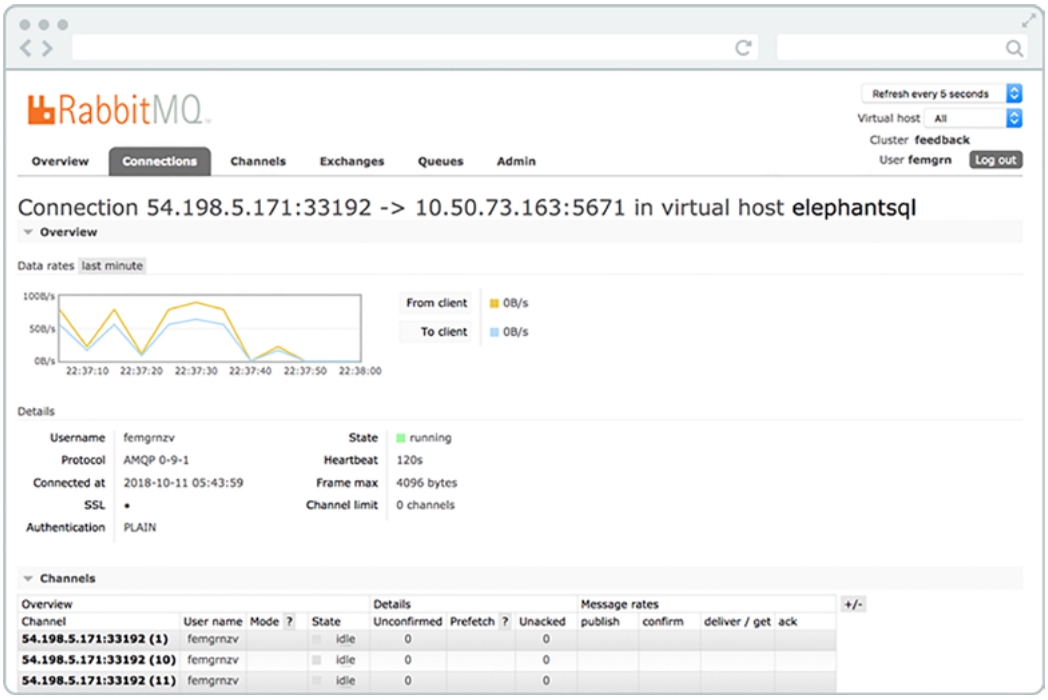


Figure 19 *- Connection information for a specific connection.*

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 a 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.

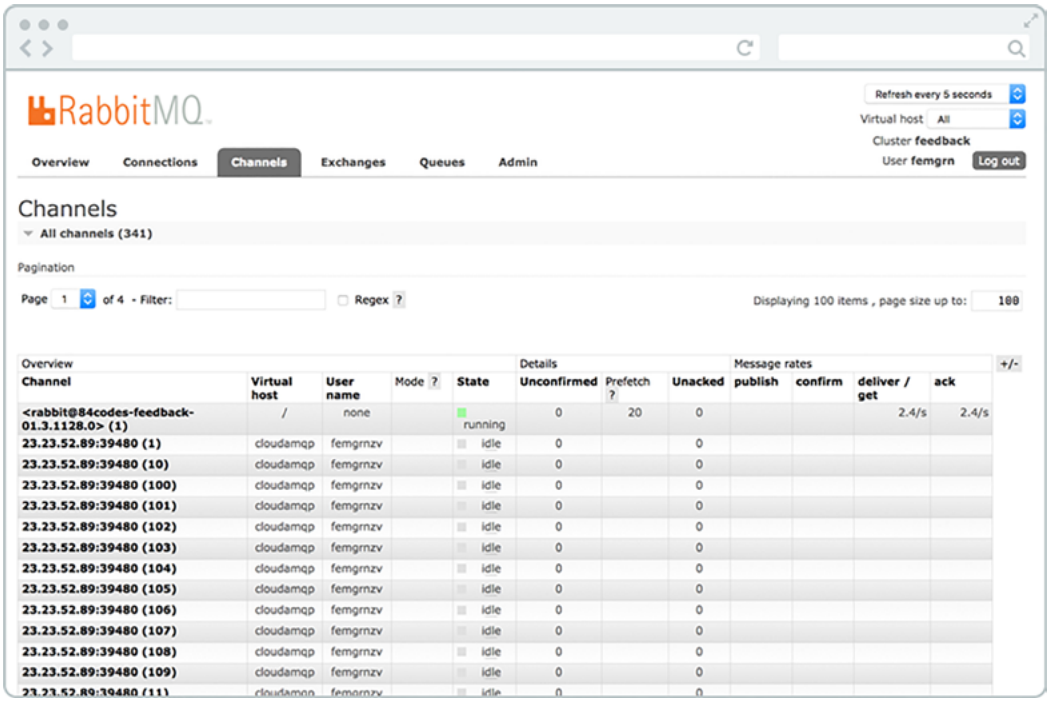


Figure 20 *– The Channels tab.*

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.

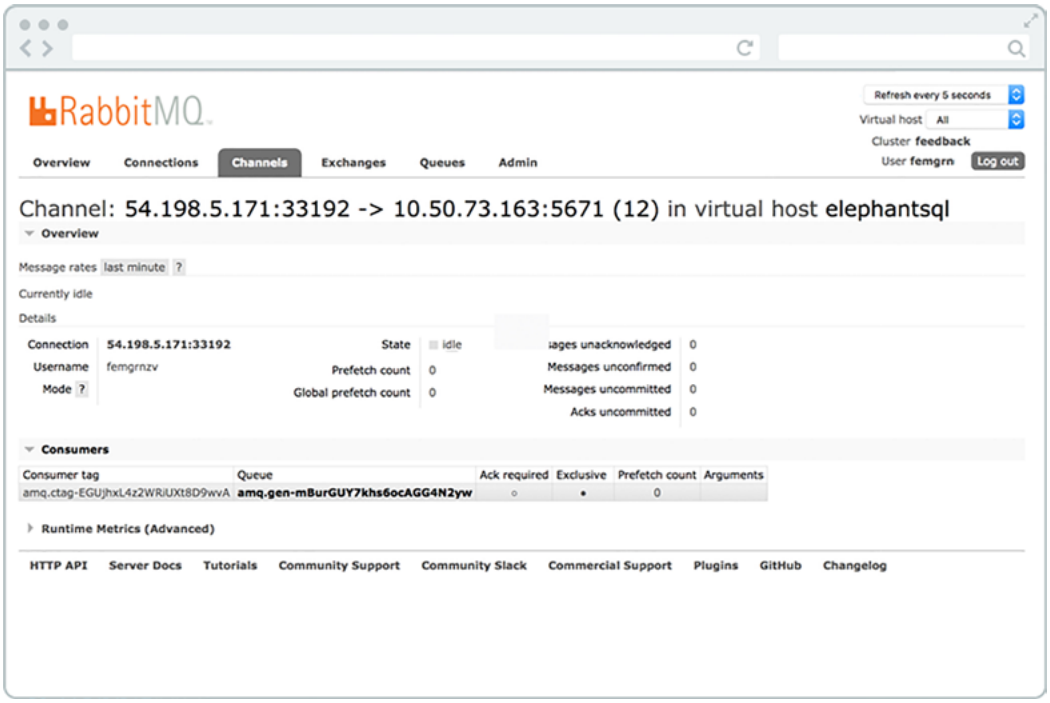


Figure 21 *- Detailed information about a specific channel.*

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.

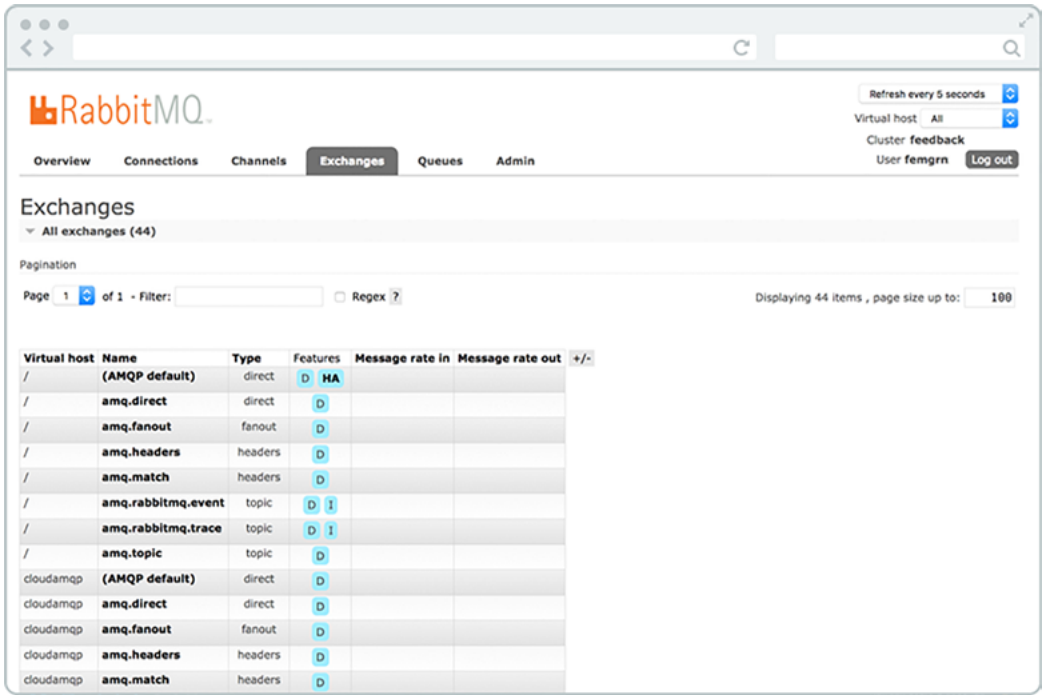


Figure 22 *- The exchanges tab in the RabbitMQ management interface.*

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.

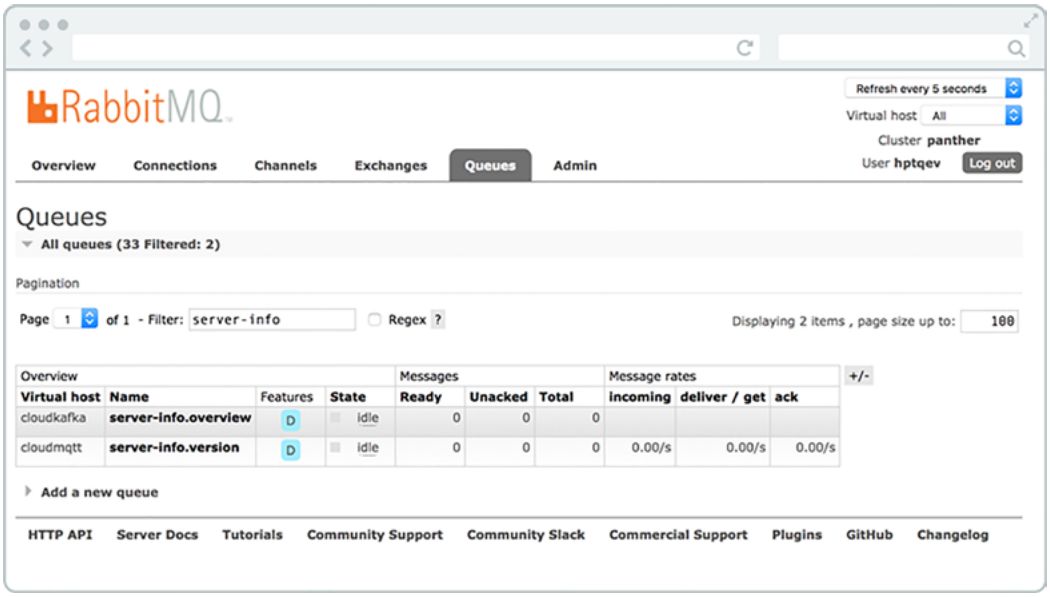


***Figure 24** - The queues tab in the RabbitMQ management interface.*

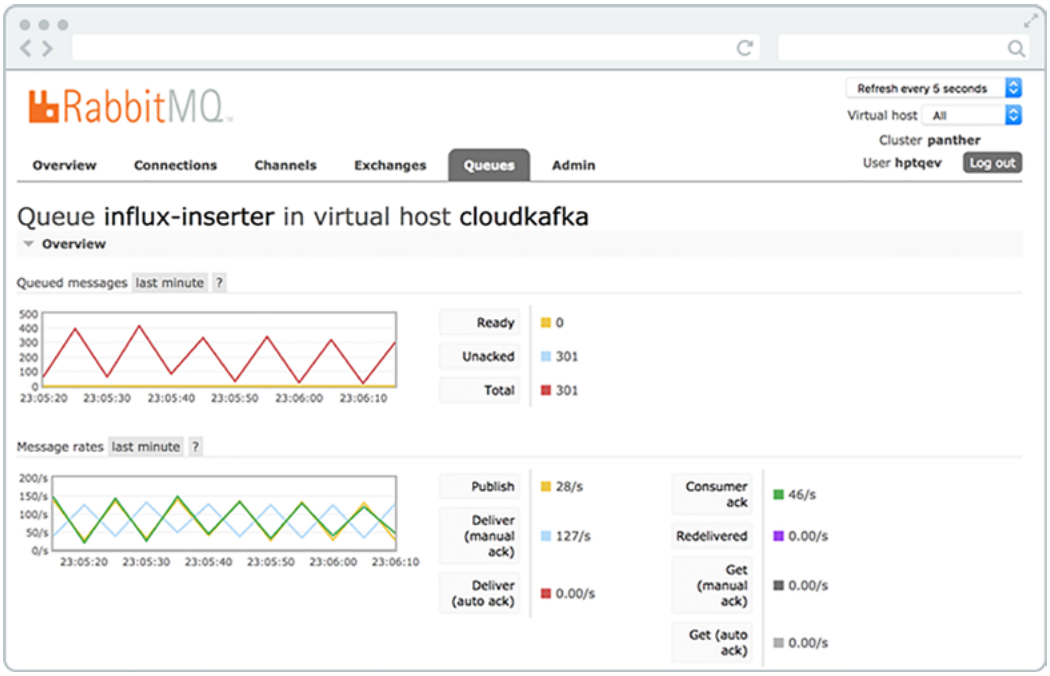


***Figure 25** - Specific information about a single queue.*

CONSUMERS

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

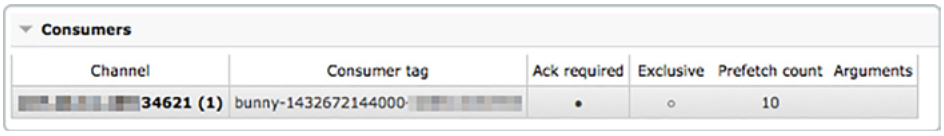


***Figure 26** - Consumers connected to a queue.*

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).

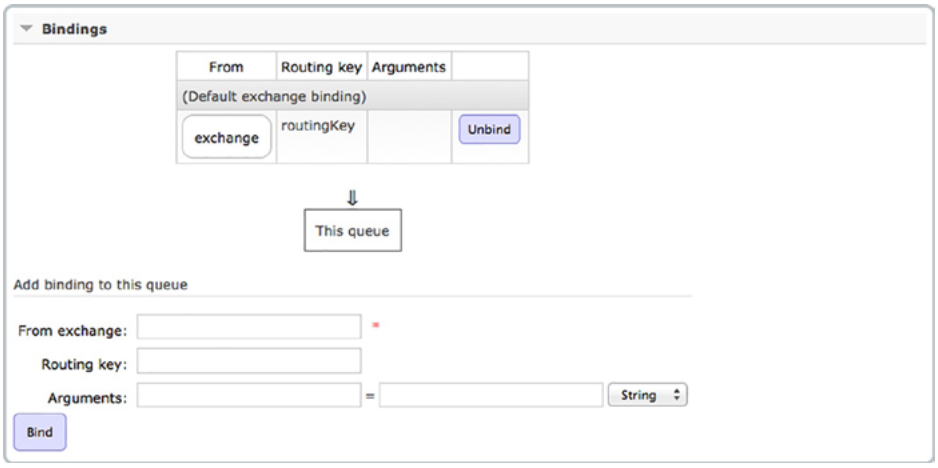


Figure 27 *- The bindings interface.*

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.

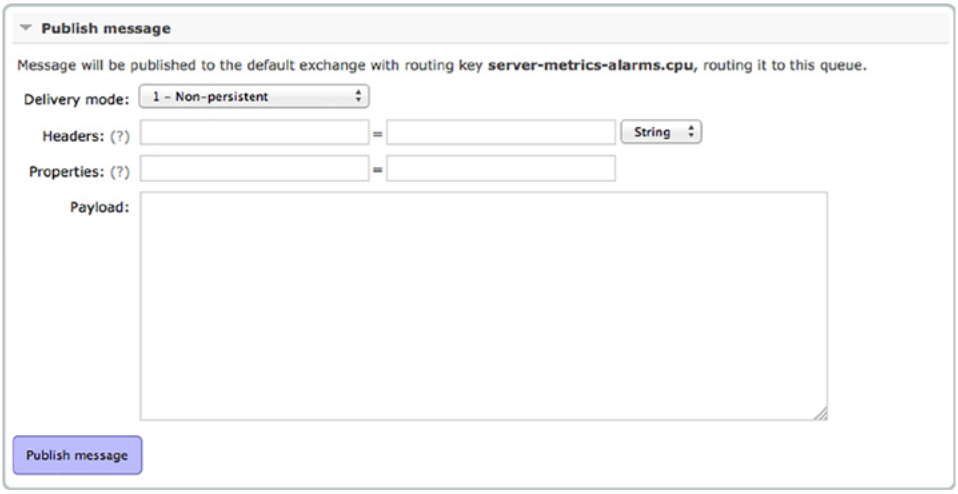


Figure 28 *- Manually publishing a message to the queue.*

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.

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.

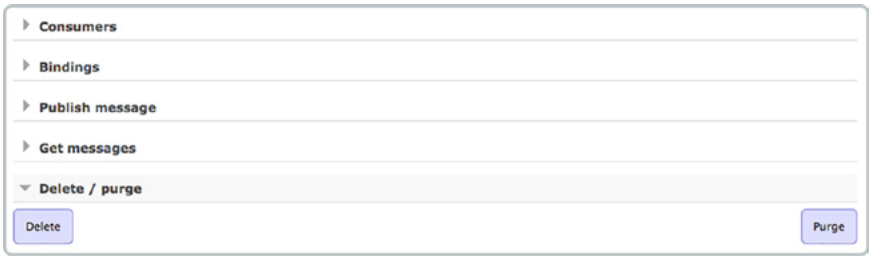


Figure 30 - Delete or purge a queue from the web interface.

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.

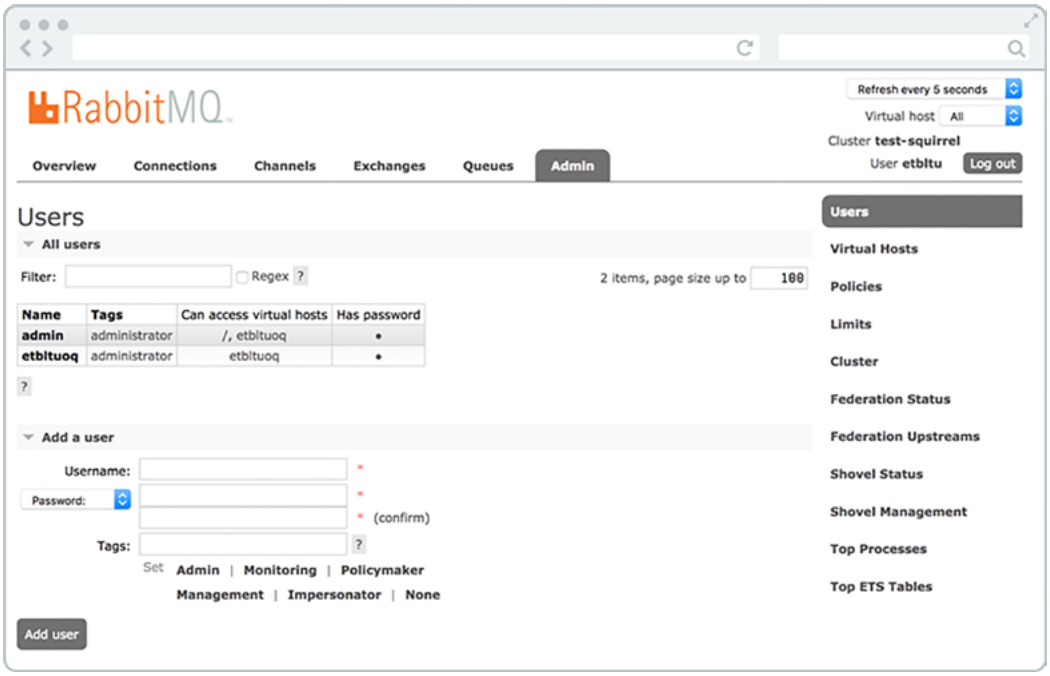


***Figure 31** - The Admin interface where users can be added.*

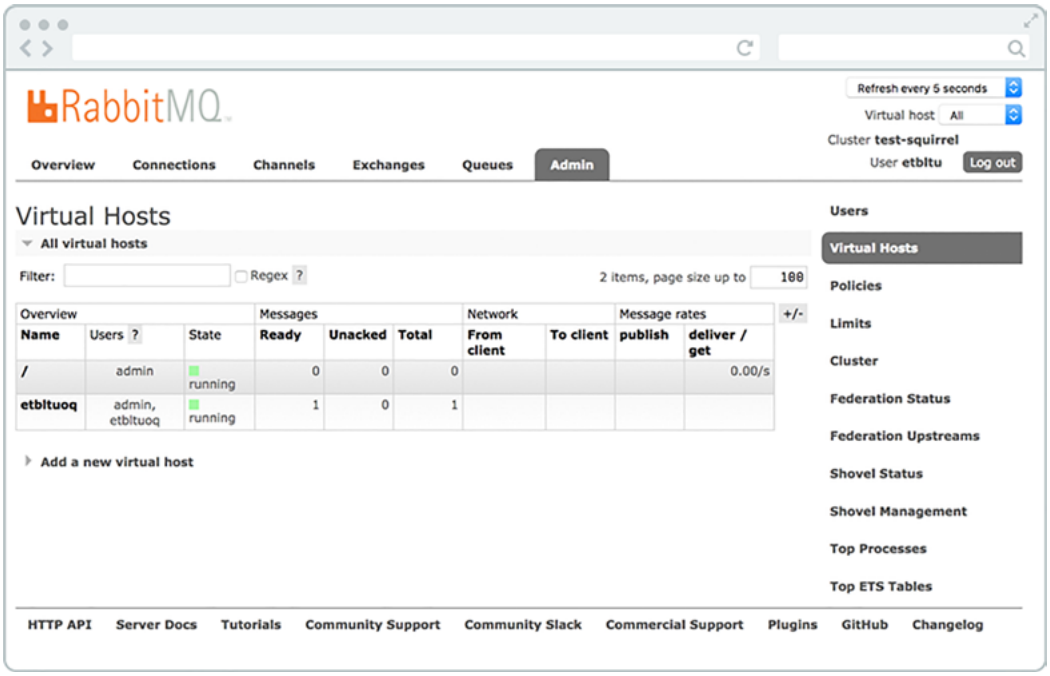


***Figure** 32 - Virtual Hosts can be added from the Admin tab.*

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).

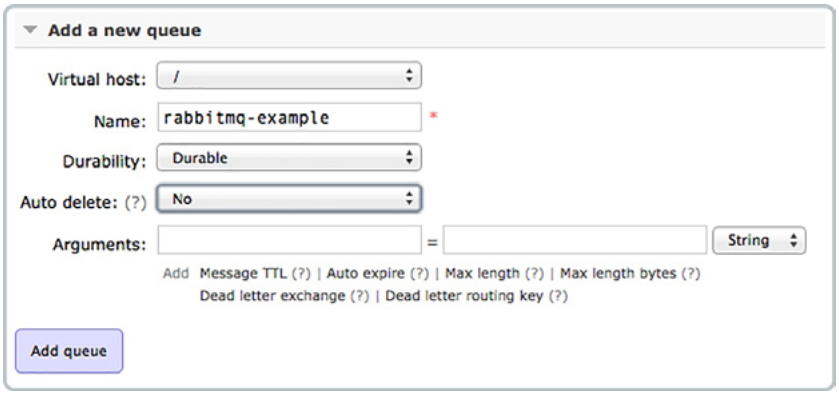


***Figure** 33 - Queue view, add queue.*

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.

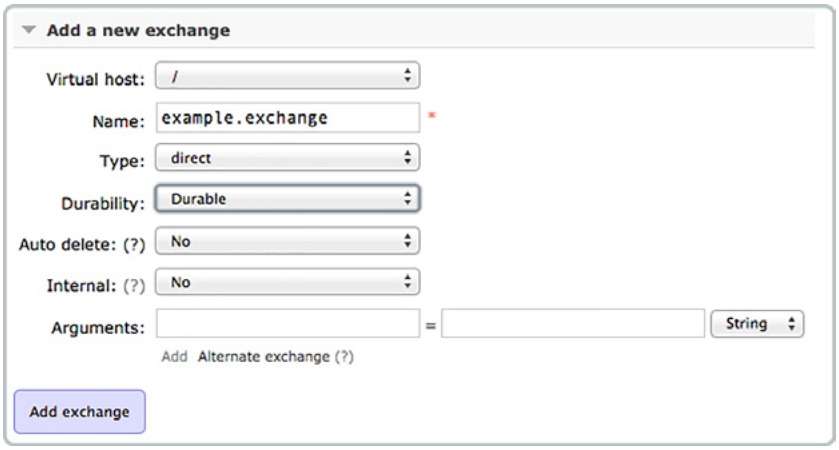


Figure 34 *- Exchange view, add exchange.*

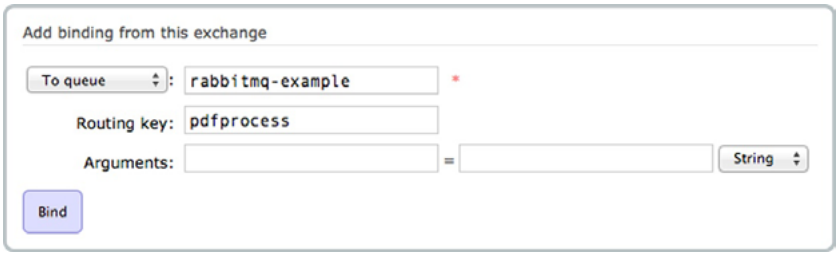


Figure 35 *- Click on the exchange or on the queue, go to "Add binding from this exchange" or "Add binding to this queue".*

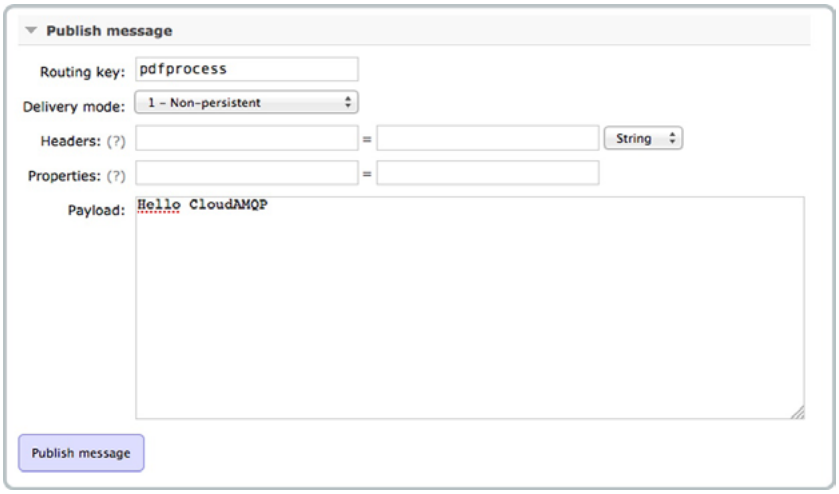


***Figure 36** - Publish a message to the exchange with the routing key "pdfprocess".*

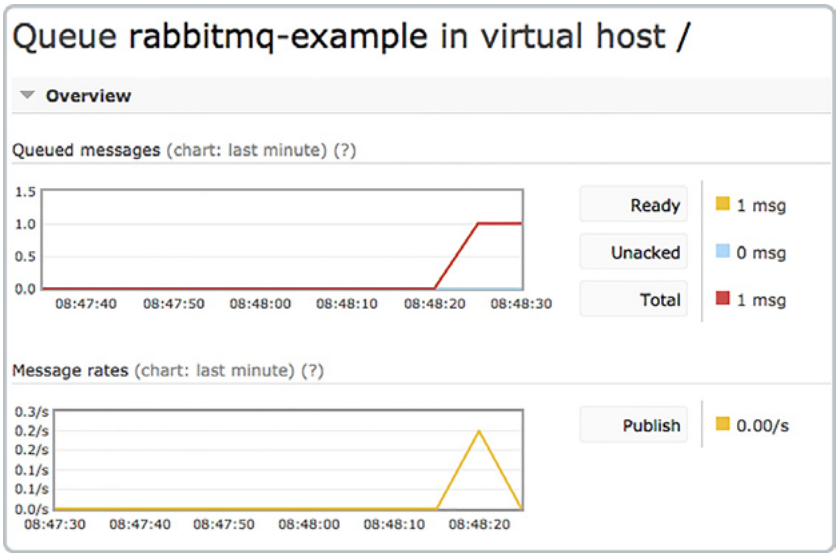


***Figure 37** - Queue overview for example-queue when a message is published.*

PART ONE

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*.

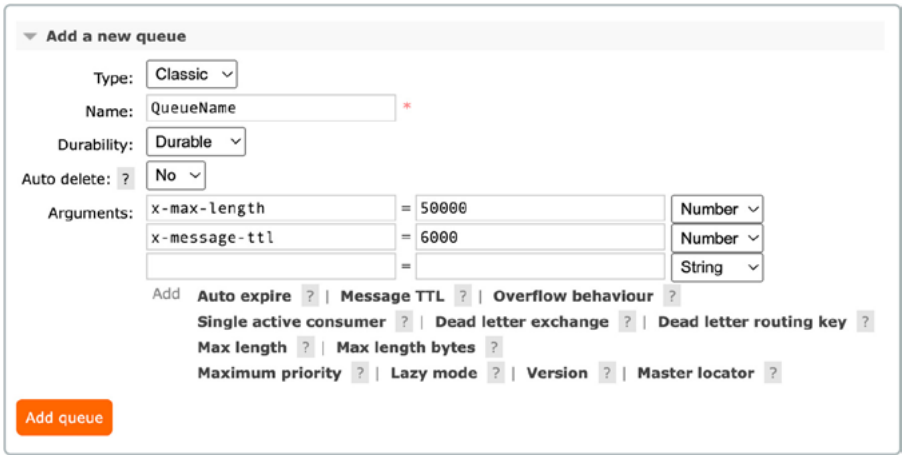


***Figure 38** - Queue created with the durability property, and two arguments, x-max-length and x-message-ttl.*

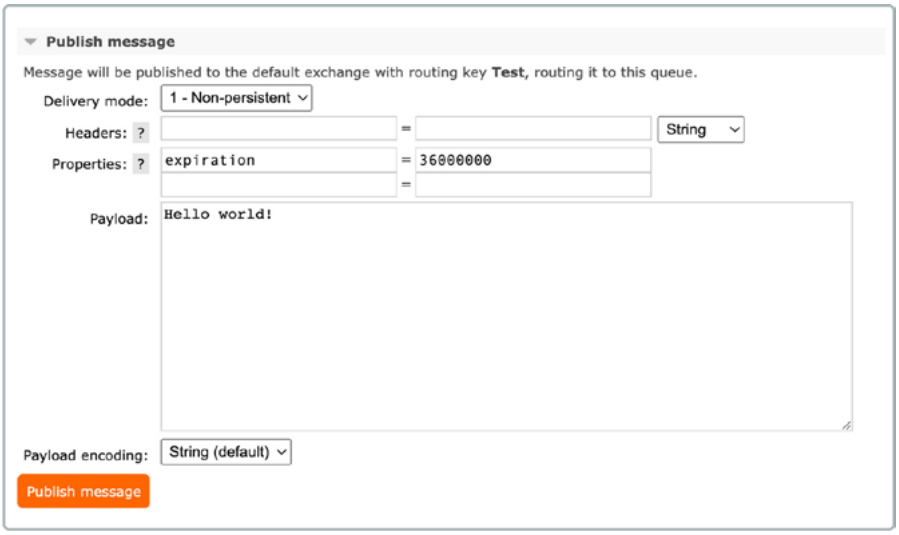


***Figure 39** - A message published with TTL expiration set to 36000000 milliseconds.*

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.

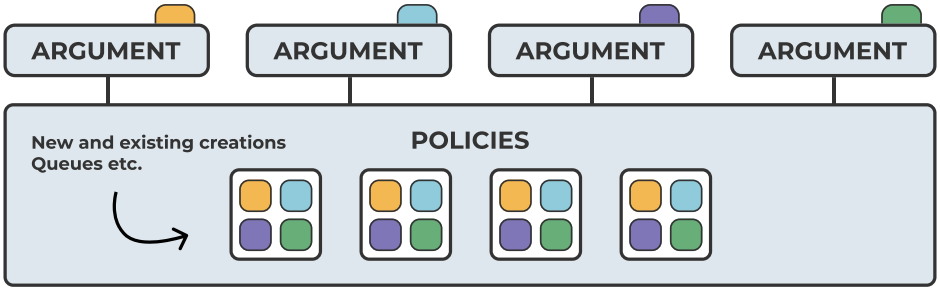


***Figure 40** - RabbitMQ Queue arguments applied through policies.*

PART ONE

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]
```

PART TWO

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.

PART TWO

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.

PART TWO

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.

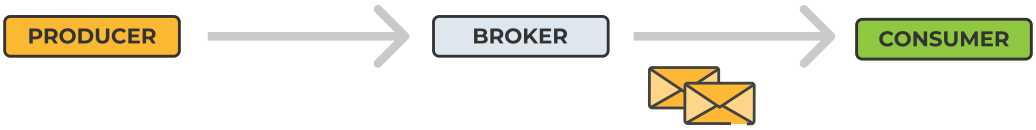


***Figure 41** - Message flow in RabbitMQ.*

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 COUND 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.

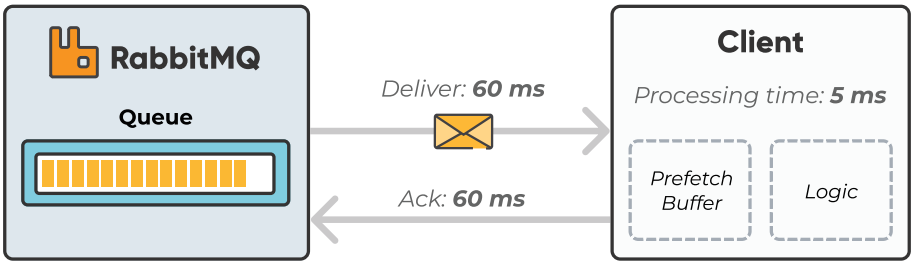


***Figure 42** - RabbitMQ Prefetch round trip*

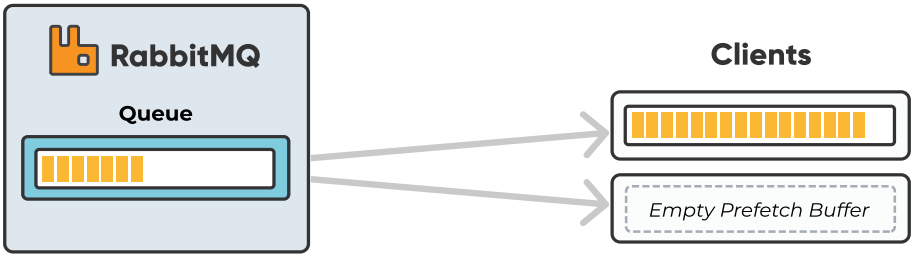


***Figure 43** - Idle consumer due to large prefetch count.*

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.

PART TWO

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.

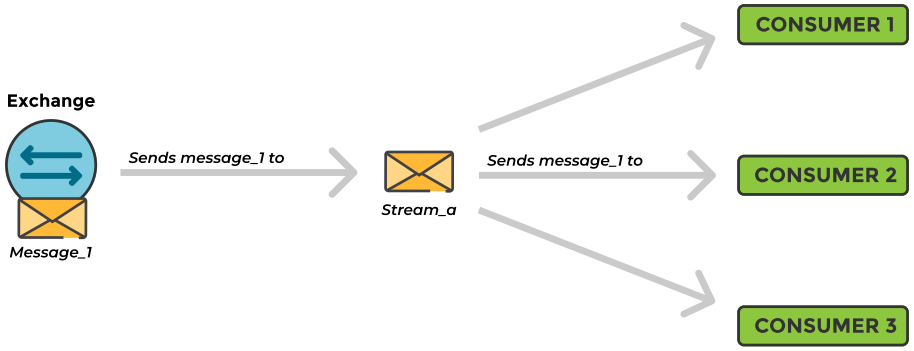
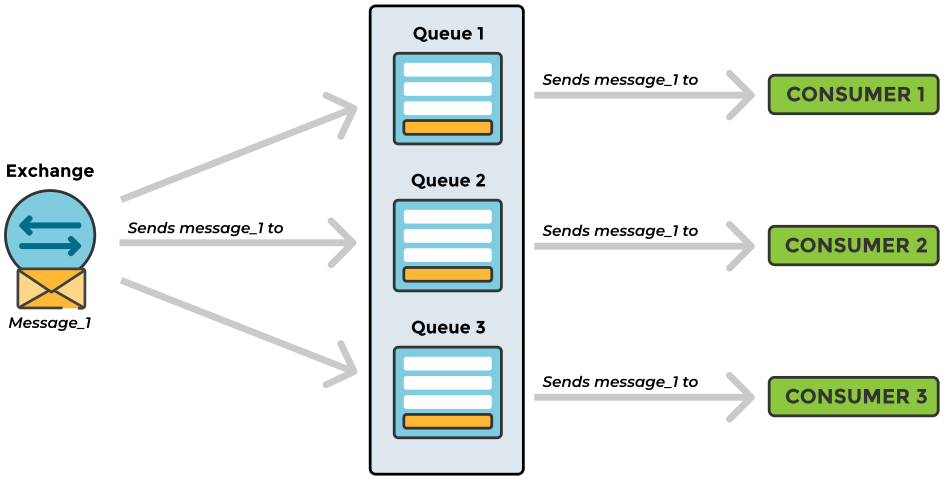


***Figure 44** - Example of fanout in RabbitMQ Streams.*

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.

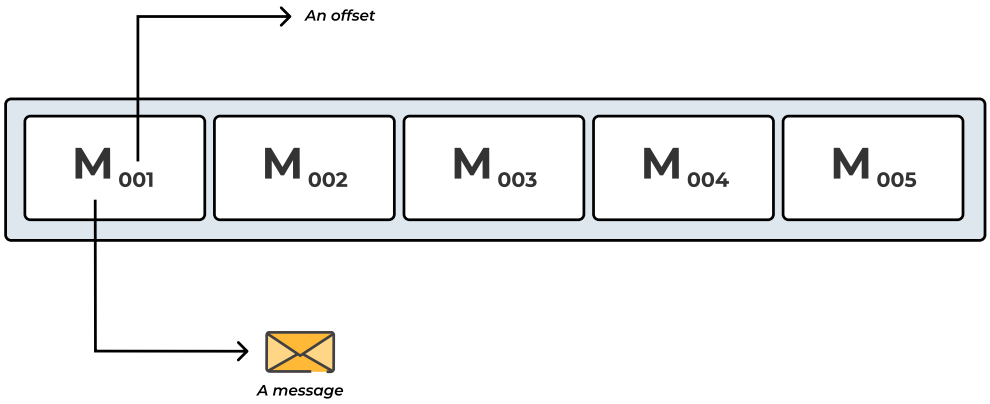


Figure 46 - Queue offset in RabbitMQ Streams.

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.

PART TWO

RABBITMQ STREAMS IMPLEMENTATION

When it comes to client applications communicating with a Stream, there are a couple of options available. One option is to use an AMQP client library, similar to how they communicate with queues. However, it is highly recommended to utilize the dedicated Streams protocol plugin along with its associated client libraries.

In the following paragraphs, we will delve into both options and help you make an informed decision.

Note: RabbitMQ client libraries abstract the low-level details of connecting to a queue or stream in RabbitMQ. Think of them as the packages that simplify image processing, or making http requests in your favorite programming languages.

USING RABBITMQ STREAMS WITH AN AMQP CLIENT LIBRARY

Like queues, there are three steps to working with RabbitMQ Streams via an AMQP client library:

- Declare/Instantiate a Stream
- Publish messages to the Stream
- Consume messages from the Stream

Declaring a RabbitMQ Stream

Streams are declared with the AMQP client libraries the same way queues are created. Set the `x-queue-type` queue argument to stream, and provide this argument at declaration time.

```
import pika, os


# Access the CLODUAMQP_URL environment variable and parse it (fallback to localhost)
rabbitmq_url = os.environ.get(

'CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f'

)


params = pika.URLParameters(rabbitmq_url)

connection = pika.BlockingConnection(params)

channel = connection.channel() # start achannel


# Declare a Stream, named test_stream

channel.queue_declare(

queue='test_stream', durable=True,

arguments={"x-queue-type": "stream"} )
```

Alternatively, a Stream can be created using the RabbitMQ Management UI. In that case, the Stream type must be specified using the queue type drop-down menu.

Publishing a Message to a RabbitMQ Stream

Publishing messages to a Stream is no different from publishing messages to a queue. As an example, below, the previous snippet has been extended to publish a message to the `test_stream` declared.

```
import pika, os


# Access the CLODUAMQP_URL environment variable and parse it (fallback to localhost)
rabbitmq_url = os.environ.get(
'CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f' )
params = pika.URLParameters(rabbitmq_url)


connection = pika.BlockingConnection(params)
channel = connection.channel() # start a channel


# Declare a Stream, named test_stream
channel.queue_declare(
        queue='test_stream',
        durable=True,
        arguments={"x-queue-type": "stream"}
)


# Publish a message to the test_stream
channel.basic_publish(
exchange='', routing_key='test_stream', body='Welcome email message'
)
```

To summarize, the script above declared a RabbitMQ Stream, `test_stream` then published a message to it with the `basic_publish` function. Next, an explanation of how to consume this message.

Assume the message published is a welcome email that needs to be forwarded to a user after signup.

Consuming a Message from a RabbitMQ Stream

Messages can be consumed from a Stream the same way queues accomplish this task, more or less, but with two major differences.

1. Consuming messages in RabbitMQ Streams requires setting the **QoS prefetch**.

2. You can specify an **offset** to start reading/consuming from any point in the log stream. If unspecified, the consumer starts reading from the most recent offset written to the log stream after it starts.

QoS prefetch

When RabbitMQ delivers a message from a Stream to a consumer, the consumer returns an acknowledgment confirming that it has received the message. Generally, this is a data safety measure designed to ensure that no message is lost while in flight. For example, a message could fail to reach its destination (a consumer) due to a network issue. But how is this related to QoS prefetch?

Oversimplified, a consumer processes one message at a time by lining up incoming messages while processing the current one (for ease of access). Let's call these lined-up messages "in progress" messages. The QoS prefetch setting specifies the maximum number of "in-progress" messages that a specific consumer could accommodate at a time.

Let's take the case of a consumer whose **QoS prefetch** is set to 3. Imagine this consumer currently having 3 "in-progress" messages(the currently executing message not included): `message-1` , `message-2` , and `message-3` .

Because this consumer's QoS prefetch is set to 3 and it already has 3 "in-progress" messages, RabbitMQ will not push a fourth message, `message-4`, until one of the "in-progress" messages has been resolved. RabbitMQ implements this check to ensure that consumers are not overwhelmed with messages.

In the below example, a consumer subscribing to `test_stream` is declared, then its QoS prefetch is set. Note that even though `test_stream` is declared from the publishing side, it's good practice to declare it from the consuming side as well.

```
import pika, os, time


def send_welcome_email(msg):
  print("Welcome Email task processing")
  print(" [x] Received " + str(msg))
  time.sleep(5) # simulate sending email to a user --delays for 5 seconds
  print("Email successfully sent!")
  return
```

```
# Access the CLODUAMQP_URL environment variable and parse it (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f')
params = pika.URLParameters(url)

connection = pika.BlockingConnection(params)
channel = connection.channel() # start a channel

# Declare our stream
channel.queue_declare(
  queue='test_stream',
  durable=True,
  arguments={"x-queue-type": "stream"}
)

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

# Set the consumer QoS prefetch
channel.basic_qos(
  prefetch_count=100
)

# Consume messages published to the stream
channel.basic_consume(
  'test_stream',
  callback,
)

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

The script above declared test_stream again, setting the QoS prefetch to 100 using the basic_qos function. The consumer triggers the callback function when it processes a new message. The callback function, in turn, invokes the send_welcome_email function that simulates sending an email to a user.

Notice how an offset isn't specified in our basic_consume

```
# Consume messages published to the stream

channel.basic_consume( 'test_stream', callback)
```

As a result, the consumer would start reading from the most recent offset written to test_ stream after the consumer starts. "After" has been deliberately emphasized here to allow for the cross-examination of an interesting behavior of Streams.

RabbitMQ Streams can receive and buffer messages even before any consumer is bound. When a consumer is eventually bound to such a Stream, it is expected that the Stream will automatically deliver all existing messages to this new consumer. At least queues behave that way.

However, RabbitMQ Streams behave differently. The only messages that would be automatically delivered to a consumer from the Stream it's bound to are the messages published to the Stream after the consumer starts.

For example, Imagine a Stream, stream_a, that already has one message, message_1. Assume stream_a currently has no consumers bound to it. Five minutes later, however, a new consumer, consumer_a connects to stream_a. Because consumer_a connected after message_1 had already been delivered to stream_a, RabbitMQ won't automatically deliver message_1 to this new consumer.

But this begs the question: how can consumer_a grab message_1 , an old message?

By using the message's offset. For example, if the ID of message_1 or the published timestamp is known, consumer_a can start reading from the message by passing the x-stream-offset argument to the basic_consume function.

```
channel.basic_consume(

  'test_stream',

  callback,

  arguments={"x-stream-offset": 5000}

)
```

In the snippet above, it is assumed that the message to start reading from has the ID of 5000.

By passing “first” or “last” to the `x-stream-offset` argument, the consumer would start reading from the first message in the log stream or from the last written chunk of messages, respectively. See the snippet below.

```
# Grabbing the first message
channel.basic_consume(
  'test_stream',
  callback,
  arguments={"x-stream-offset": "first"}
)

# Grabbing the last message
channel.basic_consume(
  'test_stream',
  callback,
  arguments={"x-stream-offset": "last"}
)
```

USING RABBITMQ STREAMS WITH THE BINARY STREAM PROTOCOL

There are four steps to working with Streams using the binary stream protocol:

- Enable the Stream plugin on the RabbitMQ instance
- Create/Declare a Stream
- Publish a message to the Stream
- Consume a message from the Stream

Enable the Stream Plugin

In the previous steps we connected to a RabbitMQ instance on CloudAMQP. Here, we’d be spinning up a local RabbitMQ instance with docker. This is because enabling a plugin on CloudAMQP isn’t supported for users on the free plan at this time.

Start a RabbitMQ docker container with the command below:

```
docker run -it --rm --name rabbitmq -p 5552:5552 \\
-e RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS='-rabbitmq_stream advertised_host localhost' \\
rabbitmq:3.9
```

Next, enable the Stream Plugin on the RabbitMQ instance

```
docker exec rabbitmq rabbitmq-plugins enable rabbitmq_stream
```

Create/Declare a RabbitMQ Stream

The rstream Python client will be used to interact with Streams. Alternatively, check out the stream java client.

Note: the stream Python client is still a work in progress and not yet officially approved. The Java stream client, on the other hand, has been approved. The Python client is used here to visualize what working with RabbitMQ Streams might look like in Python.

To create a Stream, first create a Producer(that will serve as the interface for creating streams and publishing messages to the stream).

```
from rstream import Producer


# Create a producer
producer = Producer(
  host='localhost',
  port=5552,
  username='guest',
  password='guest'
)


# Create a Stream named 'mystream'
producer.create_stream('mystream')
```

Publish a Message to the RabbitMQ Stream

Next, extend the previous snippet to publish a message to `mystream`.

```
from rstream import Producer, AMQPMessage

# Create a producer
producer = Producer(
  host='localhost',
  port=5552,
  username='guest',
  password='guest'
)

# Create a Stream named 'mystream'
producer.create_stream('mystream')

# Construct the message
message = AMQPMessage(
  body='hello world'
)

# Publish the message
producer.publish('mystream', amqp_message)
```

Consume a Message from the Stream

Next, consume the message published to `mystream`.

```
from rstream import Consumer, amqp_decoder, AMQPMessage


# Create a consumer
consumer = Consumer(
      host='localhost',
      port=5552,
      username='guest',
      password='guest',
)


# More like a callback
def on_message(msg: AMQPMessage):
  print('Got message: {}'.format(msg.body))


consumer.start()
consumer.subscribe('mystream', on_message, decoder=amqp_decoder)
consumer.run()
```

SUMMARY

This chapter illustrated two approaches to setting up communication between consumers and producers with a RabbitMQ Stream:

- With AMQP client libraries
- Or with the binary stream protocol

Even though working with the AMQP client libraries is easier, the RabbitMQ documentation recommends connecting consumers and producers to Streams with the binary stream protocol. This usually yields better performance.

To understand RabbitMQ streams a step further, let's dive into the limitations of Streams and custom configurations.

PART TWO

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.

PART TWO

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.

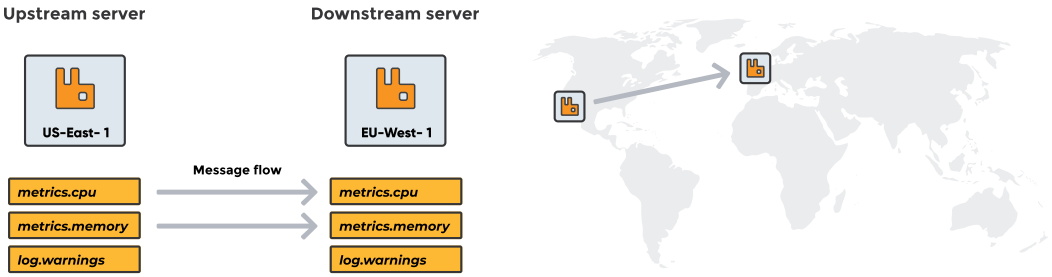


***Figure 47** - Upstream and downstream servers.*

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).

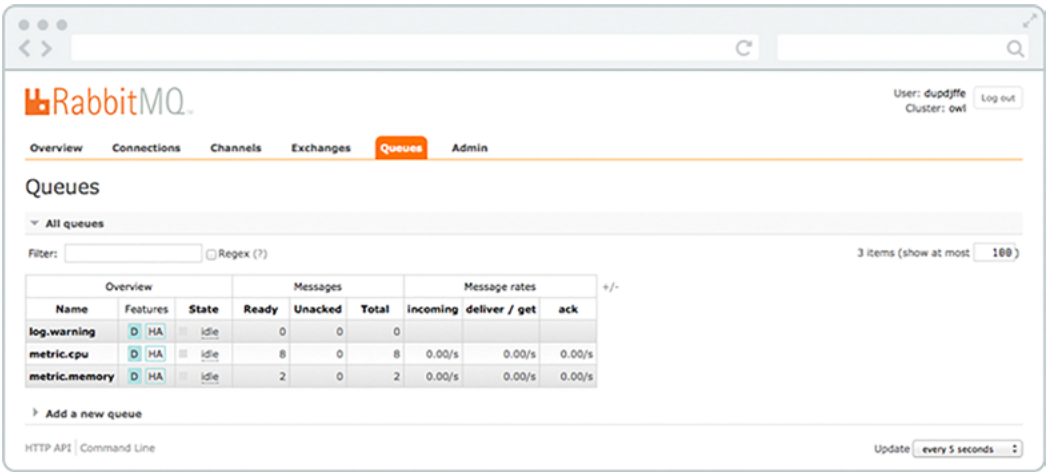


***Figure 48** - Migration of metric.cpu and metric.memory.*

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

2. **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.

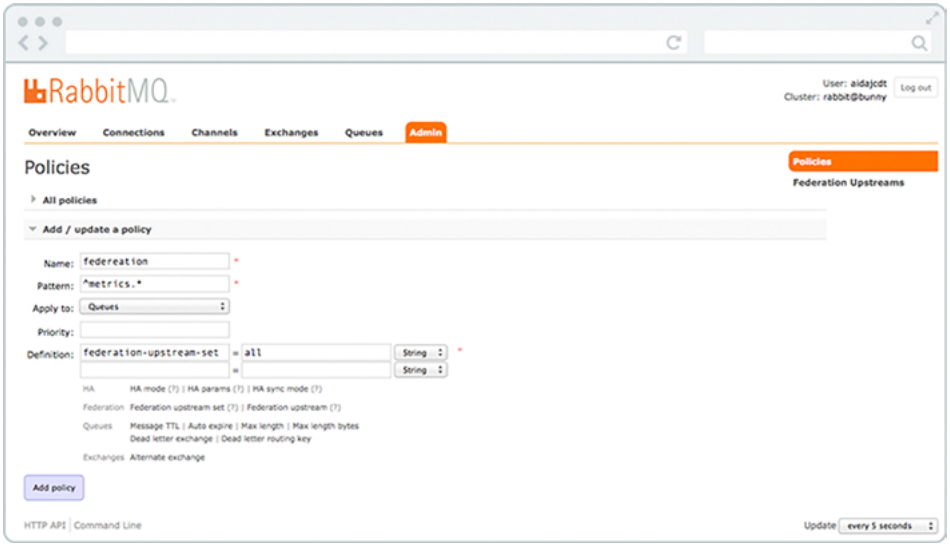


Figure 49 *- Set up the federation to the upstream.*

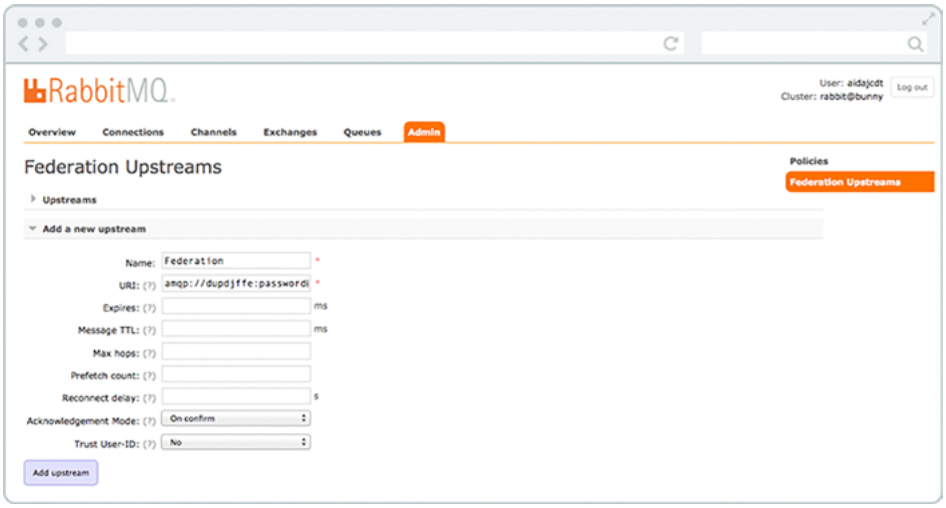


Figure 50 *- Set up the federation to the upstream.*

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.

PART TWO

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 ack(knowledge) 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, exhcanges 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.

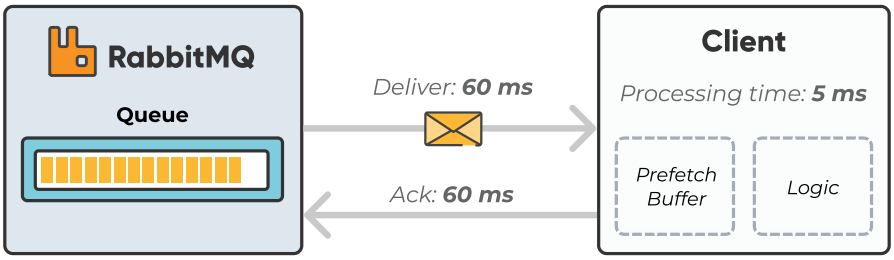


Figure 51 *- Too small prefetch count may hurt performance.*

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.

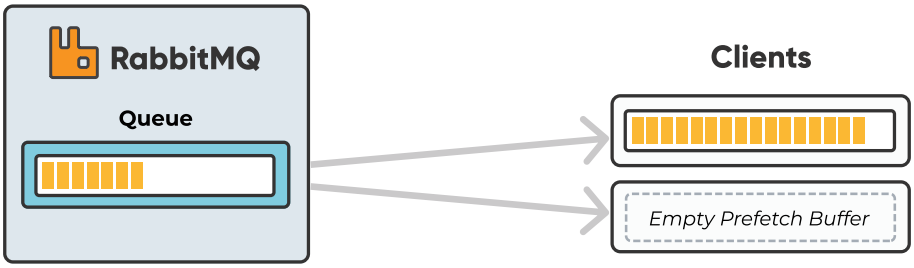


***Figure 52** - A large prefetch count could deliver lots of messages to one single consumer.*

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.

PART TWO

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 copy 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.

PART TWO

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.

PART TWO

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 onsumers 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","pay-
load_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.rmq.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);
}
```

PART THREE

RABBITMQ USER STORIES

USER STORY

User stories from customers describing their experience using RabbitMQ in production.

A story about how RabbitMQ is used in the real world is always of great value. The following are some interesting user stories from various sources that describe how RabbitMQ is used in production.

PART THREE

ADIDAS: TRACKING SPORT ACTIVITIES

You go for a run, and when you're done, your buddy, adidas Running, will tell you how you did. Maybe you hit a new goal this run or need encouragement to work harder next time. No matter what, behind the scenes there is a dedicated team of engineers relying on the stable foundation of RabbitMQ.

Ever since sports tracking apps saw the light of day, it's almost like a workout without tracking, somehow is less valuable to us. Imagine a run streak on day 1,000 without functioning measurements! What a disappointment!

Today, we're so used to the data and statistics generated from our physical activities. The numbers indicate growth and progress and motivate people to do better, but they can also tell if you are struggling with your health, which might be a wake-up call for some. That's how important these statistics have become to us. That also underlines the importance of a well-functioning application infrastructure, especially for two of the world's most popular training apps: adidas Running and adidas Training.

ANOTHER STRING ON ADIDAS' BOW

adidas hardly needs any further presentation. The classic brand known for the three stripes is seen worldwide on everything from shoes to clothing and accessories. In 2015, adidas built on its brand by acquiring Runtastic, making training apps available to its loyal customers.

Alexander Lackner is an infrastructure engineer with Runtastic, the company responsible for developing the adidas Running and Training apps, keeping them running smoothly, and providing them with new content. He tells a story of a relatively young application that has grown explosively in popularity, not least during the pandemic.

– "Most gyms were closed, which also applied to group training and places in general where people were used to doing sports. Physical activity in the outdoors grew massively overnight, and we saw a huge spike in the number of users. We were overwhelmed by the wave of tasks we had to handle simultaneously."

This could have been a bigger problem than it was for Runtastic.

– "We have been running RabbitMQ in basically all of our services from the beginning. That helped a lot when we needed it the most. RabbitMQ made it easy for us to scale up and handle the heap of workload coming in all at once."

MICROSERVICES THAT COMMUNICATE THROUGH RABBITMQ

adidas training apps are built on a microservice infrastructure, meaning different application functions are divided into smaller pieces called microservices. They communicate via a message queue, in this case, RabbitMQ. For example, messages get generated in the adidas training app every time a user starts an activity. The different services then exchange these messages and “listen“ to each other, depending on whether the service “needs” the information. For instance, if a sports activity ends, the leaderboard service listens, automatically updating it with the correct information.

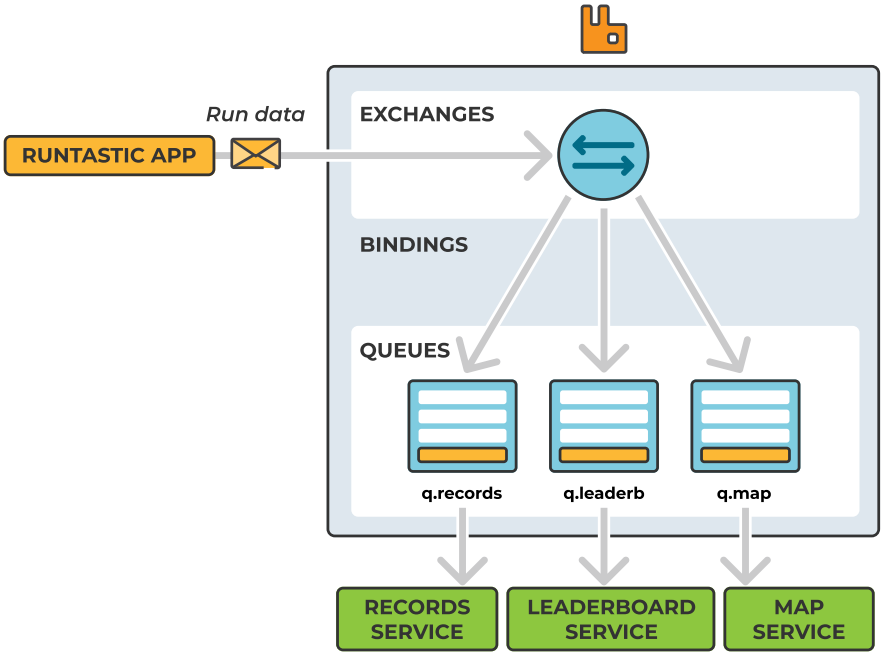


Figure 53 - RabbitMQ implementation for runtastic

Another example of when RabbitMQ comes in handy is when sending push notifications.

– You can imagine it gets sent out to a great number of users, so it's a lot of messages simultaneously. This can generate a queue; still, it's not too often that we have had issues with queues getting unmanageable long.

– RabbitMQ is a stable foundation for us and has worked smoothly since the beginning, leaving us with time to focus on other topics rather than fixing errors. If we get alerts regarding RabbitMQ, it's usually not RabbitMQ that is the issue, but some service in the back that is not sending messages or is sending too many messages, Alexander Lackner said.

PART THREE

PARKSTER: MONOLITHIC SYSTEM INTO MICROSERVICES

A growing digital parking service from Sweden is currently breaking down their monolithic application and working toward a microservices architecture. (Sept. 2017)

A PORTABLE PARKING METER IN YOUR POCKET

Founded in 2010, Parkster has become one of the fastest growing digital parking services in Sweden. Their vision is to make it quick and easy to pay parking fees with your smartphone via your Parkster app, with SMS, or with voice. They want to see a world where there is no need need to guesstimate the required parking time or stand in line waiting by a busy parking meter. It should be easy to pay for parking for everyone, everywhere. Moreover, Parkster doesn't want the customer to pay more when using its app, so that's why there are no extra fees when you are using Parkster for parking.

BREAKING UP A TIGHTLY-COUPLED MONOLITHIC APPLICATION

Like many other companies, Netflix among them, Parkster started out with a monolithic architecture. They wanted to prove their business model before they went further. A monolithic application means that the whole application is built as a single unit. All code for a system is in a single codebase that is compiled together and produces a single system.

Having one codebase seemed like the easiest and fastest solution at the time, and solved their core business problems, which included connecting devices with people, parking zones, billing, and payments. A few years later, they decided to break up the monolith into multiple small codebases, which they did through multiple microservices communicating via message queues.

Parkster tried out their parking service for the first time in Lund, Sweden. After that, they rapidly expanded into more cities and introduced new features. The core model grew and components became tightly coupled.

In the monolith system, deploying the codebase meant deploying everything at once. One big codebase made it difficult to fix bugs and to add new features. A deep knowledge was also required before attempting a single small code change, as no one wants to add new code that could disrupt operation in some unforeseen way. One day they had enough. The application had to be decoupled.

Application decoupling

Parkster's move from a monolith architecture to a microservice architecture is still a work in progress. They are breaking up their software into a collection of small, isolated services, where each service can be deployed and scaled as needed, independently of other services. Their system today has about 15-20 microservices, where the core app is written in Java.

Parkster is already enjoying the change, "It's very nice to focus on a specific limited part of the system instead of having to think about the entire system every time you do

something new or make changes. As we grow, I think we will benefit even more from this change," said Anders Davoust, a developer at Parkster.

Breaking down the codebase has also given the software developers freedom to use whatever technologies made sense for a particular service. Different parts of the application can evolve independently, be written in different languages, and/or maintained by separated developer teams. For example, one part of the system uses MongoDB and another part uses MySQL. Most code is written in Java, but parts of the system are written in Clojure. Parkster is using the open-source system Kubernetes as a container orchestration platform.

Resiliency - The capacity to recover quickly

Applications might be delayed or crash sometimes - it happens. It could be due to timeouts or errors in code that could affect the whole application.

Another thing the staff at Parkster likes about their system today is that it can still be operational even if part of the backend processing is delayed or broken. The entire sysem will not break just because one small part is not operating. Breaking up the system into autonomous components has meant that Parkster inherently becomes more resilient.

Message Queues, RabbitMQ and CloudAMQP

Parkster separates different components via message queues. A message queue may force the receiving application to confirm that it has completed a job and that it is safe to remove the job from the queue. The message will stay in the queue if anything fails in the receiving application. A message queue provides temporary message storage when the destination program is busy or not connected.

The message broker used between all microservices in Parkster is RabbitMQ. "It was a simple choice. We had used RabbitMQ in other projects before we built Parkster and we had a good experience with RabbitMQ." The reason they went for CloudAMQP was because they felt that CloudAMQP had more knowledge about the management of RabbitMQ. They simply wanted to put their focus on the product instead of spending days configuring and handling server setups. CloudAMQP has been at the forefront when it comes to RabbitMQ server configurations and optimization since 2012.

When asked what they like about CloudAMQP, the quick answer was, "I love the support that CloudAMQP gives us, always quick feedback and good help."

Now Parkster's goal is to get rid of the old monolithic structure entirely and focus on a new era where the whole system is built upon microservices.

PART THREE

FARMBOT: MACHINE-TO-MACHINE CHAT APPLICATION

FarmBot is an open-source robotic hardware kit designed for gardeners, researchers, and educators to interact with agricultural projects in a more efficient way. FarmBot uses physical sensors and actuators that require a bridge between the physical garden and the software layer. In 2017, user demand for real-time response to requests, shorter development time for new features, and a significant improvement in system up-time led Farmbot to RabbitMQ, the world's most popular open-source message broker.

FarmBot's Genesis version can plan, plant, and manage over 30 different crops such as potatoes, peas, squash, and more. Able to manage an area of 2.9 meters × 1.4 meters and a maximum plant height of 0.5 meters, FarmBot Genesis uses manual controls that users move and operate to perform a variety of tasks.

Thanks to the game-like interface that drags and drops plants into a map, FarmBot users immediately grasp the concept of controlling the entire growing season from start to finish. FarmBot tools and peripherals perform tasks such as watering plants, scaring birds away, taking photos of veggies, turning the lights on for a nighttime harvest, or simply impressing friends and neighbors!

On the other side of the coin, FarmBot identifies and eliminates weeds through image analysis via an onboard camera. To practice good conservation efforts, watering is done automatically after taking into account the degree of soil moisture and the weather forecast. There is a web app version of FarmBot to allow users to control garden tasks via smartphone or tablet.

ARCHITECTURE OVERVIEW

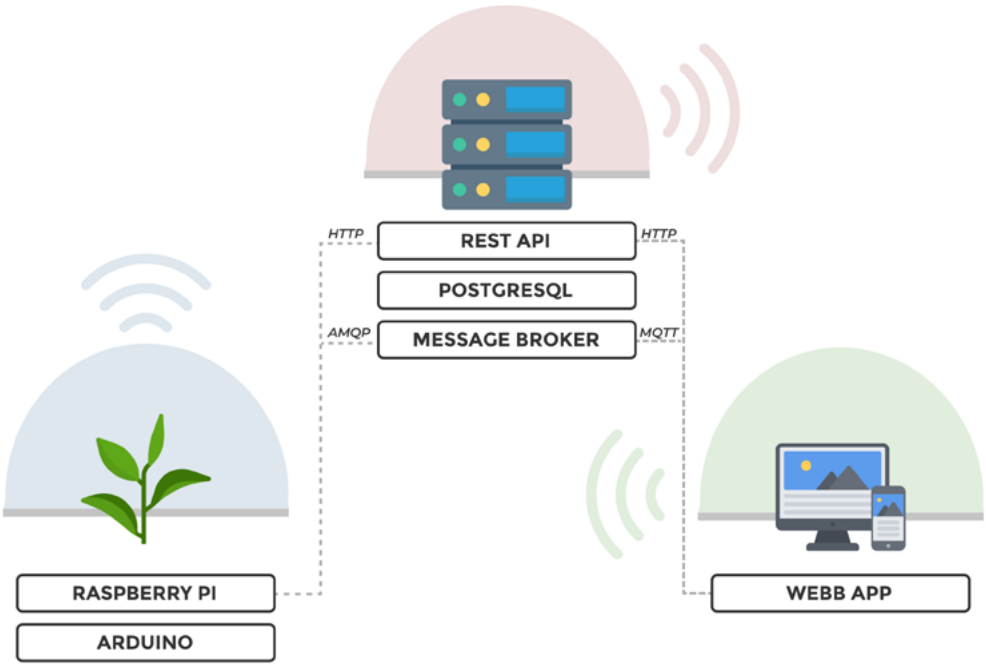


Figure 54 *- FarmBot Architectural Overview.*

FarmBot's architecture consists of three main parts:

1. The vegetable garden, Arduino microcontroller and Raspberry Pi (runs the management software and controls the Arduino CNC)
2. Web server handling requests between the field and the user
3. Web app (on mobile/iPad/browser etc.) that lets the user interact with the system

NOW FOR A DEEP-DIVE

Let's now take a closer look at all aspects of the system.

Arduino microcontroller (firmware) and Raspberry Pi (FarmBot OS)

The firmware, named Farmduino, is the software responsible for interaction with the real world. It runs on an Arduino microcontroller, written in C++ and executes sensor and actuator commands that allow users to drive the motors, the electro-irrigation valve, and read the probe measurements.

An example of this is when FarmBot turns on the water valve or probes for soil moisture. In this case, it is the device firmware that tells the peripherals directly when and how to operate. Other parts of the system may request access to peripherals by sending commands to the firmware.

A Raspberry Pi running FarmBot OS allows FarmBot to communicate with the web application over WiFi or ethernet to synchronize (download) sequences, regimens, farm designs, events, and more activity, including uploading logs and sensor data and accepting real-time commands. The Raspberry Pi also communicates with the Arduino to send commands and receive sensor and encoder data. It can take photos with a USB or Raspberry Pi camera, and upload the photos to the web application.

The REST API, PostgreSQL, and RabbitMQ

The FarmBot API provides an HTTP based REST API, which handles a number of responsibilities that prevent data loss between reflashes by storing it in a centralized database. This allows users to edit information when the device is offline and allows users to validate and control access to data via authentication and authorization mechanisms. It also sends email notifications (such as password resets and critical errors) to end-users. All other messaging is handled by a message broker - RabbitMQ, a distinctly decoupled sub-system of the Web API. The REST API does not control FarmBot - device control is handled by the Message Broker, CeleryScript and FarmBot JS.

RabbitMQ communicates to the devices in the field by AMQP and acts as a message queue to the backend services. MQTT protocol is used for real-time events to the frontend user interface.

Frontend user interface

The frontend user interface allows users to control and configure FarmBot from any desktop computer, laptop, tablet, or smartphone. Many users end up running their own web server on-premise since it's all open source. FarmBot also provides a publicly accessible server at https://my.farm.bot/ and a staging server at http://staging.farm.bot for end users that are not familiar with Ruby on Rails application development.

WHY RABBITMQ?

Some interactions do not lend themselves well to an HTTP request/response pattern, especially remote procedure calls. A device would be forced to perform long polling, constantly making HTTP requests to the API for any new remote procedure calls. Polling would create tremendous scalability issues for the web app and provide a sub-par real-time experience for users.

For example, emergency stop messages in FarmBot should be received as soon as they are created rather than the system constantly checking the API for messages like these. Other use cases include remote procedure calls and real-time data syncing.

RABBITMQ TO THE RESCUE

In 2017 the Farmbot users got the option to vote for new features to focus on. The vote fell down on auto-sync and better real-time support. This forced the Farmbot community to reevaluate the architecture, and it also made them try something new. Farmbot made the decision to transition their architecture to include RabbitMQ. In a unanimous decision by the FarmBot community, RabbitMQ was the obvious choice, in part because it has:

- Robust support of plugins such as WebSockets and MQTT
- Wide variety of wrapper libraries for Elixir, Ruby, and Typescript
- A good ecosystem, mind share, and ubiquity overall, e.g. it is not an obscure choice

RabbitMQ is now an important component of the FarmBot web API, where it handles various tasks including:

- Passing push notifications between users and devices
- Passing background messages between server background workers
- Uses a set of custom authorization plugins (to prevent unauthorized use)

Because RabbitMQ is a real-time message broker, there is no need to check for new messages. Messages, such as a user clicking the "move" button on the user interface, are sent back and forth between client, device, and server without initiating requests.

In many ways, the message broker acts as a machine-to-machine chat application. Any software package, whether it be the REST API, FarmBot OS or third-party firmware, can send a message to any other entity that is currently connected to the message broker, provided it has the correct authorization.

THE REST IS SUCCESS

Today, FarmBot is a happy, satisfied customer of the largest hosting provider of RabbitMQ, CloudAMQP. "The interactions with customer service have been quick and pleasant, and we've seen great uptime stats," said Rick Carlino, Lead Software Developer at FarmBot.

He continues, "We really don't have time to manage our own services on AWS and would much rather pay a premium to know that someone else is taking care of the problem if there is one."

Choosing the right technology and the right vendor often focuses on the best selection to optimize developer time. To address this, CloudAMQP provides support around the clock including built-in alarms that can be set up to detect issues with the server before it affects the business.

HEROKU - CLOUD APPLICATION PLATFORM

The same goes for the decision to host the production servers on Heroku. Even though there would be cost savings in hosting production a traditional AWS/Kubernetes setup, it's simply too much of a risk for a small company to save only a couple hundred dollars a month. FarmBot's Carlino added, "Every hour I spend thinking about infra is an hour that I am not spending on feature development."

THE CLOUDAMQP ADVANTAGE

User demand for real-time response to requests, shorter development time for new features, and a significant improvement in system up-time all led FarmBot to choose RabbitMQ and CloudAMQP.

As the leading hosting provider of RabbitMQ, CloudAMQP was able to help Farmbot to quickly format their old architecture into a message queue based system, transforming the agri-tech app into a farming powerhouse.

As FarmBot leads the way in the innovation of how food is grown, CloudAMQP is leading the RabbitMQ revolution through user success stories like this one. Let us know if it's time to create your own successful message queue architecture? Get in touch with our friendly team of professionals today to determine the right RabbitMQ plan for your future growth.

PART THREE

CLOUDAMQP: MICROSERVICE ARCHITECTURE BUILT ON RABBITMQ

The CloudAMQP team relies on RabbitMQ in our everyday life; in fact, a huge number of our events in the production environment pass through RabbitMQ. This chapter gives a simple overview of the automated process behind CloudAMQP, the polyglot workplace where microservices written in different languages communicate through RabbitMQ.

CloudAMQP never had a traditional monolithic set up. It is built from scratch on small, independent, manageable services that communicate with each other known as micro-services. These microservices are all highly decoupled and focused on their specific task. This chapter gives an overview and a deeper insight into the automated process behind CloudAMQP, describing some of our microservices and the use of RabbitMQ as a message broker communicating between them.

BACKGROUND OF CLOUDAMQP

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

THE AUTOMATED PROCESS BEHIND CLOUDAMQP

CloudAMQP is built upon multiple small microservices, where RabbitMQ is used as a messaging system. RabbitMQ is responsible for the distribution of events to the services that listen for them. A message can be sent without having to know if another service is able to handle it immediately or not. Messages can wait until the responsible service is ready. A service publishing a message does not need to know anything about the inner workings of the services that process that message. The pub-sub (publish-subscribe) pattern is followed, as is the retry upon failure process.

Creating a CloudAMQP instance provides the option to choose a plan and to decide how many nodes to have. The cluster will behave a bit differently depending on the cluster setup. The option to create your instance in a dedicated VPC and select RabbitMQ version is also available.

A dedicated RabbitMQ instance can be created via the CloudAMQP control panel, or by adding the CloudAMQP add-on from any of our integrated platforms, like Heroku, AWS marketplace, or Azure marketplace.

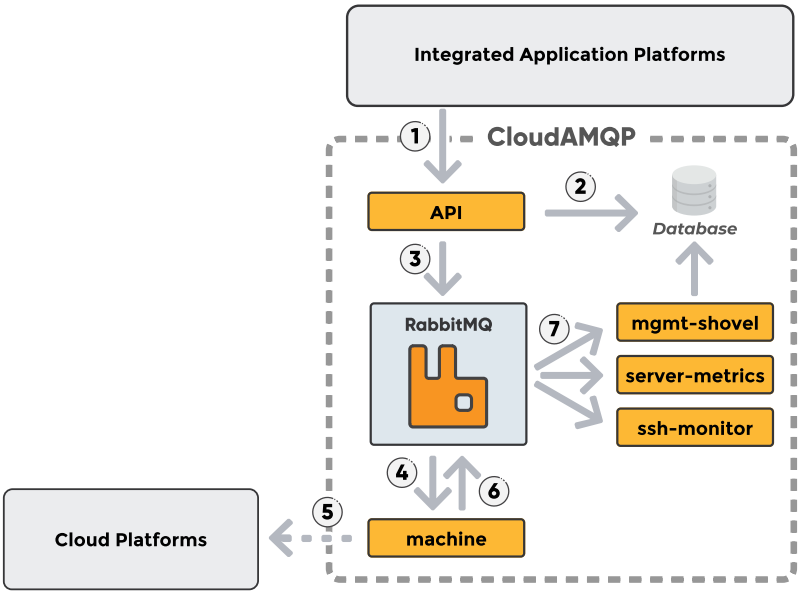


***Figure 55** - The automated process behind CloudAMQP.*

When a client creates a new dedicated instance, an HTTP request is sent from the reseller to a service called CloudAMQP-API (1). The HTTP request includes all information specified by the client: plan, server name, data center, region, number of nodes etc., as shown in Figure 55 above. CloudAMQP-API handles the request, saves information into a database (2), and finally, sends a account.create-message to one of our RabbitMQ-clusters (3).

Another service, called CloudAMQP-machine, subscribes to account.create. CloudAMQP-machine takes the account.create-message from the queue and performs actions for the new account (4).

CloudAMQP-machine triggers multiple scripts and processes. First, it creates the new server(s) in the chosen datacenter via an HTTP request (5). Different underlying instance types are used, depending on data center, plan, and number of nodes. CloudAMQP-machine is responsible for all configuration of the server, setting up RabbitMQ, mirror nodes, handle clustering for RabbitMQ etc., all depending on the number of nodes and chosen datacenter.

CloudAMQP-machine send an account.created-message back to the RabbitMQ cluster once the cluster is created and configured. Then the message is sent on the topic exchange (6). The great thing about the topic exchange is that multiple services can subscribe to the event. There are several services listening to account.created-messages (7), all of which will set up a connection to the new server. Here are three examples of services receiving a message and working toward new servers.

Figure 56 - The microservice CloudAMQP-put-metrics.

CloudAMQP-server-metrics: Continuously gathers server metrics such as CPU and disk space from all servers.

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

CloudAMQP-SSH-monitor: Monitors the many process required on all servers.

CloudAMQP offers many other services communicating with those described above and with new servers created for the client.

CloudAMQP-server-metrics

CloudAMQP-server-metrics collects metrics (CPU/memory/disk data) for all running servers. The collected metric data is sent to RabbitMQ queues defined for the specific server e.g., server.<hostname>.vmstat and server.<hostname>.free, where hostname is the name of the server.

CloudAMQP-alarm

Different services subscribe to server metrics data. One of these services is called CloudAMQP-alarm. CloudAMQP-alarm checks the server metrics from RabbitMQ against the alarm thresholds for the given server, and notifies the owner of the server if needed. Users are able to enable/disable alarms such as CPU or memory, for example, as they see fit.

CloudAMQP-put-metrics

CloudAMQP integrates the monitoring tools DataDog, Logentries, AWS Cloudwatch, Google Cloud Stackdriver Logging, and Librato, that are user enabled. Our microservice CloudAMQP-put-metrics checks the server metrics subscribed from RabbitMQ and is responsible for sending metrics data to tools that the client has integrated.

PART THREE

SOFTONIC: EVENT-BASED COMMUNICATION

Softonic, a software and app discovery portal, is accessed by over 100 million users per month and delivers more than 2 million downloads per day, with a constant flow of events and commands between services. As the world's largest software and app discovery destination, Softonic is also one of the world's most highly-trafficked websites.

Even without realizing it, you have probably landed on their website when downloading software or an application. Over 100 million users per month rely on Softonic as an app guide that assists with the discovery of the best applications for any device, and that also serves up reviews, news, articles, and free downloads.

CloudAMQP provides hosted RabbitMQ clusters in the biggest data centers around the world and Softonic is one of our valued customers. The CloudAMPQ team met up with Riccardo Piccoli, a developer at Softonic, at the RabbitMQ Summit 2018 in London where he kindly shared Softonic's customer story with us.

This article is broken down into two parts; the first part is an overview of the system, which shows a simple RabbitMQ use cases of an event-based architecture. The second part is a deep-dive into the internal architecture in Softonic and the plugins used by the company along with examples of events they are sending.

A SIMPLE RABBITMQ USE CASE

Users are able to upload files to Softonic. The system first scans the uploaded file for viruses and collects basic information. After the information is collected, the file is ready for distribution to other users.

The new binary data is first held within a dedicated service, and a notification about the upload is sent to an event bus. Other services collect this information which is eventually added to the Softonic website. In this case, the user gets notified immediately after the upload has succeeded and a scanning event is simply placed on an event-bus for other services to handle.

The message queue, in this section called an event-bus, allows web servers to respond to requests quickly instead of being forced to perform a resource-heavy process on the spot, which could cause user wait-time issues. Virus scanning is an example of a resource-heavy process. The virus scanning application takes a message from the event bus, such as a "ScanFile" command, and starts processing. At the same time, other users are able to upload new files to Softonic and processing tasks are able to join the queue. The event "FileScanned" is added back to the event bus, once the resource-consuming application has handled the event.

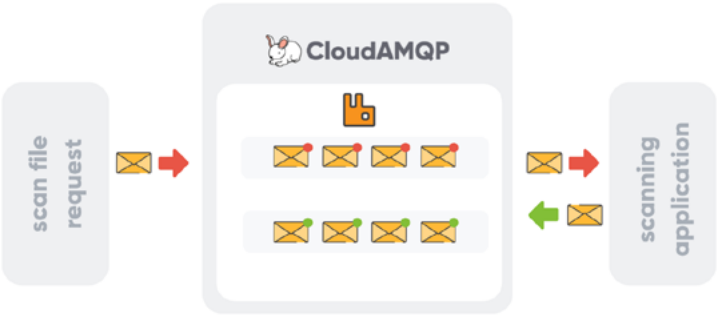


Figure 57 *- Scan file request.*

Figure 58 *- Softonic RabbitMQ architecture.*

Architecture like this creates two simple services and low coupling between the sender and the receiver. Users can still upload files, even if the scanning application is busy or is under maintenance.

1. Different events or commands are published to the event bus, e.g., a "ScanFile" command.

2. Softonic uses RabbitMQ as an event bus, wherein events or commands are simply added to the queue.

3. The resource-consuming application retrieves the event and starts to process it, while some data is stored to the database and more events can be published back to another event queue (more about this in "Internal Structure of RabbtiMQ").

4. The resource-consuming application is able to store a great deal of information in a database (MySQL).

When a microservice receives an event, it can update its own status, which leads to more events being published, which is the case here.

THE INTERNAL STRUCTURE OF RABBITMQ

It's time for a deep-dive into the internal architecture of RabbitMQ and into the Softonic application. Softonic is using the Consistent Hash Exchange Plugin and RabbitMQ Sharding.

Image description external usage: Softonic services are built upon Node.js and PHP and communicate with the RabbitMQ event bus, from which information from the services are transferred from a PHP application to a MySQL event store.

Image description internal usage: Information from the first application retrieves data from the MySQL Event Share and pushes it through consistent hash exchanges in two internal RabbitMQ event buses using sharded queues. From there, the information reaches the orchestration layer and an elasticsearch cluster, where it becomes visible for users.

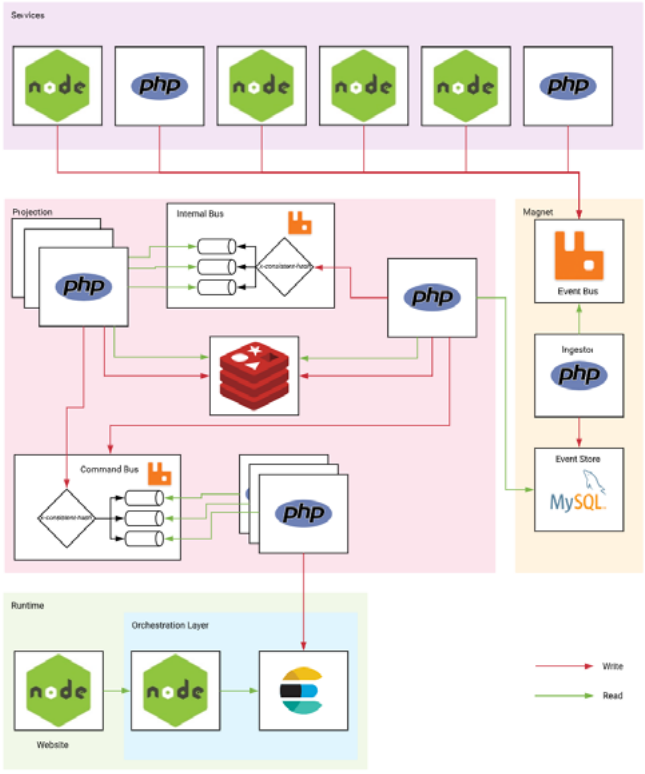


Figure 59 *- Softonic internal architecture.*

The consistent hash exchange plugin and RabbitMQ sharding

The consistent hash exchange plugin has the task of load balancing messages between queues. Messages sent to the exchange are consistently and equally distributed across many queues based on the routing key of the message. The plugin creates a hash of the routing key and distributes the messages between queues that have a binding to that exchange.

The RabbitMQ sharding plugin partitions queues automatically. Once you define an exchange as sharded, the supporting queues are automatically created on every cluster node and messages are sharded across them. RabbitMQ sharding shows one queue to the consumer but could be many queues running in the background. .

An example sequence of Softonic events and commands

Below is an example of events and commands sent via RabbitMQ by Softonic using the consistent hash exchange plugin. Events 1 and 2 end up in the same queue (with the order preserved) while event 3 may or may not end up in the same queue. Data is sharded, and processed with consistent hashing F(id_program) in order to preserve order by the program.

- Event 0: Create category "antivirus" (name: "antivirus")
- Event 1: Create program A (name "foobar", category "antivirus", developer "softonic")
- Event 2: Create a review for program A
- Event 3: Create program B (name "foo", category "antivirus", developer "84codes")
- Event 4: Update category "antivirus" name to "Antivirus"

In this example, event 0 and event 4 need to be processed synchronously, while events 1, 2, and 3 can be processed asynchronously. Event 0 will be processed immediately and events 1,2, and 3 will be re-published to the queue so that other sharded consumers can process them.

CloudAMQP - Message Queueing as a Service

Softonic did run RabbitMQ in-house before moving to the cloud. The biggest reason for choosing CloudAMQP as a provider was for simplicity of installing RabbitMQ without the hassle of maintaining a RabbitMQ cluster.

CloudAMQP offers many different plans designed for different uses, including a free plan with Little Lemur.

The CloudAMQP team is grateful for the information from Softonic, an impressive example of microservice architecture. A special thanks to Riccardo for your time at the RabbitMQ Summit 2018, hope to see you again at the next event. We wish Softonic thebest of luck with their continued success!

PART THREE

REVER: SOLVING ISSUES IN MANUFACTURING WITH RABBITMQ

Having had infrastructural problems in the past made Luis Elizondo, Chief Technology Officer of Rever, even more satisfied with not having to deal with trouble in the present. “RabbitMQ has been very stable and consistent since the beginning. It is an extremely critical component for us.”

On the market since 2016, Rever identifies and solves issues connected to workflow, mainly in manufacturing, and has grown steadily to now be available in 40 countries and 16 languages, engaging 100,000+ users. Rever’s users bring the application with them to the factory floor via phone, tablet or laptop, and all employees can report their observations on safety hazards, quality problems, maintenance issues and improvement ideas in real-time.

“Basically, what we do is help companies find permanent solutions to their problems, and we can do so thanks to our users reporting issues found while performing different work-related tasks,” Luis Elizondo, Chief Technology Officer at Rever, said.

Rever’s whole infrastructure is built with RabbitMQ as a base, handling hundreds of thousands of messages back and forth between 15 different microservices daily. Tasks that demand a reliable backbone that assures everything runs smoothly. "And it does! In fact, we have only had minor problems since we migrated to CloudAMQP.”

Elizondo provided a brief outline of what RabbitMQ is handling behind the scenes of Rever: “Audit logs, notifications, searches, information about users, reports generated in the platform, all of the intelligent insights that live in a microservice. All of it runs through RabbitMQ.”

Can you give an example of when a message is sent through RabbitMQ?

“Our search engine lives in a different microservice. We need to send the request to the microservice in charge of the search mechanism, wait until the search is completed, and send the response back to our API so it can communicate with the client and send the results. That type of communication is happening all of the time.”

“There are also a lot of operations that we just trigger and we don't want to wait for a response. Let’s say there’s a new comment in the community, then we need to send push notifications or emails to multiple users simultaneously. We trigger a message and there’s a particular service in our architecture that is going to handle all of those notifications. The way we do it is again through RabbitMQ. We have a public API so essentially every request that comes from the client will go through the API and from there into the microservice. And back again through RabbitMQ.”

Elizondo is happy with CloudAMQP as a provider, saying,

"There will always be some problems, but so far its only been minor ones. Overall, it has been very stable with you guys."

PART THREE

TRUSTT: AUTOMATED E-MAIL SERVICE WITH RABBITMQ

If you think you're having a hard time keeping up with your company's growth, check out Trustt.io. Thanks to RabbitMQ, Trustt went from sending 1,800 emails per day to 70,000 by switching several of its processes from a synchronous system. Read on to learn how automating emails and avoiding bottlenecks was a gamechanger for Trustt.

Trustt is a SaaS platform that accelerates brand growth with solutions for recruiting and engaging trusted communities: authentic review generation and user-generated content, recommendations and influence marketing, and market research. To keep candidates well informed, Trustt sends emails automatically throughout the lifecycle.

Trustt's platform generates more than 20 types of emails, including non-selection emails, selection and reception reminder emails, content creation emails, and more. Multiply all these types by the number of candidates and client campaigns, and you can understand the exponential volume of emails they have to send.

The goal was to keep up with Trustt's hyper-commercial growth. Trustt migrated its email system to include asynchronous RabbitMQ processes with temporary queues to absorb the load induced by the consumers. Today they have also implemented several other application services on this model. They have gone from one process using RabbitMQ to seven in a few months on several of their products.

Today, Trustt works with a consumer to process email requests. With additional consumers, the number of processes can quickly be increased on demand. The previous email system was in synchronous mode and caused bottlenecks within their processes.

"RabbitMQ has several advantages in our application stack. It allows us to better control the load used on processing and to spread it over time. RabbitMQ has become one of the essential building blocks of our infrastructure to ensure the scalability required to support our growth," says Nadia, CEO of Trustt.

"We chose CloudAMQP because we didn't want to set up our own RabbitMQ instances and manage all the inherent maintenance and upgrades, plus we need two instances of RabbitMQ; one instance for our development and staging environment and one instance for production. With CloudAMQP we can adapt our plans to our future use."

Trustt uses RabbitMQ to manage interactions with multiple services or other external web services. A customer success management software takes customer data from the existing tech stack and automatically combines it to get a full view of the customer. Payloads are sent containing UUIDs to RabbitMQ. The consumer receives the UUID and looks for the necessary information to process the data. Below is an example of a message payload sent between Trustt's services, through RabbitMQ.

```
{
  “nodeId:78966,
  “params”:[],
  “userID”: 70627,
  “eventId”: 0,
  ”uuid_message”: ”e251710c-ac18-11-ec…”,
  ”dateSendBefore”: {
    “date”: ”2022-10-22 09:53:21.779867”,
    “timezone_type”: 3,
    “timezone”: ”Europe\/Brussels”
  }
}
```

In order not to slow down the tool, the synchronization is done through a RabbitMQ process. The events are sent to a queue and a consumer processes them in the background.

“Trustt chose CloudAMQP because the service configuration is automated, leaving our engineering team to focus on the integration of the business processes and not on the implementation and monitoring.”

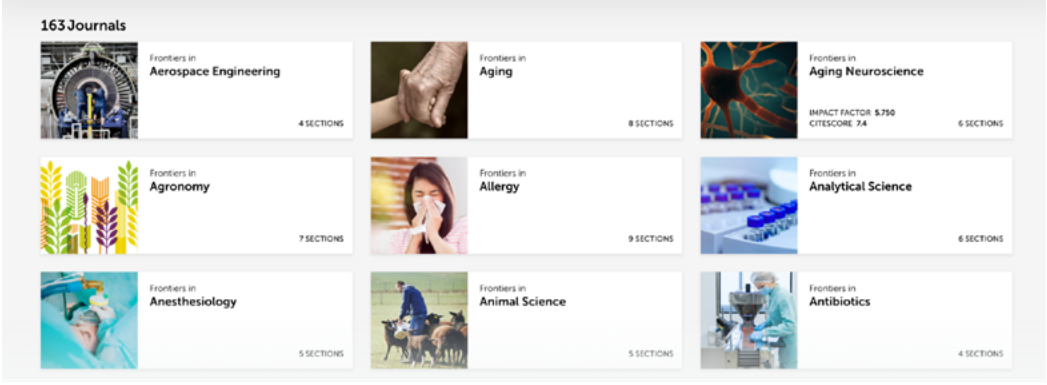
PART THREE

FRONTIERS: SCIENTIFIC DISCOVERIES IN PUBLIC

Frontiers has made it its mission to make scientific publications free to access and available for everyone. A quest in which RabbitMQ plays an important role behind the scenes.

Consistently throughout history, science has been the key player in taking humanity forward thanks to great discoveries. Everyone has seen the impact essential milestones like the invention of vaccines and antibiotics have had, from the individual perspective to humanity as a whole. Frontiers' primary mission is to make all science open and freely accessible so that important scientific discoveries can be shared fast and nationwide for the benefit of all.

Today, Frontiers is one of the most-cited and fastest-growing Open Access Publishers and Open Science Platforms globally. As of June 2022, they have passed 273,000 published articles, viewed, and downloaded over 1,4 billion times.



Frontiers make the publications on their platform free to read. They have turned the legacy model upside down, in which scientists had to pay to access research through a subscription. Instead, at Frontiers, publishing costs are offset by a one-off fee, and then the publications are freely and immediately accessible to anyone who would like to read them.

-"When scientists make new discoveries, they need a platform to share their publications. So first, they submit their publications to us, then we oversee a peer review where other scientists validate them. Then, finally, they get published on our platform," said Marc Bettex, Software Development Manager at Frontiers.

Frontiers started based on monolithic architecture. They described their early days this way:

- "One application with everything gathered inside. We were already a lot of engineers and were hitting the limits of this model. We wanted to translate to a microservice architecture to have many different applications that can be developed by independent teams, which would enable us to grow. The problem is that we needed to communicate between these applications, and this is where RabbitMQ and CloudAMQP came into the picture."

Frontiers started using RabbitMQ in-house but didn't have someone in the company who knew RabbitMQ that well. They quickly realized that they didn't know how to set up a bulletproof infrastructure. That's when they turned to CloudAMQP.

Can you give us an example of how RabbitMQ plays a part in your architecture?

- "We send messages that are like commands. For example, let's say we want to move an article from one stage to another. So we send a message, then other applications receive it, take action, and follow up on the process."

The figure below shows an example of a message flow in Frontiers architecture.

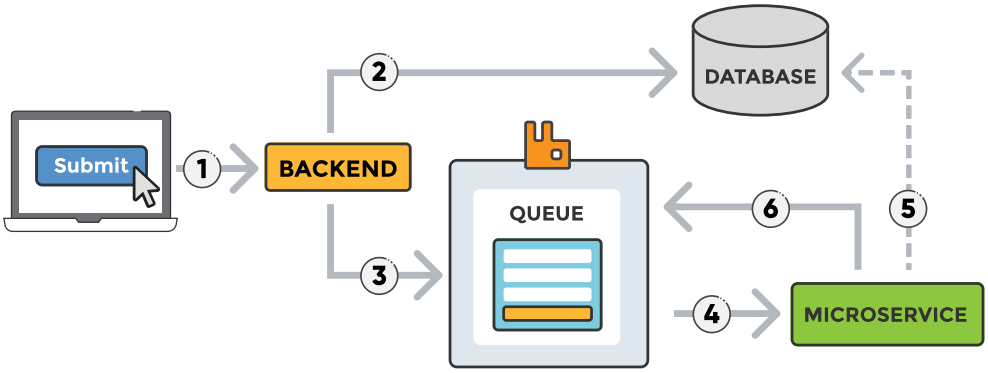


Figure 6o - Example of a message flow through RabbitMQ.

1. A user clicks "Submit an article", causing the browser to make an http request to the backend.

2. The backend saves the status in the database.

3. The backend sends an "Initiate review process" message to RabbitMQ.

4. Another microservice consumes the message from the queue and stores additional information in the database, like an article owner.

5. The microservice also begins processing through RabbitMQ with new tasks such as "execute basic checks" and "suggest editor and reviewers for the article".

"We also synchronize data with RabbitMQ. So, for instance, when a system does something, maybe update data with some user, we can broadcast a message saying 'this user is updated.' All the other systems can listen to that message and get the new user data."

Why are microservices the best solution for Frontiers?

"It allows the organization to scale. Different teams can work on different microservices simultaneously, on different code bases, without stepping on each other's toes."

What do you think about CloudAMQP's service?

"Overall, we have been delighted. Of course, we have had some minor hiccups from the beginning, but that was more related to being new with RabbitMQ and making beginner mistakes. Other than that, RabbitMQ has been very stable, in particular CloudAMQP and the infrastructure you provide to us."

"If there's one thing that we are very pleased with CloudAMQP, it's the support. Always speedy answers in the time of need. We don't need you when everything is okay, which is almost always. It's almost like you don't exist. And then when we need you, you are there, no matter what hours. Also, one time when I needed help, I contacted the support, and I got in contact with Carl Hörberg. I looked him up on your website and realized he's the company's founder. I thought that was a nice touch, him helping out in support matters. We have other cloud providers too, but your support experience is different. You should be proud of yourself!"

TIMES ARE CHANGING!

Reliability and scalability are more important than ever. Therefore companies are rethinking their architecture. Monoliths are evolving into microservices and servers are moving into the cloud. Message Queues and especially RabbitMQ has come to play a significant role in the growing world of microservices. After all, it's one of the most widely deployed open source message brokers.

This book will help you along your RabbitMQ journey. It consists of three main parts:

- Introduction to Message Queueing and RabbitMQ
- Advanced Message Queueing with RabbitMQ
- User Stories

Essentially, this book is about RabbitMQ;
and who knows about queueing better than a bunch of Swedes*?

* In case you haven't heard: queueing is an integral part of the Swedish everyday life.

This book is produced by the Swedish tech company, 84codes AB, which provides the service CloudAMQP — RabbitMQ as a service.