**RabbitMQ**

**(Open Source Enterprise Messaging)**

**RabbitMQ**

* + RabbitMQ is a message-queueing software called a message broker or queue manager.
  + It is a software where queues can be defined, applications may connect to the queue and transfer a message onto it.
  + It is a way of exchanging data between processes, applications, and servers.

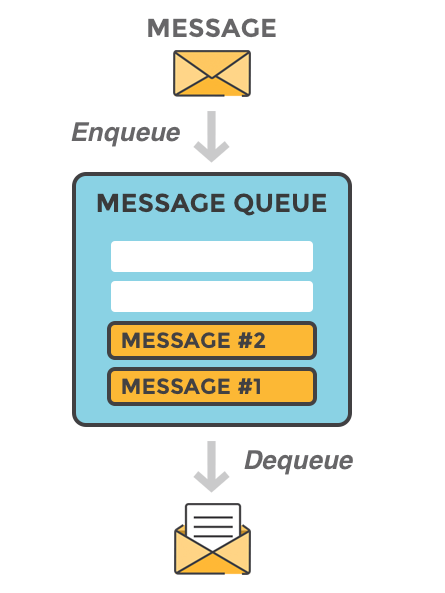


Fig 1. Message Queue

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

**RabbitMQ Example**

* A message brooking solutions can act like a middleman for various services (e.g. a web application, as in this example).
* They can be used to reduce loads and delivery times by web application servers since tasks, which would normally take quite a bit of time to process, can be delegated to a third party whose only job is to perform them.
* Consider a scenario where a web application allows users to upload information to a web site. The site will handle this information and generate a PDF and email it back to the user.
* Handling the information, generating the PDF and sending the email will in this example case take several seconds and that is one of the reasons of why a message queue will be used.
* When the user has entered user information into the web interface, the web application will put a "PDF processing" - task and all information into a message and the message will be placed onto a queue defined in RabbitMQ.



Fig 2. Workflow of RabbitMQ

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

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

**When and why should you use RabbitMQ?**

* Message queueing allow web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures on the spot.
* Message queueing is also good when you want to distribute a message to multiple recipients for consumption or for balancing loads between workers.
* The consumer can take a message of the queue and start the processing of the PDF at the same time as the producer is queueing up new messages on the queue.
* The consumer can be on a totally different server than the publisher, or they can be located on the same server.
* Request can be created in one programming language and handled in another programming language - the two applications will only communicate through the messages they are sending to each other. Due to that, the two applications will have a low coupling between the sender and the receiver.

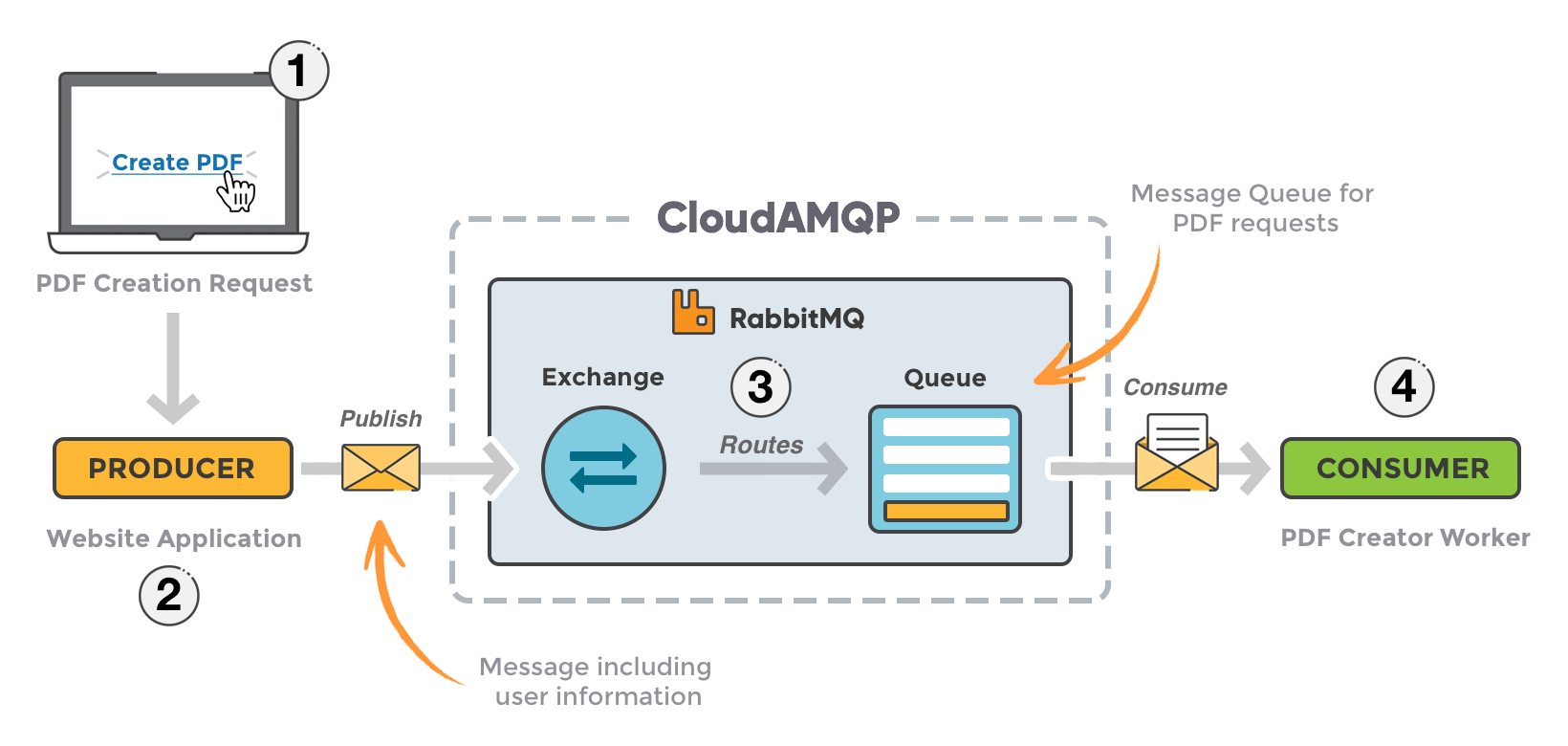


Fig 3. Overview of RabbitMQ

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

**Message flow in RabbitMQ**

1. The producer publish a message to an exchange. When you create the exchange, you have to specify the type of it.
2. The exchange receives the message and is now responsible for the routing of the message. The exchange take different message attributes into account, such as routing key, depending on the exchange type.
3. Bindings have to be created from the exchange to queues. In this case we see two bindings to two different queues from the exchange. The Exchange routes the message in to the queues depending on message attributes.
4. The messages stay in the queue until they are handled by a consumer
5. The consumer handles the message.

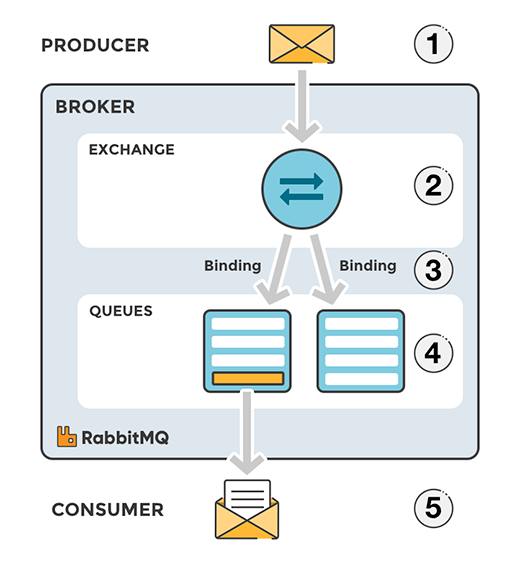


Fig 4. Message flow in RabbitMQ

**EXCHANGES**

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

**Exchange Types**

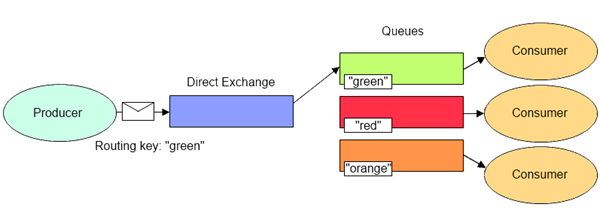
Exchanges control the routing of messages to queues. Each exchange type defines a specific routing algorithm which the server uses to determine which bound queues a published message should be routed to.

RabbitMQ provides four types of exchanges:

* Direct
* Fan-out
* Topic
* Headers.

**Direct Exchanges**

The Direct exchange type routes messages with a routing key equal to the routing key declared by the binding queue.  The following illustrates how the direct exchange type works:

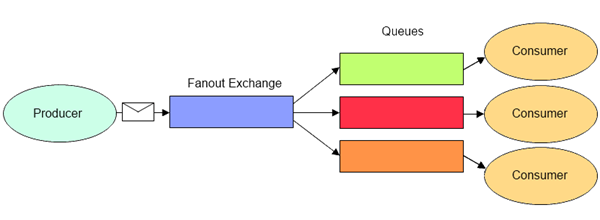
[](http://lostechies.com/derekgreer/files/2012/03/DirectExchange1.png)

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

* This is the type of exchange that was used in our Hello World example.  As discussed in part 3 of our series, every queue is automatically bound to a default exchange using a routing key equal to the queue name.  This default exchange is declared as a Direct exchange.
* In the above mentioned example, the queue named “hello-world-queue” was bound to the default exchange with a routing key of “hello-world-queue”, so publishing a message to the default exchange (identified with an empty string) routed the message to the queue named “hello-world-queue”.

**Fanout Exchanges**

The Fanout exchange type routes messages to all bound queues indiscriminately.  If a routing key is provided, it will simply be ignored.  The following illustrates how the fanout exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/FanoutExchange2.png)

 The Fanout exchange type is useful for facilitating the publish-subscribe pattern.

* When using the fanout exchange type, different queues can be declared to handle messages in different ways.
* For instance, a message indicating a customer order has been placed might be received by one queue whose consumers fulfill the order, another whose consumers update a read-only history of orders, and yet another whose consumers record the order for reporting purposes.

**Topic Exchanges**

* The Topic exchange type routes messages to queues whose routing key matches all, or a portion of a routing key.  With topic exchanges, messages are published with routing keys containing a series of words separated by a dot (e.g. “word1.word2.word3”).
* Queues binding to a topic exchange supply a matching pattern for the server to use when routing the message.  Patterns may contain an asterisk (“\*”) to match a word in a specific position of the routing key, or a hash (“#”) to match zero or more words.
* For example, a message published with a routing key of “honda.civic.navy” would match queues bound with “honda.civic.navy”, “\*.civic.\*”, “honda.#”, or “#”, but would not match “honda.accord.navy”, “honda.accord.silver”, “\*.accord.\*”, or “ford.#”.
* The following illustrates how the fanout exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/TopicExchange2.png)

* The Topic exchange type is useful for directing messages based on multiple categories (e.g. product type and shipping preference ), or for routing messages originating from multiple sources (e.g. logs containing an application name and severity level).

**Headers Exchanges**

* The Headers exchange type routes messages based upon a matching of message headers to the expected headers specified by the binding queue.
* The headers exchange type is similar to the topic exchange type in that more than one criteria can be specified as a filter, but the headers exchange differs in that its criteria is expressed in the message headers as opposed to the routing key, may occur in any order, and may be specified as matching any or all of the specified headers.
* The following illustrates how the headers exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/HeadersExchange2.png)

* The Headers exchange type is useful for directing messages which may contain a subset of known criteria where the order is not established and provides a more convenient way of matching based upon the use of complex types as the matching criteria (i.e. a serialized object).

**RABBITMQ AND SERVER CONCEPTS**

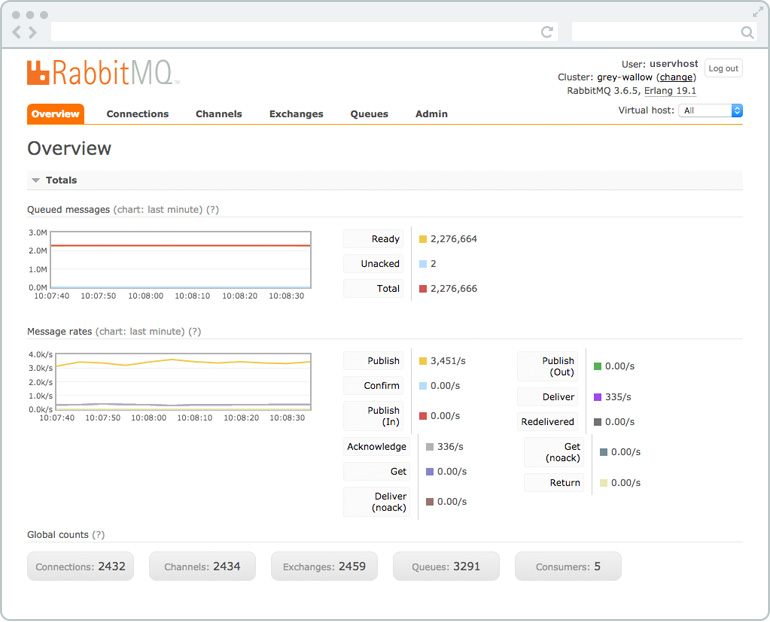
Here are some important concepts that needs to be described before we dig deeper into RabbitMQ. The default virtual host, the default user, and the default permissions are used in the examples that follow, but it is still good to have a feeling of what it is.

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

**THE MANAGEMENT INTERFACE - MANAGEMENT AND MONITORING**

RabbitMQ provides a web UI for management and monitoring of your RabbitMQ server. The RabbitMQ management interface is enabled by default in CloudAMQP and a link can be found on the details page for your CloudAMQP instance.

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



More information about the management interface can be found in [Part 3 - The management interface.](https://www.cloudamqp.com/blog/2015-05-27-part3-rabbitmq-for-beginners_the-management-interface.html)

**STEPS TO PUBLISHING A MESSAGE/CONSUMING A MESSAGE:**

RabbitMQ speaks a protocol called AMQP by default. To be able to communicate with RabbitMQ you need a library that understands the same protocol as RabbitMQ. You need to download the client-library for the programming language that you intend to use for your applications.

A client-library is an applications programming interface (API) for use in writing client applications. A client library has several methods that can be used, in this case to communicate with RabbitMQ.

The methods should be used when you, for example, connect to the RabbitMQ broker (using the given parameters, host name, port number, etc) or when you declare a queue or an exchange. There is a choice of libraries for almost every programming language.

**STEPS TO DO:**

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

References for example:

1. <http://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html>
2. <http://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html>
3. <http://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html>