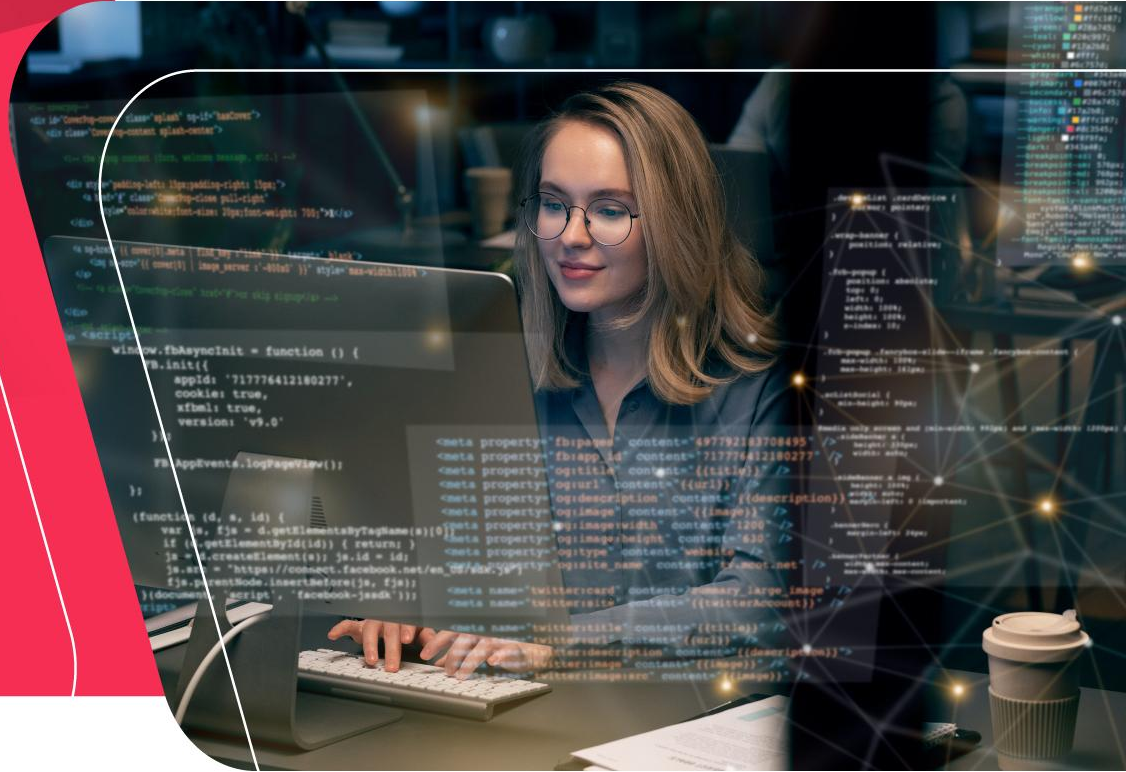


# Java Bootcamp

Day 39



- JDK 8/11/15
- JRE 8/11/15
- **IntelliJ IDEA Community Edition**
- JAVA 3<sup>rd</sup> Party Library (Network, DB, etc)
- JAVA Framework (Spring Boot & Spring JDBC)
- **MySQL Server Community**

# RabbitMQ



- **RabbitMQ** is an **AMQP** messaging broker and it is the most popular open source and cross-platform message broker.
- RabbitMQ is also a way to exchange the data between different platform applications such as a message sent from **.Net** application can be read by a **Node.js** application or **Java** application.
- The RabbitMQ is built on **Erlang** general-purpose programming language and it is also used by WhatsApp for messaging.

- The Advanced Message Queuing Protocol (**AMQP**) is an open standard application layer protocol for message-oriented and the features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.
- It was developed by JPMorgan and iMatix Corporation. AMQP was designed with the following main characteristics as goals:
  - Security
  - Reliability
  - Interoperability

- RabbitMQ is lightweight and easy to deploy on available premises and it supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.
- Following is the pictorial representation of how RabbitMQ will act as a mediator between sender &



- Now a days most people will perform a multiple tasks in single application like sending emails or SMS, reports and it will create a heavy load on application so if you separate these tasks, then we will get more space (memory) to serve more requests.
- By using RabbitMQ, we can remove some heavy work from our web applications such as sending a reports in Excel or Pdf format's or sending an email, SMS or another task such as trigger some other applications to start processing.
- RabbitMQ is an open source and cross-platform message broker so it's easy to use with many languages such as .Net, Java, Python, Ruby, Node.Js.

- RabbitMQ will support multiple operating systems and programming languages.

RabbitMQ has provided a various client libraries for following programming languages.

- .Net
- **Java**
- Spring Framework
- Ruby
- Python
- PHP
- Objective-C and Swift
- JavaScript
- GO
- Perl



- Here we will learn how to setup or install Erlang and RabbitMQ on windows machines and how to enable RabbitMQ web management plugin in step by step manner.
- The RabbitMQ is built on **Erlang** runtime environment so before we install RabbitMQ, first we need to download and install **Erlang** in our machines.

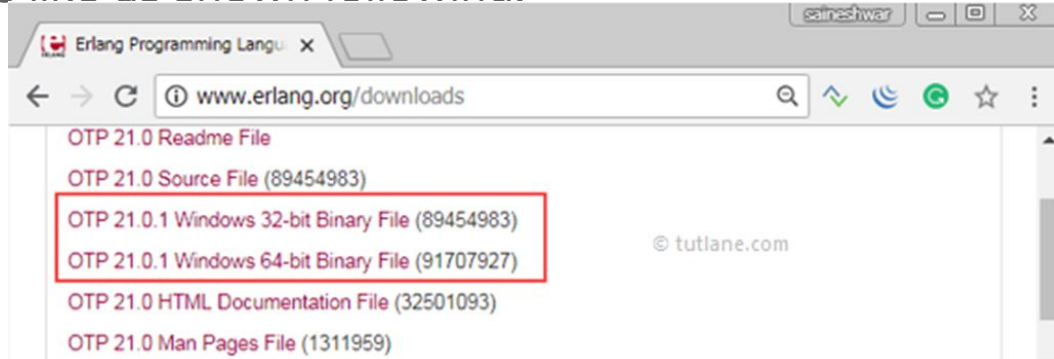
- Here we will learn how to setup or install Erlang and RabbitMQ on windows machines and how to enable RabbitMQ web management plugin in step by step manner.
- The RabbitMQ is built on **Erlang** runtime environment so before we install RabbitMQ, first we need to download and install **Erlang** in our machines.

- **Erlang** is a general-purpose programming language and runtime environment.
- Erlang has built-in support for concurrency, distribution and fault tolerance. Erlang is used in several large telecommunication systems from Ericsson.

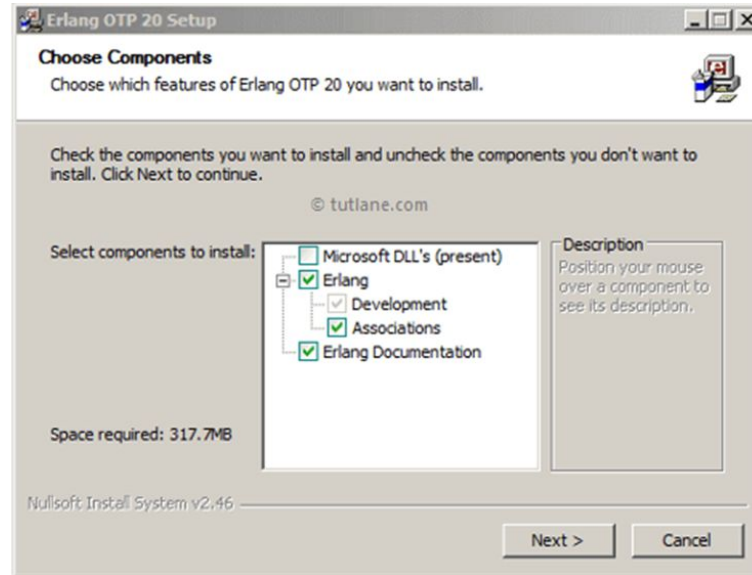
- Following is the link to download and install Erlang on your windows machine.

<http://www.erlang.org/downloads>

- Once you open above URL, select a required version to download and install on your machines like as shown following.



- After downloading the required **Erlang** version, now we will install Erlang for that double click on downloaded install file and follow the instructions like as shown below.



- After completion of Erlang installation, now we will install RabbitMQ on windows for that download a RabbitMQ setup file from following link for windows.

<https://www.rabbitmq.com/download.html>

- After opening the above URL, we will get a RabbitMQ setup files information like as shown below. Here we are going to install a RabbitMQ server on windows machine so we will download the windows version like as shown below.

## Downloading and Installing RabbitMQ

The latest release of RabbitMQ is **3.7.7**. For release notes, see [changelog](#).

### RabbitMQ Server

© tutlane.com

#### Installation guides

Linux, BSD, UNIX: [Debian](#) | [Ubuntu](#) | [RHEL](#) | [CentOS](#) | [Fedora](#) | [Generic Unix](#) | [Solaris](#)

Windows: [With installer \(recommended\)](#) | [Manual](#)

MacOS: [Homebrew](#) | [Standalone](#) | [Generic Unix](#)

[Erlang/OTP for RabbitMQ](#)

- After completion of Erlang installation, now we will install RabbitMQ on windows for that download a RabbitMQ setup file from following link for windows.

<https://www.rabbitmq.com/download.html>

- After opening the above URL, we will get a RabbitMQ setup files information like as shown below. Here we are going to install a RabbitMQ server on windows machine so we will download the windows version like as shown below.

## Downloading and Installing RabbitMQ

The latest release of RabbitMQ is **3.7.7**. For release notes, see [changelog](#).

### RabbitMQ Server

© tutlane.com

#### Installation guides

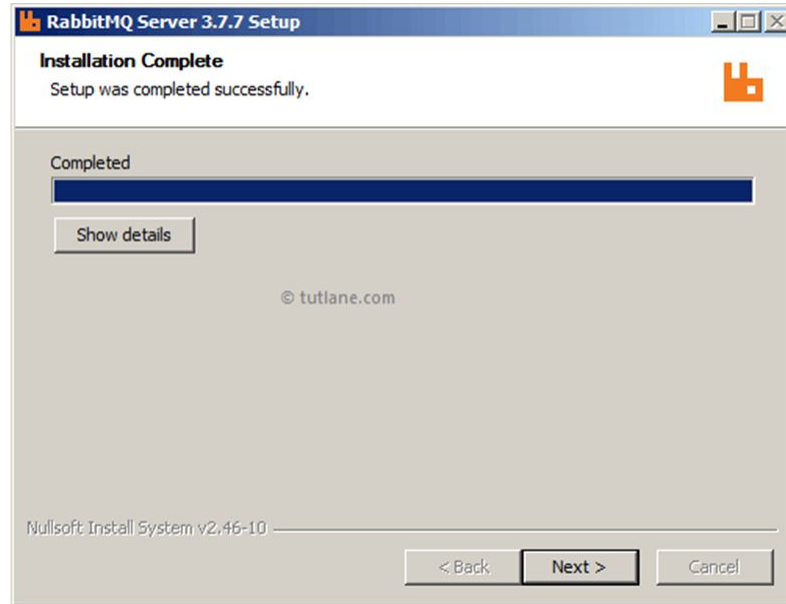
Linux, BSD, UNIX: [Debian](#) | [Ubuntu](#) | [RHEL](#) | [CentOS](#) | [Fedora](#) | [Generic Unix](#) | [Solaris](#)

Windows: [With installer \(recommended\)](#) | [Manual](#)

MacOS: [Homebrew](#) | [Standalone](#) | [Generic Unix](#)

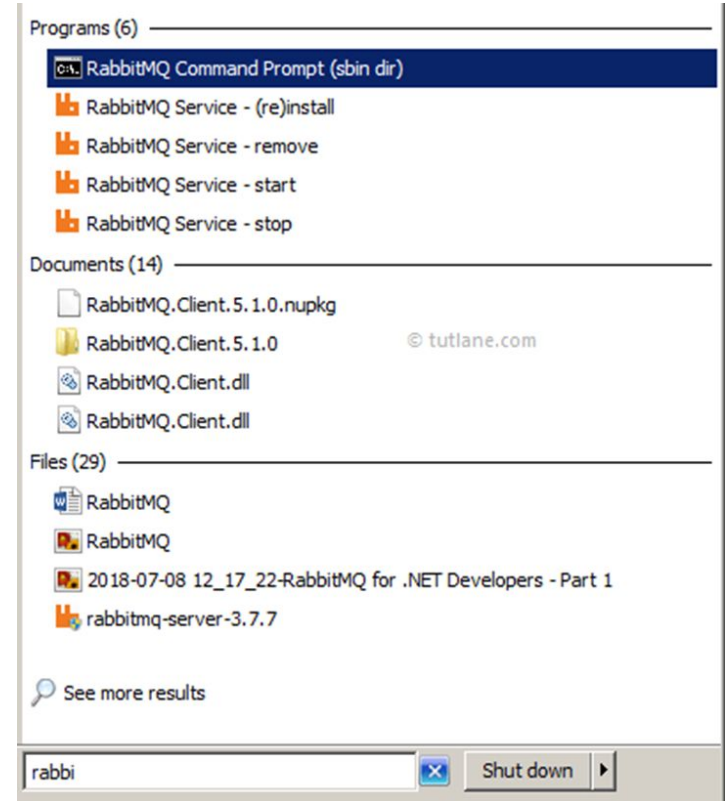
[Erlang/OTP for RabbitMQ](#)

- After downloading the required setup file, just double click on setup file to install rabbitmq on windows like as shown below.



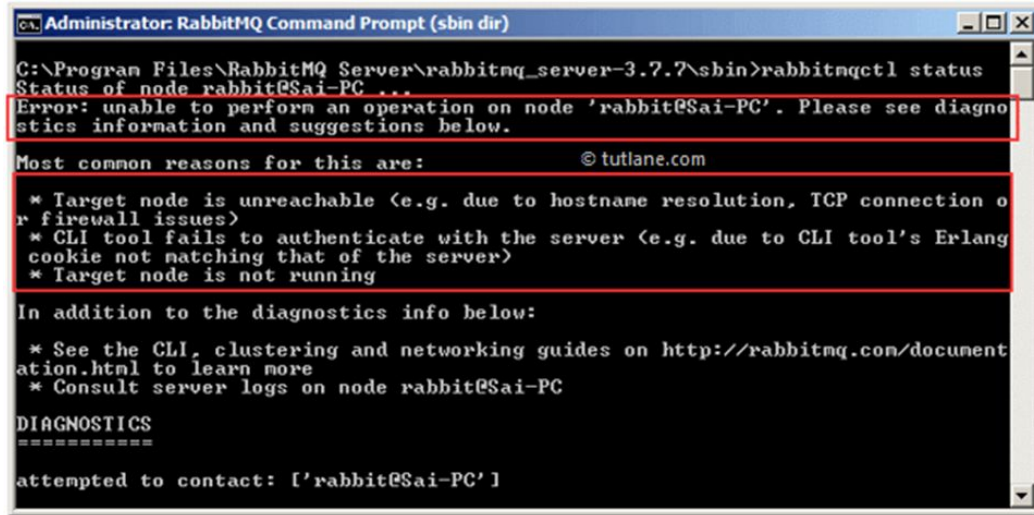


- After completion of installing the RabbitMQ server, we will check the status of RabbitMQ server for that search for **RabbitMQ Command Prompt** and open it with admin privilege like as shown below.



- After opening the command prompt, enter command “**rabbitmqctl status**” and click enter to check the status of RabbitMQ server.

- If it shows an error like as below, then you need to follow the given steps to fix this error.



```
Administrator: RabbitMQ Command Prompt (sbin dir)

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.7\sbin>rabbitmqctl status
Status of node rabbit@Sai-PC ...
Error: unable to perform an operation on node 'rabbit@Sai-PC'. Please see diagnostics information and suggestions below.

Most common reasons for this are: © tutlane.com

* Target node is unreachable (e.g. due to hostname resolution, TCP connection or firewall issues)
* CLI tool fails to authenticate with the server (e.g. due to CLI tool's Erlang cookie not matching that of the server)
* Target node is not running

In addition to the diagnostics info below:

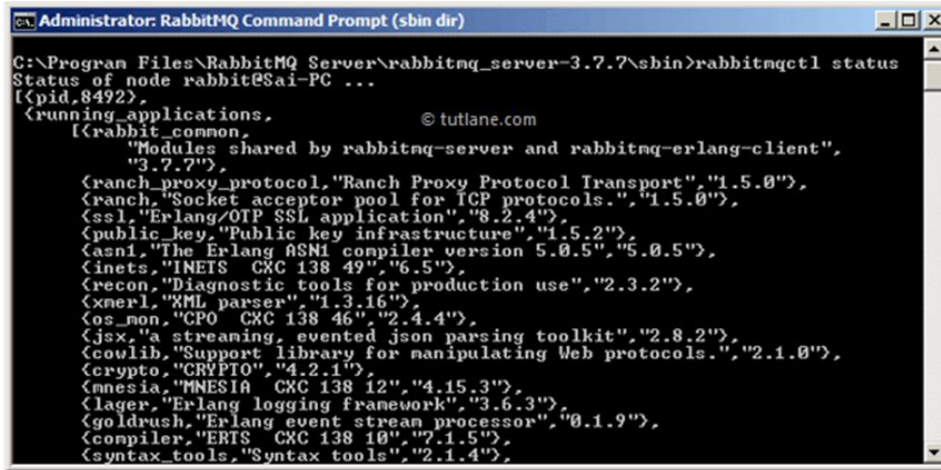
* See the CLI, clustering and networking guides on http://rabbitmq.com/documentation.html to learn more
* Consult server logs on node rabbit@Sai-PC

DIAGNOSTICS
=====
attempted to contact: ['rabbit@Sai-PC']
```

- Following are the steps to fix an error like “**Authentication failed (rejected by the remote node), check the Erlang cookie**”.

1. In file explorer navigate to your user directory by pasting **%userprofile%** in your address bar.
2. In case if already **.erlang.cookie** file available in that location, just delete it otherwise go to the next step.
3. In a second File Explorer, navigate to **C:\Windows\System32\config\systemprofile**.
4. Find the file **.erlang.cookie** and copy it to your user directory.

- After completion of above steps, then again run “**rabbitmqctl status**” command in rabbitmq command prompt as an administrator, then it will shows the screen like as shown below.



```
Administrator: RabbitMQ Command Prompt (sbin dir)
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.7\sbin>rabbitmqctl status
Status of node rabbit@Sai-PC ...
[{pid,8492},
 {running_applications,
  [{rabbit_common,
    "Modules shared by rabbitmq-server and rabbitmq-erlang-client",
    "3.7.7"},
   {ranch_proxy_protocol, "Ranch Proxy Protocol Transport", "1.5.0"},
   {ranch, "Socket acceptor pool for ICP protocols.", "1.5.0"},
   {ssl, "Erlang/OTP SSL application", "8.2.4"},
   {public_key, "Public key infrastructure", "1.5.2"},
   {asn1, "The Erlang ASN1 compiler version 5.0.5", "5.0.5"},
   {inet, "INET: CXC 138 49", "6.5"},
   {recon, "Diagnostic tools for production use", "2.3.2"},
   {xmerl, "XML parser", "1.3.16"},
   {os_mon, "CPO: CXC 138 46", "2.4.4"},
   {jsx, "a streaming, evented json parsing toolkit", "2.8.2"},
   {cowlib, "Support library for manipulating Web protocols.", "2.1.0"},
   {crypto, "CRYPTO", "4.2.1"},
   {mnesia, "MNESIA: CXC 138 12", "4.15.3"},
   {lager, "Erlang logging framework", "3.6.3"},
   {goldrush, "Erlang event stream processor", "0.1.9"},
   {compiler, "ERTS: CXC 138 10", "7.1.5"},
   {syntax_tools, "Syntax tools", "2.1.4"}]}
```

- Now, we are done with starting the RabbitMQ server, next we will enable a web management plugin for RabbitMQ.

- To enable a rabbitmq web management plugin on windows, we need to start **RabbitMQ Command Prompt** with administrator privilege, enter the command “**rabbitmq-plugins enable rabbitmq\_management**” and execute it.
- After executing the above web management command, the web management plugins will be enabled

and it will show the plugins list which are enabled.

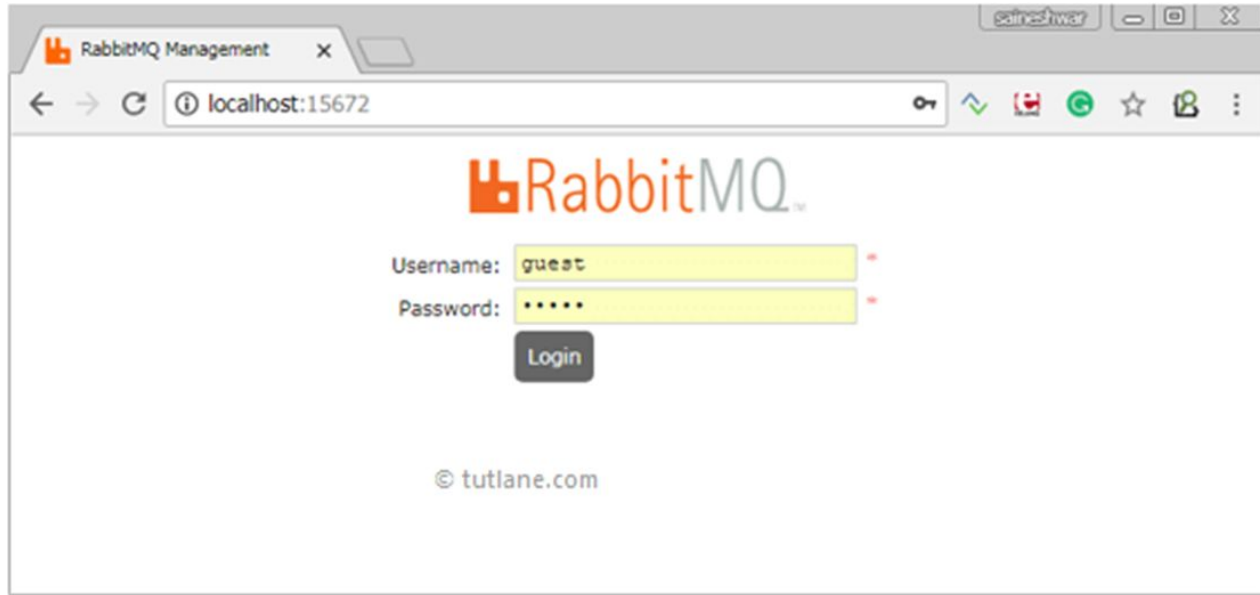
```
The following plugins have been enabled:  
mochiweb  
webmachine  
rabbitmq_web_dispatch      © tutlane.com  
amqp_client  
rabbitmq_management_agent  
rabbitmq_management
```

- Now you can open web management plugin in the browser for that enter the following URL in browser and click enter.

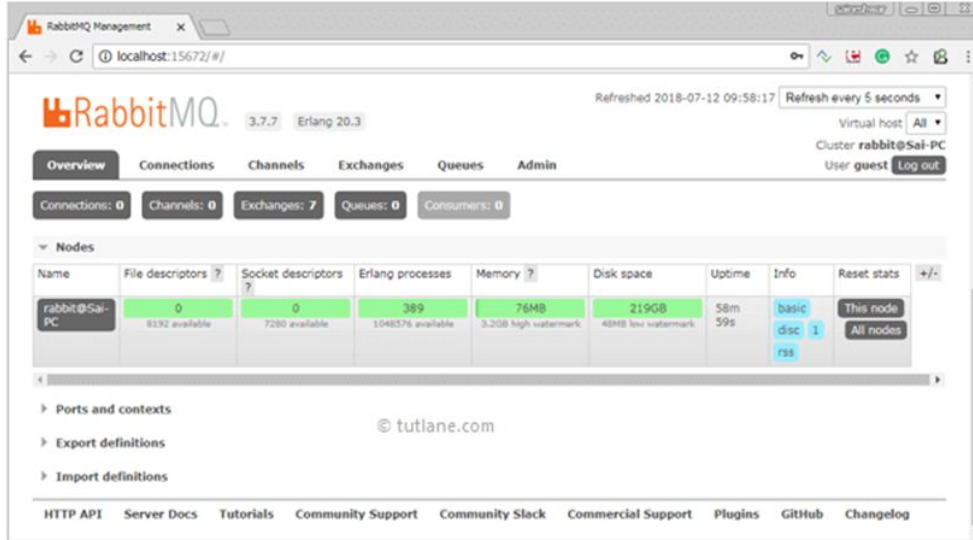
<http://localhost:15672>

- After opening the localhost URL in browser, it will ask you for credentials to access web management plugin.

- To access rabbitmq web management dashboard, the default Username and password of is “**guest**” (Username: “**guest**” | Password: “**guest**”).



- After login with default credentials, the following overview screen will appear.



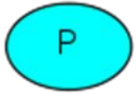
- Now we have successfully started the RabbitMQ server and web management plugin.



- RabbitMQ is a message broker: it accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. or Ms. Mailperson will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.
- The major difference between RabbitMQ and the post office is that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data – *messages*.

RabbitMQ, and messaging in general, uses some jargon.

- *Producing* means nothing more than sending. A program that sends messages is a *producer* :

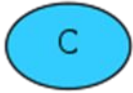


- A *queue* is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a *queue*. A *queue* is only bound by the host's memory & disk limits, it's essentially a large message buffer. Many *producers* can send messages that go to one queue, and many *consumers* can try to receive data from one *queue*.

queue name  
THIS IS HOW we represent a queue:



- *Consuming* has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages:

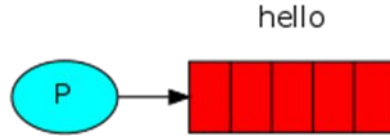


- Note that the producer, consumer, and broker do not have to reside on the same host; indeed in most applications they don't. An application can be both a producer and consumer, too.

- In this part of the tutorial we'll write two programs in Java; a producer that sends a single message, and a consumer that receives messages and prints them out. We'll gloss over some of the detail in the Java API, concentrating on this very simple thing just to get started. It's a "Hello World" of messaging.
- In the diagram below, "P" is our producer and "C" is our consumer. The box in the middle is a queue - a message buffer that RabbitMQ keeps on behalf of the consumer.



- RabbitMQ speaks multiple protocols. This tutorial uses AMQP 0-9-1, which is an open, general-purpose protocol for messaging. There are a number of clients for RabbitMQ in [many different languages](#). We'll use the Java client provided by RabbitMQ.
- Download the [client library](#) and its dependencies ([SLF4J API](#) and [SLF4J Simple](#)). Copy those files in your working directory, along the tutorials Java files.
- Please note SLF4J Simple is enough for tutorials but you should use a full-blown logging library like [Logback](#) in production.
- (The RabbitMQ Java client is also in the central Maven repository, with the `groupId com.rabbitmq` and the artifactId `amqp-client`.)



- We'll call our message publisher (sender) `Send` and our message consumer (receiver) `Recv`.  
The publisher will connect to RabbitMQ, send a single message, then exit.
- In [Send.java](#), we need some classes imported:

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

- Set up the class and name the queue:

```
public class Send {  
    private final static String QUEUE_NAME = "hello";  
    public static void main(String[] argv) throws Exception {  
        ...  
    }  
}
```

- then we can create a connection to the server:

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setHost("localhost");  
try (Connection connection = factory.newConnection();  
    Channel channel = connection.createChannel()) {  
  
}
```



- The connection abstracts the socket connection, and takes care of protocol version negotiation and authentication and so on for us.
- Here we connect to a broker on the local machine - hence the *localhost*. If we wanted to connect to a broker on a different machine we'd simply specify its name or IP address here.
- Next we create a channel, which is where most of the API for getting things done resides. Note we can use a try-with-resources statement because both Connection and Channel implement java.io.Closeable.
- This way we don't need to close them explicitly in our code.

- To send, we must declare a queue for us to send to; then we can publish a message to the queue, all of this in the try-with-resources statement:

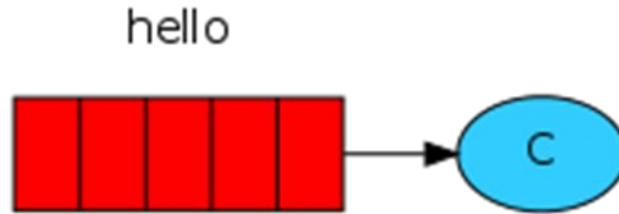
```
channel.queueDeclare(QueueName, false, false, false, null);  
String message = "Hello World!";  
channel.basicPublish("", QueueName, null, message.getBytes());  
System.out.println(" [x] Sent '" + message + "'");
```

- Declaring a queue is idempotent - it will only be created if it doesn't exist already. The message content is a byte array, so you can encode whatever you like there.

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.nio.charset.StandardCharsets;

public class Send {
    private final static String QUEUE_NAME = "hello";
    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
            Channel channel = connection.createChannel()) {
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            String message = "Hello World!";
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes(StandardCharsets.UTF_8));
            System.out.println(" [x] Sent '" + message + "'");
        }
    }
}
```

- That's it for our publisher. Our consumer listening for messages from RabbitMQ, so unlike the publisher which publishes a single message, we'll keep it running to listen for messages and print them out.



- The code (in [Recv.java](#)) has almost the same imports as Send:

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

- The extra DefaultConsumer is a class implementing the Consumer interface we'll use to buffer the messages pushed to us by the server.

- Setting up is the same as the publisher; we open a connection and a channel, and declare the queue from which we're going to consume. Note this matches up with the queue that send publishes to.

```
public class Recv {  
  
    private final static String QUEUE_NAME = "hello";  
  
    public static void main(String[] argv) throws Exception {  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("localhost");  
        Connection connection = factory.newConnection();  
        Channel channel = connection.createChannel();  
  
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");  
  
    }  
}
```

- Note that we declare the queue here, as well. Because we might start the consumer before the publisher, we want to make sure the queue exists before we try to consume messages from it.
- Why don't we use a try-with-resource statement to automatically close the channel and the connection?
- By doing so we would simply make the program move on, close everything, and exit! This would be awkward because we want the process to stay alive while the consumer is listening asynchronously for messages to arrive.

- We're about to tell the server to deliver us the messages from the queue.
- Since it will push us messages asynchronously, we provide a callback in the form of an object that will buffer the messages until we're ready to use them.
- That is what a DeliverCallback subclass does.

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
    String message = new String(delivery.getBody(), "UTF-8");  
    System.out.println(" [x] Received '" + message + "'");  
};  
channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
```



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

public class Recv {

    private final static String QUEUE_NAME = "hello";

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

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

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Received '" + message + "'");
        };
        channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
    }
}
```

- You can compile both of these with just the RabbitMQ java client on the classpath:

```
javac -cp amqp-client-5.7.1.jar Send.java Recv.java
```

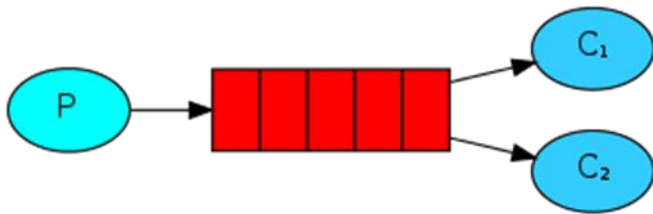
- To run them, you'll need rabbitmq-client.jar and its dependencies on the classpath. In a terminal, run the consumer (receiver):

```
java -cp .:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar Recv
```

- then, run the publisher (sender):

```
java -cp .:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar Send
```

- On Windows, use a semicolon instead of a colon to separate items in the classpath.
- The consumer will print the message it gets from the publisher via RabbitMQ. The consumer will keep running, waiting for messages (Use Ctrl-C to stop it), so try running the publisher from another terminal.



- In the [first tutorial](#) we wrote programs to send and receive messages from a named queue. In this one we'll create a *Work Queue* that will be used to distribute time-consuming tasks among multiple workers.
- The main idea behind Work Queues (aka: *Task Queues*) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a *task* as a message and send it to a queue.
- A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.
- This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

- In the previous part of this tutorial we sent a message containing "Hello World!".
- Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the `Thread.sleep()` function.
- We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by `Hello...` will take three seconds.

- Our old *Recv.java* program also requires some changes: it needs to fake a second of work for every dot in the message body.
- It will handle delivered messages and perform the task, so let's call it *Worker.java*:

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");

    System.out.println(" [x] Received '" + message + "'");
    try {
        doWork(message);
    } finally {
        System.out.println(" [x] Done");
    }
};

boolean autoAck = true; // acknowledgment is covered below
channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback, consumerTag -> { });
```

- Our fake task to simulate execution time:

```
private static void doWork(String task) throws InterruptedException {  
    for (char ch: task.toCharArray()) {  
        if (ch == '.') Thread.sleep(1000);  
    }  
}
```

- Compile them as in tutorial one (with the jar files in the working directory and the environment variable CP):

```
javac -cp $CP NewTask.java Worker.java
```

- One of the advantages of using a Task Queue is the ability to easily parallelise work.
- If we are building up a backlog of work, we can just add more workers and that way, scale easily.
- First, let's try to run two worker instances at the same time.
- They will both get messages from the queue, but how exactly? Let's see.

- You need three consoles open. Two will run the worker program. These consoles will be our two consumers - C1 and C2.

```
# shell 1
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
```

```
# shell 2
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
```



- In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

```
# shell 3
java -cp $CP NewTask First message.
# => [x] Sent 'First message.'
java -cp $CP NewTask Second message..
# => [x] Sent 'Second message..'
java -cp $CP NewTask Third message...
# => [x] Sent 'Third message...'
java -cp $CP NewTask Fourth message....
# => [x] Sent 'Fourth message....'
java -cp $CP NewTask Fifth message.....
# => [x] Sent 'Fifth message.....'
```

- Let's see what is delivered to our workers:

```
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'First message.'
# => [x] Received 'Third message...'
# => [x] Received 'Fifth message.....'
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'Second message..'
# => [x] Received 'Fourth message.....'
```

- By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages.
- This way of distributing messages is called round-robin. Try this out with three or more workers.

- Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done.
- With our current code, once RabbitMQ delivers a message to the consumer it immediately marks it for deletion.
- In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.
- But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.
- In order to make sure a message is never lost, RabbitMQ supports [message acknowledgments](#). An `ack(nowledge)` is sent back by the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it.

- If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it.
- If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.
- There aren't any message timeouts; RabbitMQ will redeliver the message when the consumer dies. It's fine even if processing a message takes a very, very long time.

- Manual message acknowledgments are turned on by default. In previous examples we explicitly turned them off via the `autoAck=true` flag. It's time to set this flag to false and send a proper acknowledgment from the worker, once we're done with a task.

```
channel.basicQos(1); // accept only one unack-ed message at a time (see below)
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received '" + message + "'");
    try {
        doWork(message);
    } finally {
        System.out.println(" [x] Done");
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
    }
};
boolean autoAck = false;
channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback, consumerTag -> { });
```

- Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost.
- Soon after the worker dies all unacknowledged messages will be redelivered.
- Acknowledgement must be sent on the same channel that received the delivery.
- Attempts to acknowledge using a different channel will result in a channel-level protocol exception

- It's a common mistake to miss the `basicAck`. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.
- In order to debug this kind of mistake you can use `rabbitmqctl` to print the `messages_unacknowledged` field:

```
sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

- On Windows, drop the `sudo`:

```
rabbitmqctl.bat list_queues name messages_ready messages_unacknowledged
```

- We have learned how to make sure that even if the consumer dies, the task isn't lost. But our tasks will still be lost if RabbitMQ server stops.
- When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.
- First, we need to make sure that the queue will survive a RabbitMQ node restart. In order to do so, we need to declare it as *durable*:

```
boolean durable = true;  
channel.queueDeclare("hello", durable, false, false, null);
```



- Although this command is correct by itself, it won't work in our present setup. That's because we've already defined a queue called hello which is not durable.
- RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that.
- But there is a quick workaround - let's declare a queue with different name, for example task\_queue:

```
boolean durable = true;  
channel.queueDeclare("task_queue", durable, false, false, null);
```

- This queueDeclare change needs to be applied to both the producer and consumer code.
- At this point we're sure that the task\_queue queue won't be lost even if RabbitMQ restarts.
- Now we need to mark our messages as persistent - by setting MessageProperties (which implements BasicProperties) to the value PERSISTENT\_TEXT\_PLAIN.

```
import com.rabbitmq.client.MessageProperties;

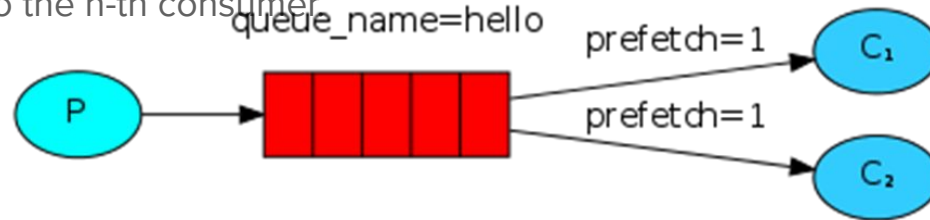
channel.basicPublish("", "task_queue",
    MessageProperties.PERSISTENT_TEXT_PLAIN,
    message.getBytes());
```

- Marking messages as persistent doesn't fully guarantee that a message won't be lost.

Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet.

- Also, RabbitMQ doesn't do `fsync(2)` for every message -- it may be just saved to cache and not really written to the disk.
- The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use [publisher confirms](#).

- You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.
- This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



- In order to defeat that we can use the basicQos method with the prefetchCount = 1 setting. This tells RabbitMQ not to give more than one message to a worker at a time.
- Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

```
int prefetchCount = 1;  
channel.basicQos(prefetchCount);
```

- If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

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

public class NewTask {
    private static final String TASK_QUEUE_NAME = "task_queue";
    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
            Channel channel = connection.createChannel()) {
            channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);
            String message = String.join(" ", argv);
            channel.basicPublish("", TASK_QUEUE_NAME,
                MessageProperties.PERSISTENT_TEXT_PLAIN,
                message.getBytes("UTF-8"));
            System.out.println(" [x] Sent '" + message + "'");
        }
    }
}
```



Worker.java

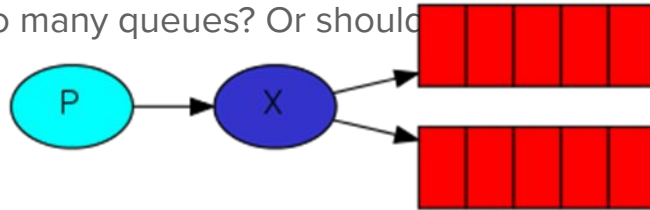
- Using message acknowledgments and prefetchCount you can set up a work queue. The durability options let the tasks survive even if RabbitMQ is restarted.
- For more information on Channel methods and MessageProperties, you can browse the [JavaDocs online](#).
- Now we can move on to [tutorial 3](#) and learn how to deliver the same message to many consumers.



- In the [previous tutorial](#) we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".
- To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs -- the first will emit log messages and the second will receive and print them.
- In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.
- Essentially, published log messages are going to be broadcast to all the receivers.

- In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.
- Let's quickly go over what we covered in the previous tutorials:
  - A *producer* is a user application that sends messages.
  - A *queue* is a buffer that stores messages.
  - A *consumer* is a user application that receives messages.

- The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.
- Instead, the producer can only send messages to an *exchange*. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues.
- The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it be dropped. The rules for that are defined by the *exchange type*.



- There are a few exchange types available: direct, topic, headers and fanout. We'll focus on the last one -- the fanout. Let's create an exchange of this type, and call it logs:

```
channel.exchangeDeclare("logs", "fanout");
```

- The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

## Listing exchanges

- To list the exchanges on the server you can run the ever useful rabbitmqctl:

```
sudo rabbitmqctl list_exchanges
```

- In this list there will be some amq.\* exchanges and the default (unnamed) exchange. These are created by default, but it is unlikely you'll need to use them at the moment.

## Nameless exchange

- In previous parts of the tutorial we knew nothing about exchanges, but still were able to send messages to queues. That was possible because we were using a default exchange, which we identify by the empty string ("").

- Recall how we published a message before:

```
channel.basicPublish("", "hello", null, message.getBytes());
```

- The first parameter is the name of the exchange. The empty string denotes the default or *nameless* exchange: messages are routed to the queue with the name specified by routingKey, if it exists.

- Now, we can publish to our named exchange instead:

```
channel.basicPublish( "logs", "", null, message.getBytes());
```

- As you may remember previously we were using queues that had specific names (remember hello and task\_queue?). Being able to name a queue was crucial for us -- we needed to point the workers to the same queue.
- Giving a queue a name is important when you want to share the queue between producers and consumers.
- But that's not the case for our logger. We want to hear about all log messages, not just a subset of them.
- We're also interested only in currently flowing messages not in the old ones. To solve that we need two things.
  - Firstly, whenever we connect to Rabbit we need a fresh, empty queue. To do this we could create a queue with a random name, or, even better - let the server choose a random queue name for us.
  - Secondly, once we disconnect the consumer the queue should be automatically deleted.



- In the Java client, when we supply no parameters to `queueDeclare()` we create a non-durable, exclusive, autodelete queue with a generated name:

```
String queueName = channel.queueDeclare().getQueue();
```

- At that point `queueName` contains a random queue name. For example it may look like `amq.gen-JzTY20BRgKO-HjmUJj0wLg`.

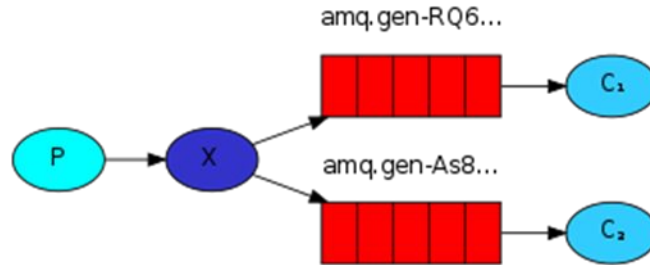
- We've already created a fanout exchange and a queue.
- Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a *binding*.

```
channel.queueBind(queueName, "logs", "");
```

- From now on the logs exchange will append messages to our queue.

## Listing bindings

- You can list existing bindings using, you guessed it,



- The producer program, which emits log messages, doesn't look much different from the previous tutorial.
- The most important change is that we now want to publish messages to our logs exchange instead of the nameless one.
- We need to supply a routingKey when sending, but its value is ignored for fanout exchanges. Here goes the code for EmitLog.java program:

```
public class EmitLog {  
    private static final String EXCHANGE_NAME = "logs";  
    public static void main(String[] argv) throws Exception {  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("localhost");  
        try (Connection connection = factory.newConnection();  
            Channel channel = connection.createChannel()) {  
            channel.exchangeDeclare(EXCHANGE_NAME, "fanout");  
  
            String message = argv.length < 1 ? "info: Hello World!" :  
                String.join(" ", argv);  
            channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));  
            System.out.println(" [x] Sent '" + message + "'");  
        }  
    }  
}
```

- As you see, after establishing the connection we declared the exchange.
- This step is necessary as publishing to a non-existing exchange is forbidden.
- The messages will be lost if no queue is bound to the exchange yet, but that's okay for us;
- if no consumer is listening yet we can safely discard the message.

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

public class ReceiveLogs {
    private static final String EXCHANGE_NAME = "logs";
    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
        String queueName = channel.queueDeclare().getQueue();
        channel.queueBind(queueName, EXCHANGE_NAME, "");
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Received '" + message + "'");
        };
        channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });
    }
}
```

- Compile as before and we're done.

```
javac -cp $CP EmitLog.java ReceiveLogs.java
```

- If you want to save logs to a file, just open a console and type:

```
java -cp $CP ReceiveLogs > logs_from_rabbit.log
```

- If you wish to see the logs on your screen, spawn a new terminal and run:

```
java -cp $CP ReceiveLogs
```

- And of course, to emit logs type:

```
java -cp $CP EmitLog
```



- Using `rabbitmqctl list_bindings` you can verify that the code actually creates bindings and queues as we want.
- With two `ReceiveLogs.java` programs running you should see something like:

```
sudo rabbitmqctl list_bindings
# => Listing bindings ...
# => logs      exchange      amq.gen-JzTY20BRgKO-HjmUJj0wLg  queue      []
# => logs      exchange      amq.gen-vso0PVvyiRIL2WoV3i48Yg  queue      []
# => ...done.
```

- The interpretation of the result is straightforward: data from exchange logs goes to two queues with server-assigned names. And that's exactly what we intended.

# ASSIGNMENT 00 (HOME ASSIGNMENT)



- <https://www.tutlane.com/tutorial/rabbitmq/introduction-to-rabbitmq>

# Thank You

