

# Fondements des Systèmes repartis

## TP1 : Serveur de messagerie pour les applications réparties : RabbitMQ

The first part of this lab is to understand the RabbitMQ messaging distributed systems by running same examples. Then, at the final step, you need to create a distributed application that use communication based on RabbitMQ API.

### 1. Installation

#### Download the Server

Installer for Windows systems (from [GitHub](https://github.com/rabbitmq/rabbitmq-server), recommended)  
**rabbitmq-server-3.7.10.exe** (form:  
<https://www.rabbitmq.com/management.html>)

#### Install the Server

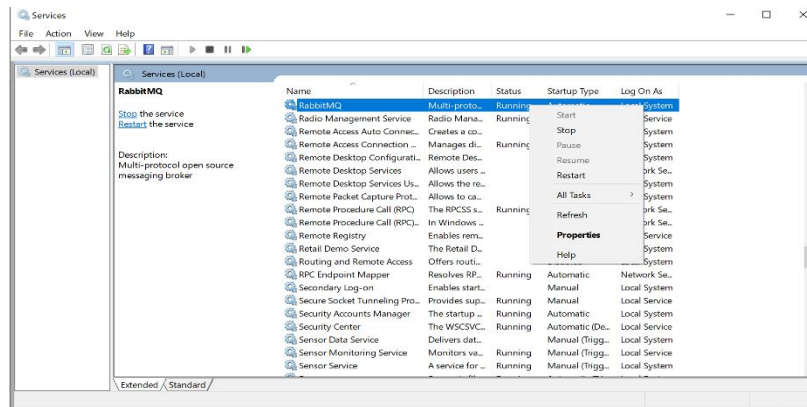
RabbitMQ requires a 64-bit **supported version of Erlang** for Windows to be installed before installing RabbitMQ server.

### 2. Setting web interface for RabbitMQ server

The management plugin is included in the RabbitMQ distribution. Like any other **plugin** it must be enabled before it can be used. That's done using **rabbitmq-plugins**: on the relative path (C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.7.10\sbin) :

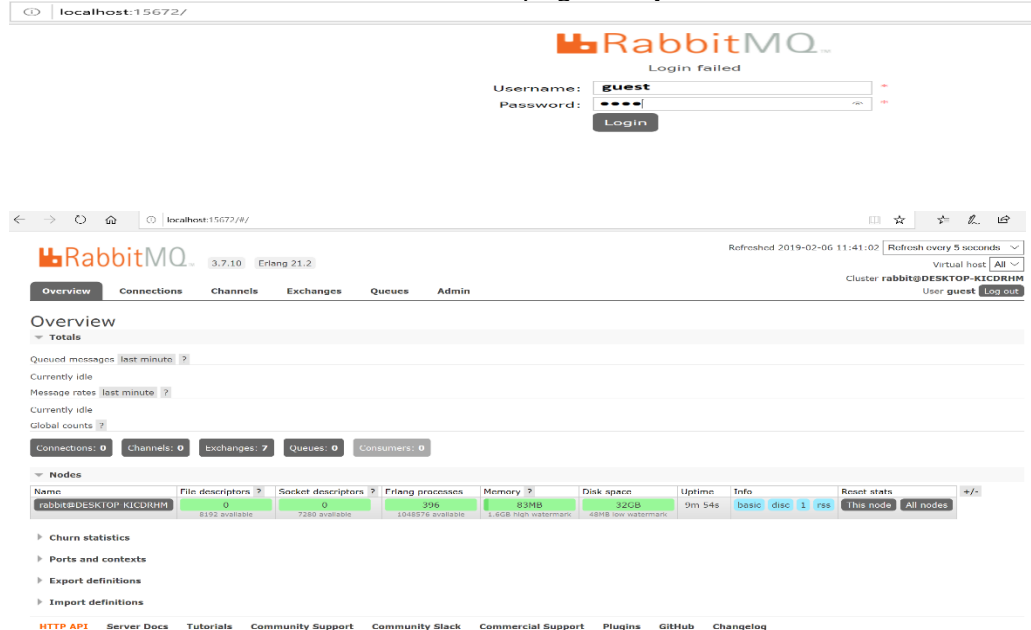
```
rabbitmq-plugins enable rabbitmq_management
```

### Restart associated service

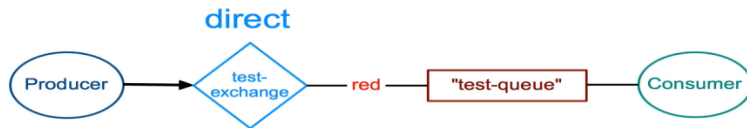


### 3. Use the web interface

Run the web RabbitMQ administration page : **<http://localhost:15672/>**



Use the web administration interface to create manually a Queue, then an exchange, a binding to the new queue as the flowing figure. Make a test by publishing/sending a message and then read/consume this message.



The following figures shows the step to realise that.

Page 1 of 1 - Filter:  ☐ Regex (?)

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
test-queue	D		NaN	NaN	NaN			

**Add a new exchange**

Name:  \*

Type:

Durability:

Add binding from this exchange

To queue:  \*

Routing key:

Arguments:  =  String

**Publish message**

Routing key:

Delivery mode:

Headers: (?)  =  String

Properties: (?)  =

Payload:

**Get messages**

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

Requeue:

Encoding:  (?)

Messages:

## 4. Send Client code: hello message

Write the following code and add libs to run it correctly.

```

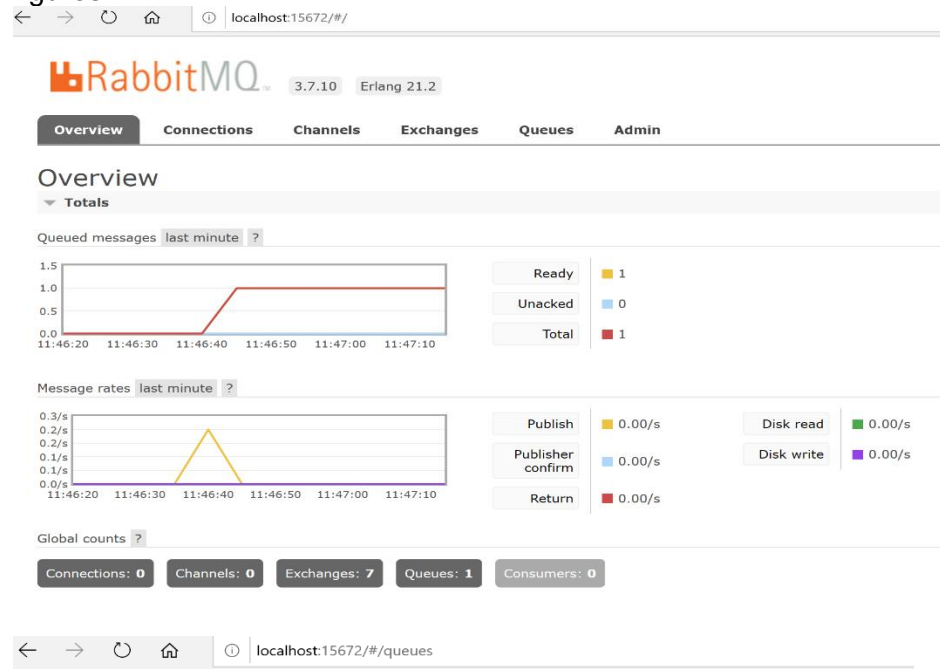
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;

public class send {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] args) 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());
            System.out.println(" [x] Sent '" + message + "'");
        }
    }
}

```

The message is end to the RabbitMQ server, that you can display form web interface. Verify that on your server as in the following figures.



The screenshot shows the RabbitMQ Queues page. At the top, there's a navigation bar with tabs: Overview, Connections, Channels, Exchanges, Queues, and Admin. The Queues tab is selected. Below the navigation bar, there's a section titled 'Queues' with an 'All queues (1)' dropdown. Under 'All queues (1)', there's a 'Pagination' section with a 'Page 1 of 1' and a 'Filter' input field. Below the pagination, there's a table with the following data:

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
hello		idle	1	0	1	0.00/s		

Below the table, there's a link to 'Add a new queue'.

## 5. Receiving message from the RabbitMQ :

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. We use the DeleverCallback interface to receive message.

```
@FunctionalInterface
public interface DeliverCallback {

    /**
     * Called when a <code><b>basic.deliver</b></code> is received for this consume
     *
     * @param consumerTag the <i>consumer tag</i> associated with the consumer
     * @param message the delivered message
     * @throws IOException if the consumer encounters an I/O error while processing
     * the message
     */
    void handle(String consumerTag, Delivery message) throws IOException;
}
```

We also use basicConsume function that start a non-exclusive consumer, with explicit acknowledgement and a server-generated consumerTag. The consumerTag Specifies the identifier for the consumer. It is local to a channel, so two clients can use the same consumer tag).

**String basicConsume(String queue, DeliverCallback deliverCallback, CancelCallback cancelCallback) throws IOException;**

Write the following code for receive/consume the message:

```

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

public class recieve {

    private final static String QUEUE_NAME = "hello";

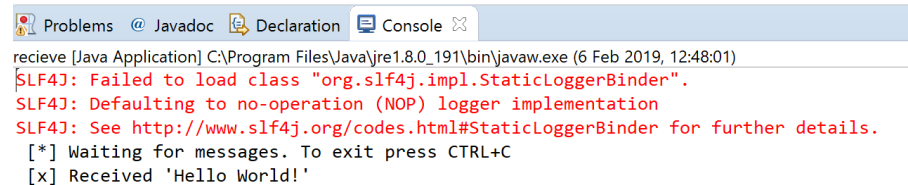
    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare(QUEUE_NAME,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

```

The display result is :



```

Problems @ Javadoc Declaration Console
recieve [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (6 Feb 2019, 12:48:01)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Hello World!'

```

Use the API documentation link to understand some specific codes:

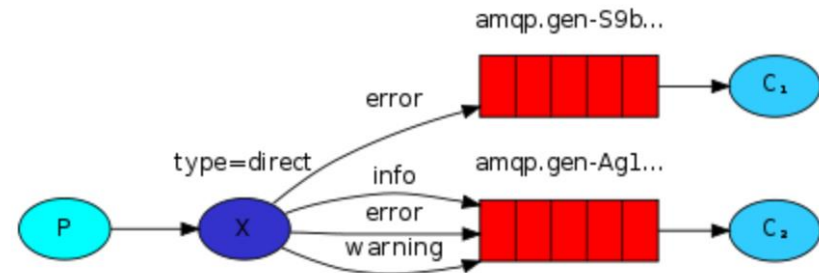
<https://www.javatips.net/api/rabbitmq-java-client-master/src/main/java/com/rabbitmq/client/DeliverCallback.java>

## 6. On using message exchange and routing:

The producer sends messages to an *exchange* (X). 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 routing algorithm behind a **direct** exchange is simple - a message goes to the queues whose **binding key** exactly matches the **routing key** of the message.

In this section, we built a simple logging system. We were able to broadcast log messages to many receivers. We will be able to direct only critical error messages to the log file, while still being able to print all of the log messages on the console.



```

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

public class EmitLogDirect {

    private static final String EXCHANGE_NAME = "direct_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, BuiltinExchangeType.DIRECT);

            String severity = getSeverity(argv);
            String message = getMessage(argv);

            channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes("UTF-8"));
            System.out.println(" [x] Sent '" + severity + "':" + message + "'");
        }
    }
}

```

```

import com.rabbitmq.client.*;

public class ReceiveLogsDirect {
    private static final String EXCHANGE_NAME = "direct_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, BuiltinExchangeType.DIRECT);

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

        if (argv.length < 1) {
            System.err.println("Usage: ReceiveLogsDirect [info] [warning] [error]");
            System.exit(1);
        }

        for (String severity : argv) {
            channel.queueBind(queueName, EXCHANGE_NAME, severity);
        }

        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

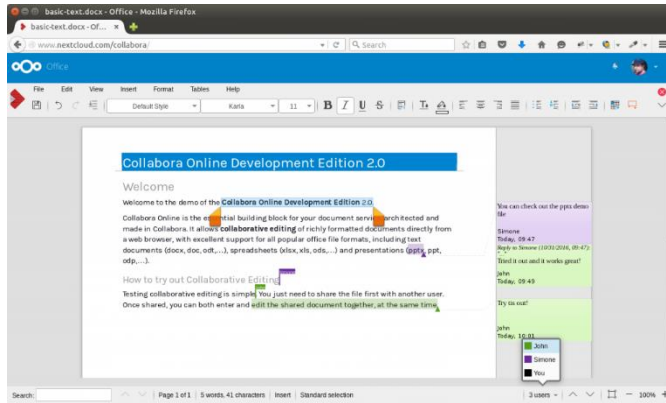
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Received '" + delivery.getEnvelope().getRoutingKey()
                               + "':" + message + "'");
        };

        channel.basicConsume(queueName, true, deliverCallback, consumerTag -> {
        });
    }
}

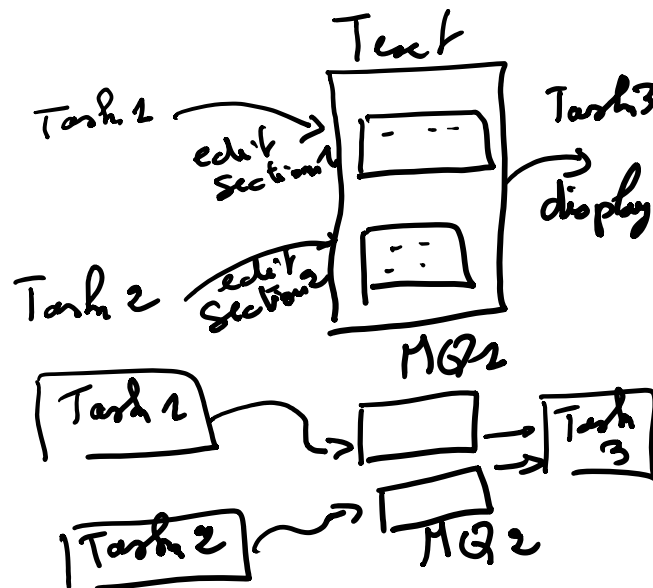
```

## 7- Create a simple of collaborative text editing

There are many collaborative text edition applications provided in Internet. It is one of the relevant examples of distributed applications where users can share and modify at the same time a text to be edited on collaborative way.



1. In this lab, we start by creating a simple application that contains 2 tasks to edit 2 different sections of a text document, and one more task (third) to read and display the text. The 3 tasks are distributed in different computers and the communication will be held with RabbitMQ. This is a start illustration of the application:



2. The second part of this application is to **create exclusion** to access and modify text of the same synthesis or paragraph at the same time from many users. As illustrated in the following, each user has in his interface a set of paragraphs. Only one user can edit at the same time in a paragraph. You need also to display the assigned users to paragraphs if they are writing in. All communications are made with RabbitMQ messaging. Each user has his one program or process.

