

Rabbit-MQ Broker

{{ بسم الله الرحمن الرحيم و الصلاة و السلام على محمد صلى الله عليه و سلم }}



Othman BOUAZZAOUI – Chercheur en IT & Spécialiste Java EE/Angular

Email : oth.bouazzaoui@gmail.com

LinkedIn : [linkedin.com/in/othmanbouazzaoui](https://www.linkedin.com/in/othmanbouazzaoui)

E-mail : oth.bouazzaoui@gmail.com

Mobile : 0690683880

Sommaire

- 1 – Présentation de Rabbit-MQ
- 2 – Message & Binding & Exchanges & Queue
- 3 – Rabbit-MQ avec Spring boot
- 4 - Conclusion

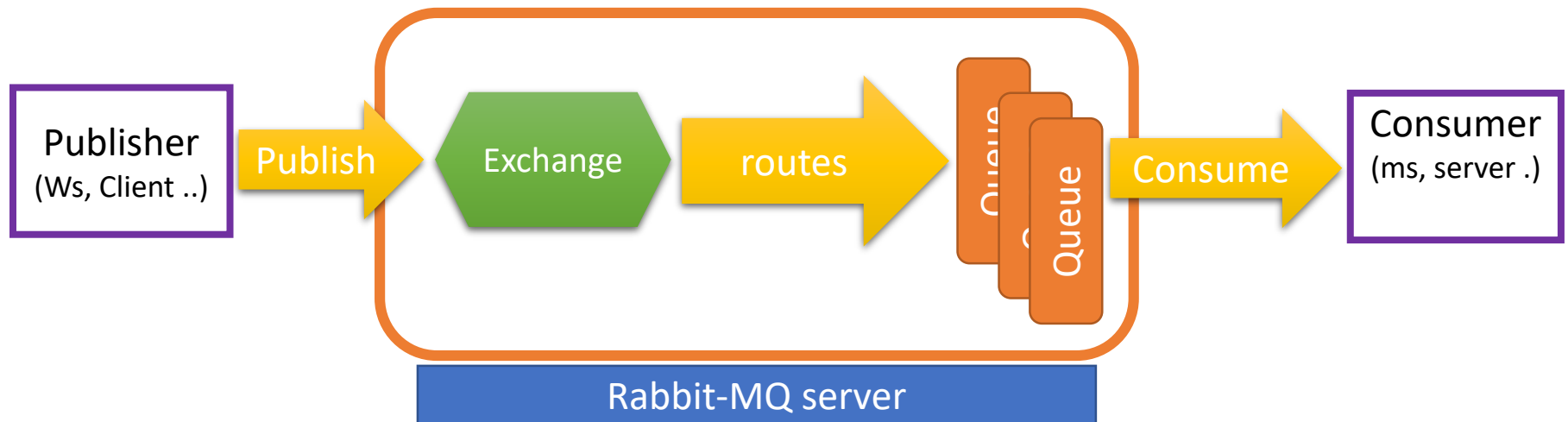
C'est quoi Rabbit-MQ ?

RabbitMQ est un logiciel d'agent de messages open source qui implémente le protocole Advanced Message Queuing, mais aussi avec des plugins Streaming Text Oriented Messaging Protocol et Message Queuing Telemetry Transport.

Le serveur RabbitMQ est écrit dans le langage de programmation Erlang.

Aussi c'est un système de *clustering* pour la haute disponibilité et la scalabilité.

Les vhost(virtuel host) permettent de cloisonner des environnements (mutualiser le serveur, env dev/preprod/prod)



Pourquoi RabbitMQ ?

Puisque rabbitmq permettant la gestion des files de messages afin de permettre à différents clients de communiquer très simplement.

Pour que chaque client puisse communiquer avec RabbitMQ, celui-ci s'appuie sur le protocole AMQP(Advanced Message Queuing).

Ce protocole définit précisément la façon dont vont communiquer les différents clients avec RabbitMQ.

AMQP n'étant qu'un protocole et non une implémentation, chaque client est libre d'implémenter le protocole comme il le souhaite, ou de s'appuyer sur une bibliothèque.

Des bibliothèques existent pour énormément de langages de programmation différents (Java, php, Spring amq ...), ce qui permet de faire communiquer facilement des applicatifs utilisant des technologies très différentes.

Message

Message : Tout simplement c'est comme une requête HTTP mais là on parle du protocole AMQP, qui contient des attributs ainsi qu'un payload. Sauf que parmi les attributs du protocole vous pouvez y ajouter des headers depuis votre publisher(client ou un micro-service).

En plus AMQP plus rapide en terme de transfert des données que http (4 fois),

Ainsi AMQP est asynchrone et http synchrone.

Liste des propriétés du protocole :

content_type, content_encoding, priority, correlation_id, reply_to, expiration, message_id, timestamp, type, user_id, app_id, cluster_id

Les headers seront disponibles dans attributes[headers].

L'attribut routing_key, bien qu'optionnel, n'en est pas moins très utile dans le protocole.

Binding

Les bindings : ce sont les règles que les exchanges utilisent pour déterminer à quelle queue il faut délivrer le message.

Les différentes configurations peuvent utiliser la routing key (direct/topic exchanges) ainsi que les headers(header exchanges) et delayed messages.

Dans le cas des exchanges fanout, les queues n'ont qu'à être bindées pour recevoir le message.

Exchanges

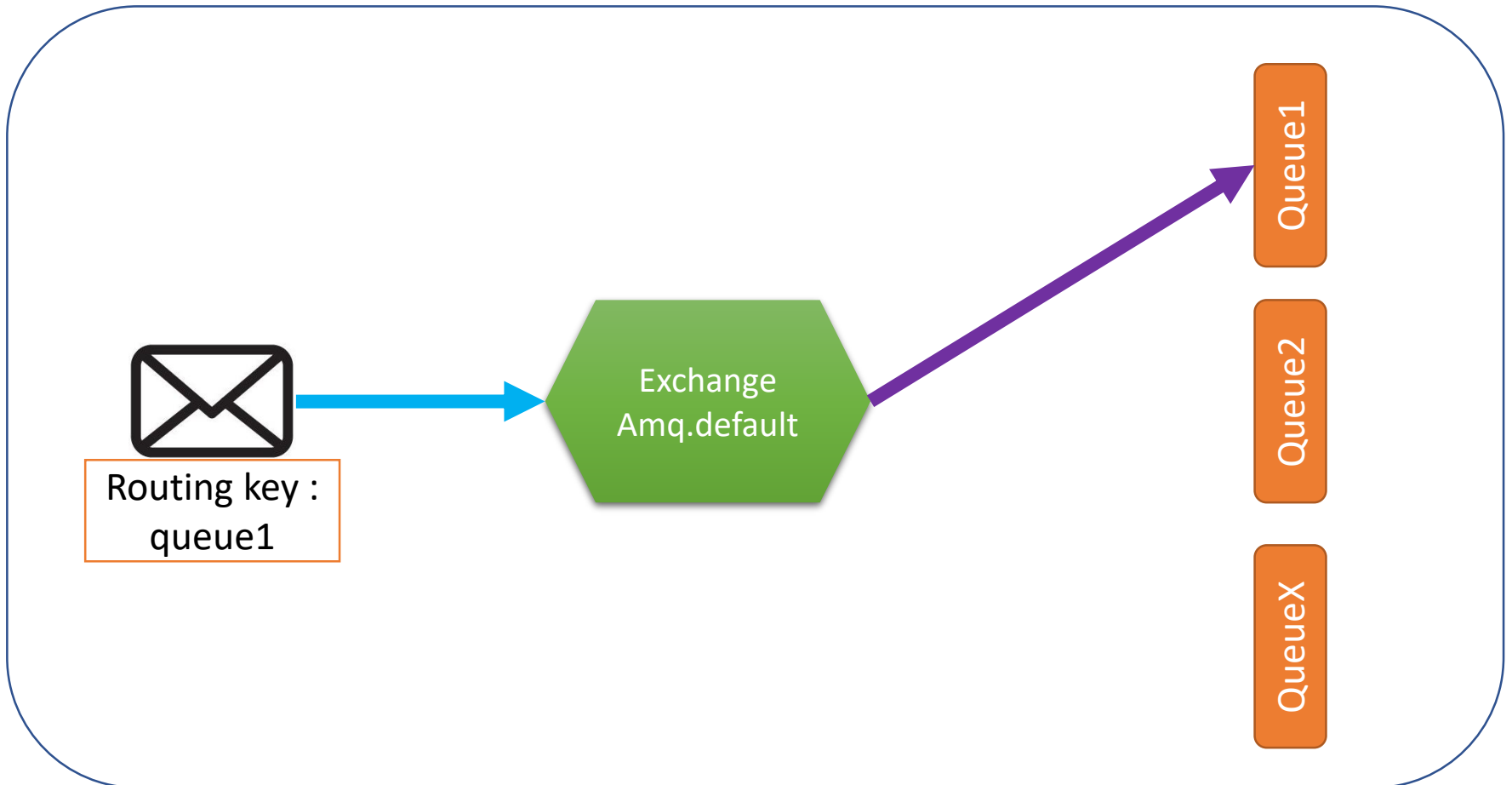
Exchange : ce sont des agents de routage des messages, définis par l'hôte virtuel dans RabbitMQ.

Un échange est responsable du routage des messages vers différentes files d'attente à l'aide d'attributs d'en-tête, de liaisons et de clés de routage.

Une liaison est un "lien" que vous configurez pour lier une file d'attente à un échange.

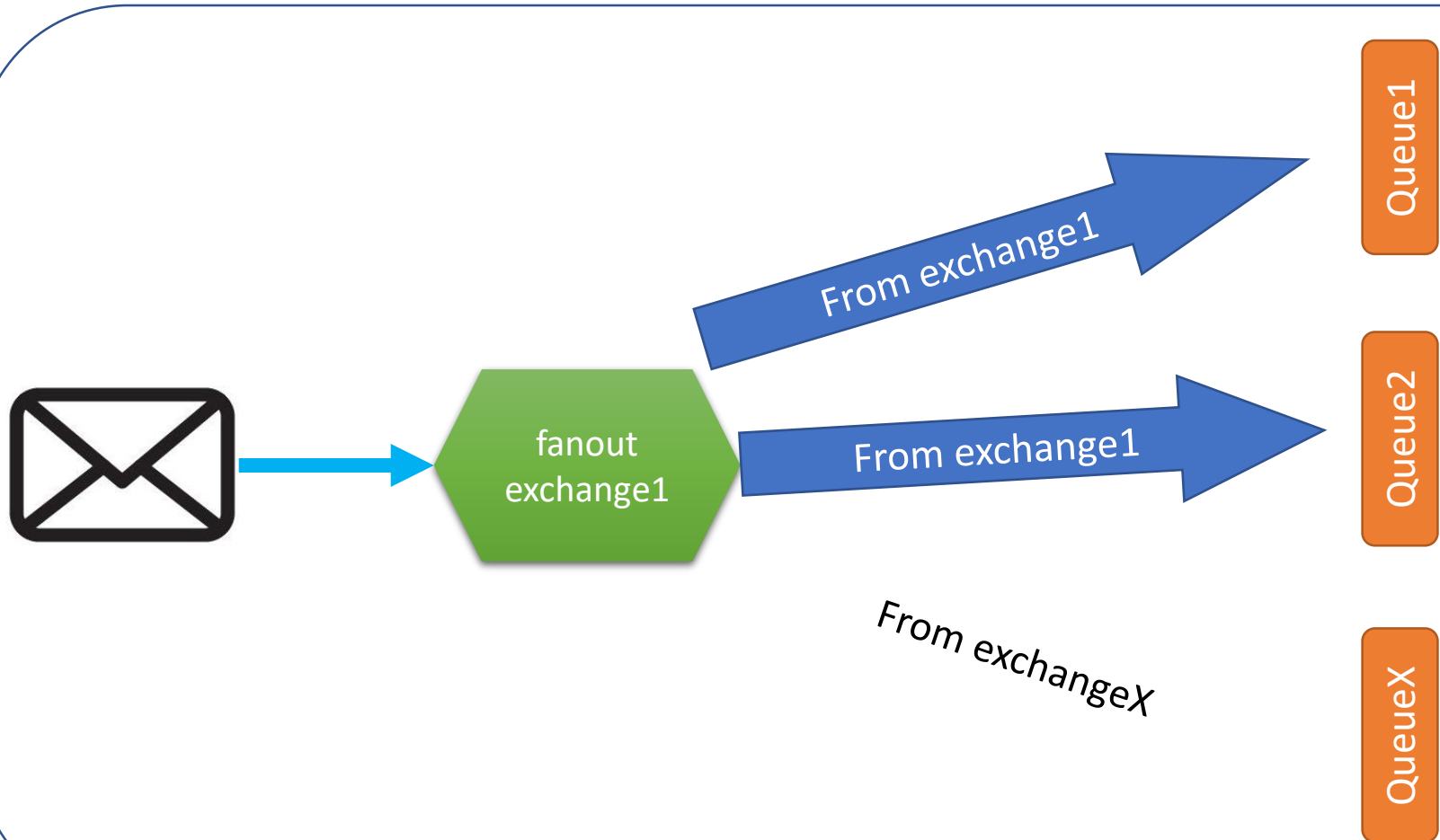
Exchanges Types

Default exchange : auto bindé avec toutes les queues avec une routing key égale au nom de la queue.



L'échange fanout : il envoie le message à toutes les queues bindées.

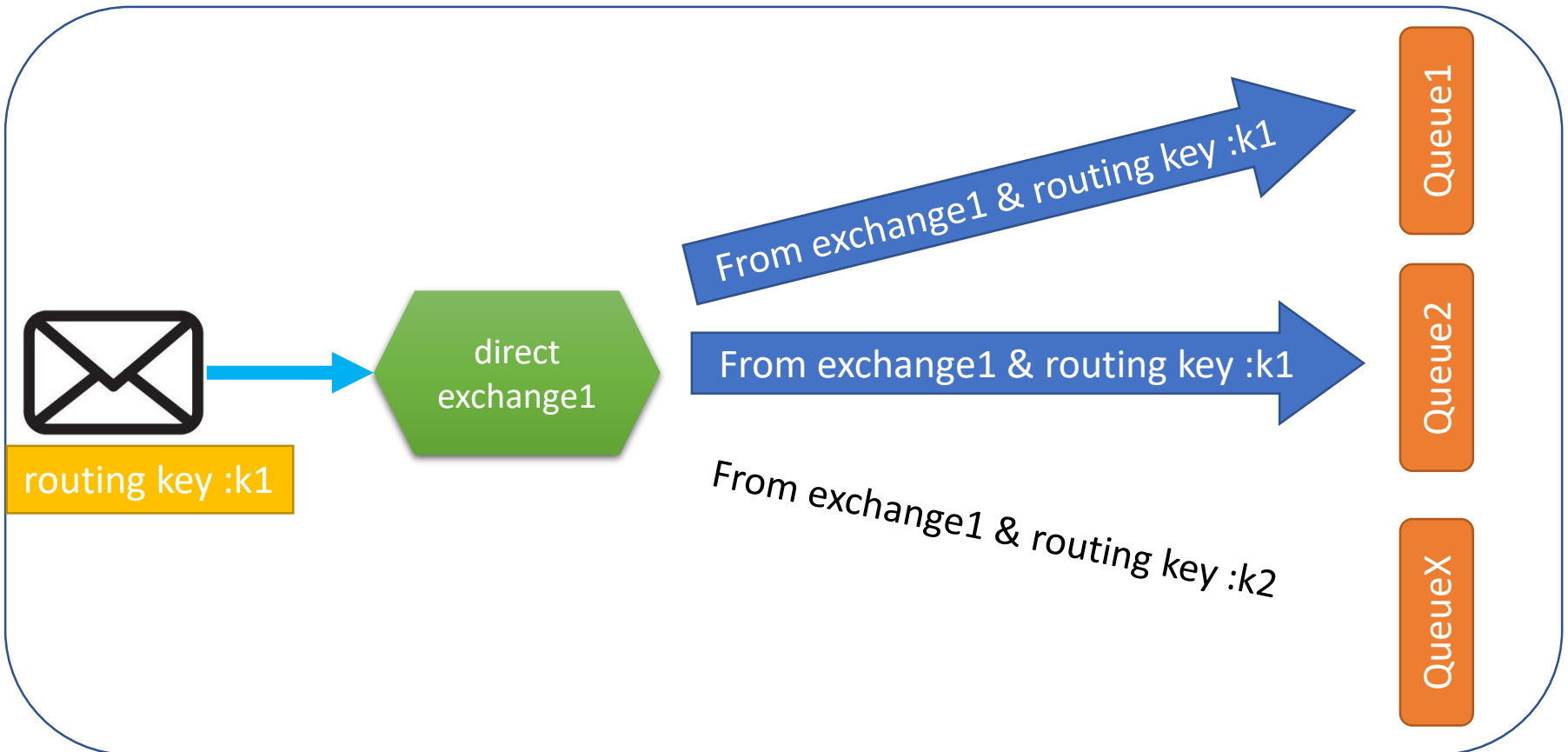
Dans l'exemple : exchange1 va distribuer les messages au queue1 et 2 seulement.



L'exchange direct : n'autorise que le binding utilisant strictement la routing key.

Si la routing_key du message est strictement égale à la routing_key spécifiée dans le binding alors le message sera envoyé à la queue.

```
binding.routing_key == message.routing_key
```



L'exchange Topic : délivre le message si routing_key du message matche le pattern défini dans le binding.

Une routing key est composé de plusieurs segments séparés par des .. Il y a également 2 caractères utilisés dans le matching.

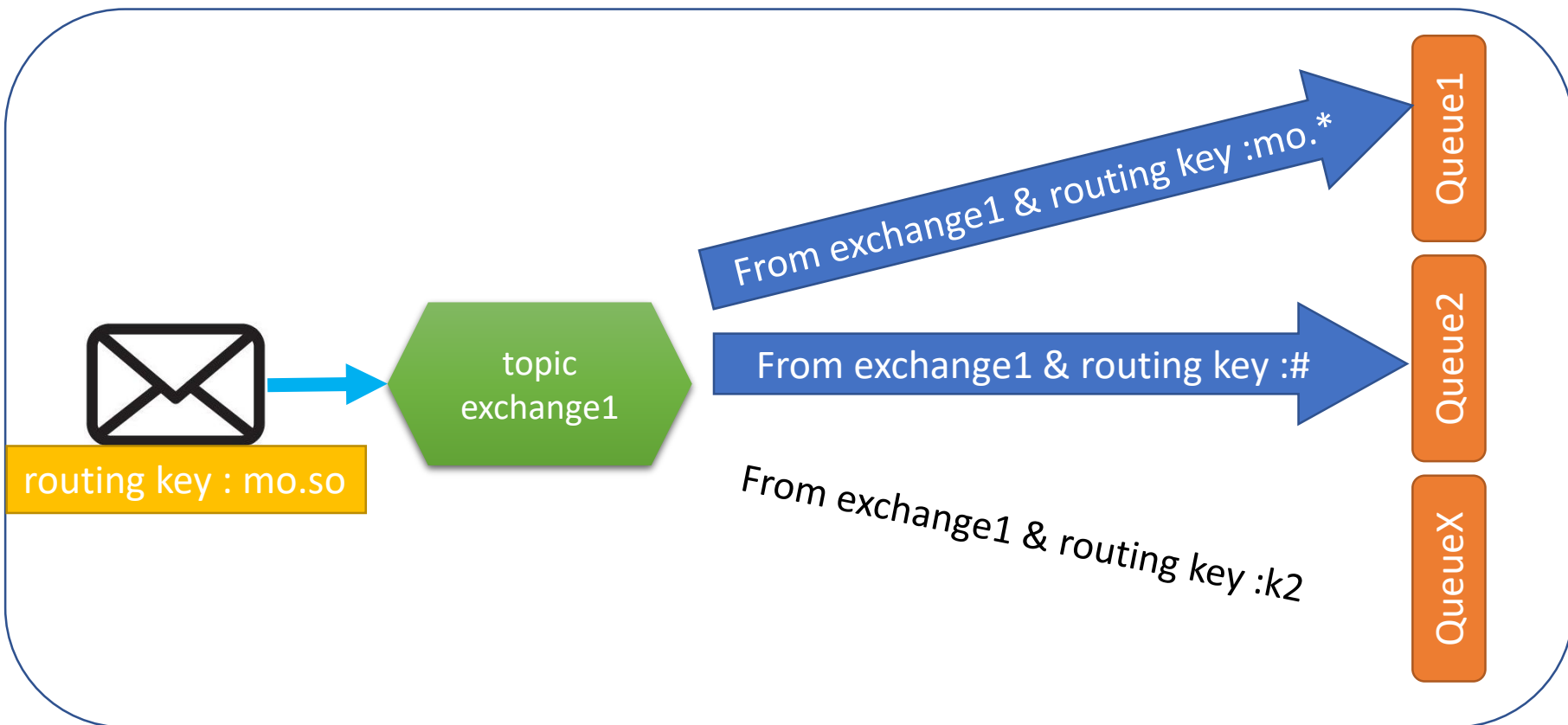
* : n'importe quelle valeur de segment

: n'importe quelle valeur de segment une ou plusieurs fois

Exemple routing key moo.soo.loo :

moo.#.loo => match

moo.# => not match

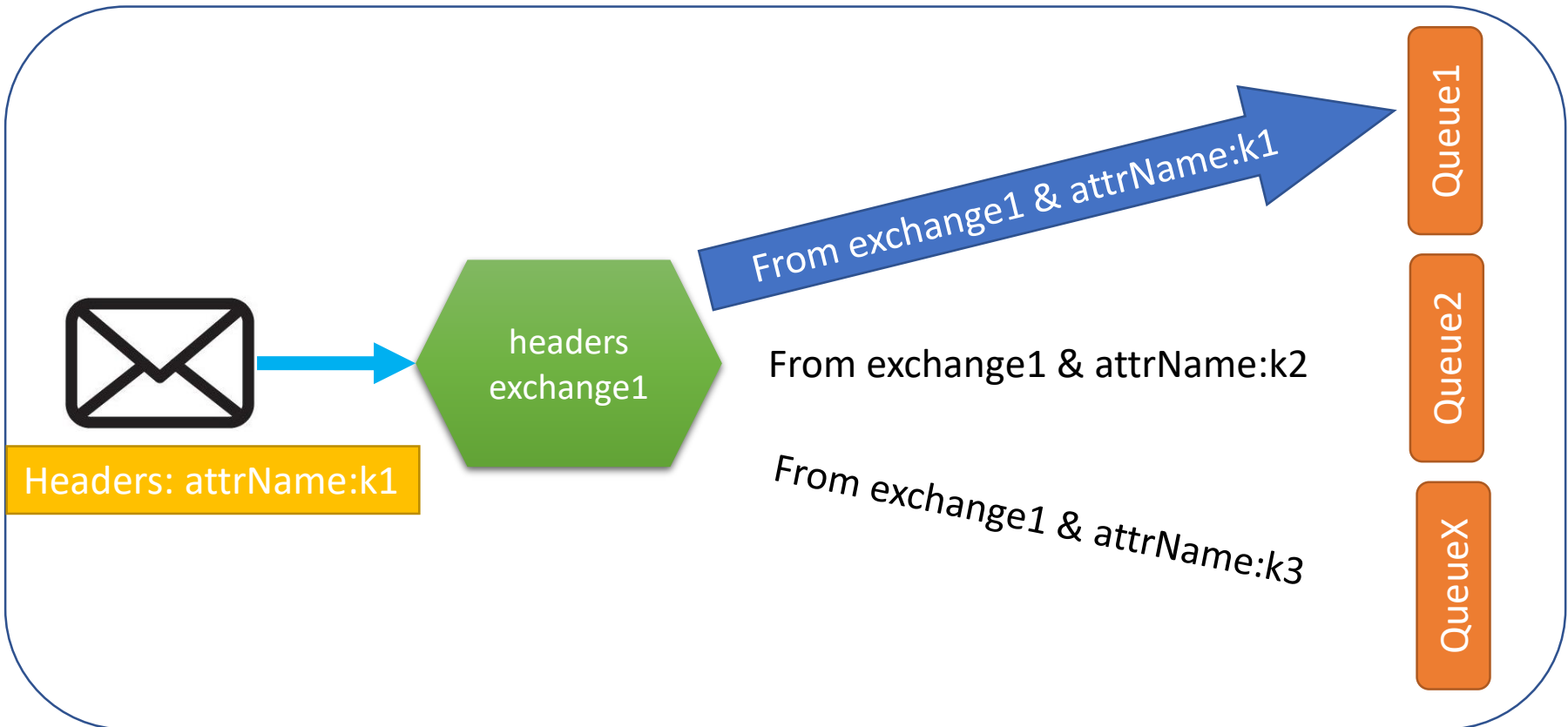


L'exchange Headers : délivre le message si les headers du binding matchent les headers du message.

L'option `x-match` dans le binding permet de définir si un seul header ou tous doivent matcher.

x-match = any le message sera délivré si un seul des headers du binding correspond à un header du message.

x-match = all le message sera délivré si tous les headers du binding correspondent aux headers du message.

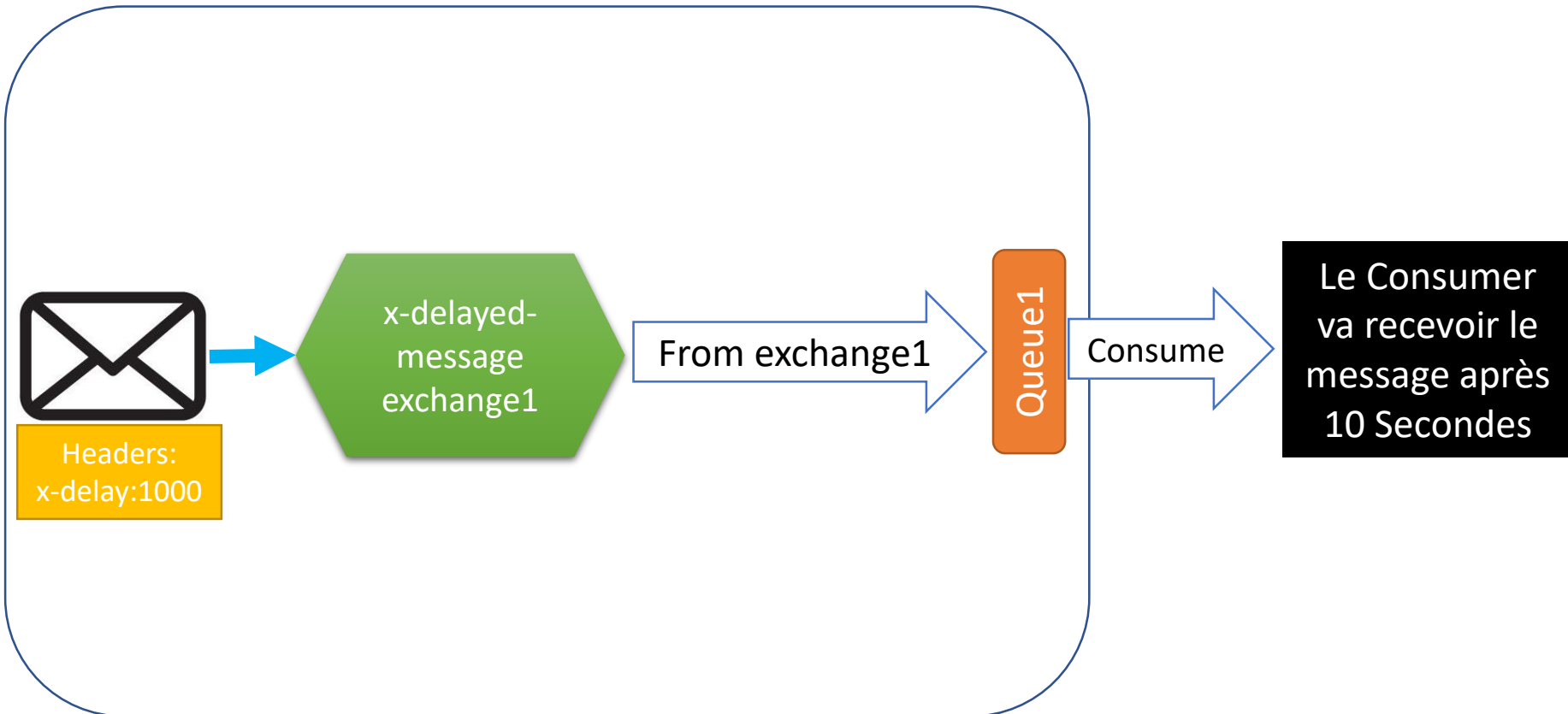


L'échange x-delayed-message : envoie le message avec un planificateur(scheduler).

Par exemple si je veux que le message publié par le Producer soit consommé après 10 secondes, je dois utiliser ce type d'échange.

x-delay : pour indiquer le temps qu'il faut attendre pour consommer ce message le Producer doit envoyer cette propriété dans le header qui prend un entier en milliseconde x-delay=10000 (10 secondes)

Référence : <https://blog.rabbitmq.com/posts/2015/04/scheduling-messages-with-rabbitmq>



Queue

Queue :

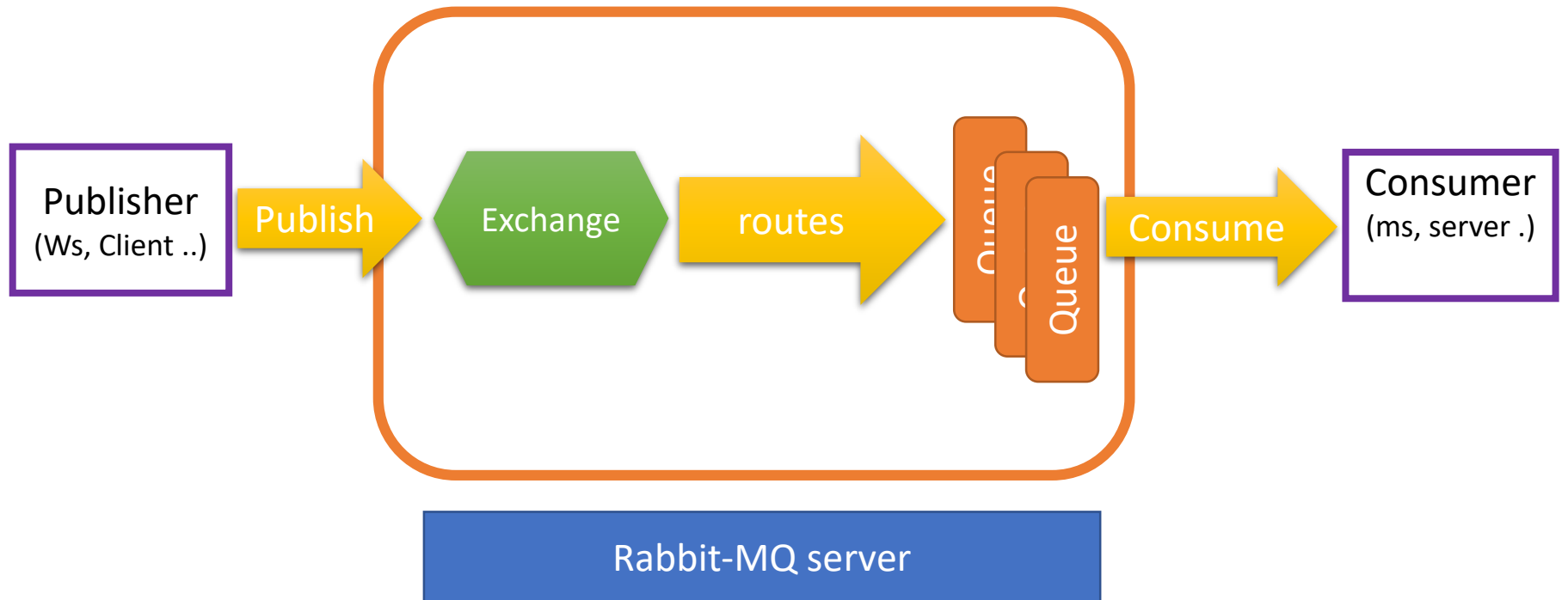
Une queue est l'endroit où sont stockés les messages. Il existe des options de configuration afin de modifier leurs comportements.

Voici quelque types :

- **Durable** : (stockée sur disque) la queue survivra au redémarrage du broker. Attention seuls les messages persistants survivront au redémarrage.
- **Exclusive** : sera utilisable sur une seule connexion et sera supprimée à la clôture de celle-ci.
- **Auto-delete** : la queue sera supprimée quand toutes les connections sont fermées (après au moins une connexion).

Rabbit-MQ avec Spring boot

Nous allons créer une application spring boot qui implémentent cette architecture :



Comment j'installe rabbitMQ ?

Il y a 2 options soit je le télécharge, je l'installe, comme n'importe quel autre logiciel ou j'utilise la version docker.

1 – l'installation manuel :

Ouvrir ce lien et télécharger rabbitMQ :

<https://www.rabbitmq.com/install-windows.html#downloads>

Puis, vous l'installez de la manière classique

Après vous vous positionnez sur le dossier de l'installation :

C:\Program Files\RabbitMQ Server\rabbitmq_server-xxx\sbin

Vérifier si vous avez le même chemin 😊

Exécuter ces 2 commandes(en tant qu'admin)

- **rabbitmq-plugins enable rabbitmq_management --offline**

- ***rabbitmq-server.bat start***

La première pour accéder depuis votre navigateur et l'autre pour démarrer rabbit-srv

Alors vous pouvez maintenant accéder à rabbit-mq server sur l'adresse :

<http://localhost:15672/>

2 – l'installation avec l'image docker ou docker compose et autres :

Cette page github vous explique comment le faire avec le projet Spring boot

<https://github.com/othman-bouazzaoui/rabbitmq>

Vous restez sur la branche master, pour avoir le même code, sinon il y a des évolutions sur les autres branches .






La création d'un projet Maven avec Spring boot

Vous pouvez utiliser STS avec éclipse.

Ou avec IntelliJ Idea

Oui avec <https://start.spring.io/>



Project

☒ Maven Project

☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT) ☐ 3.0.0 (M5) ☐ 2.7.5 (SNAPSHOT) ☒ 2.7.4

☐ 2.6.13 (SNAPSHOT) ☐ 2.6.12

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 19 ☒ 17 ☐ 11 ☐ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring for RabbitMQ MESSAGING

Gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.



Lombok DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...



Puis, cliquer sur GENERATE et ouvrir le projet sur un IDE comme IntelliJ ou Eclipse

La structure de notre projet :

The screenshot displays an IDE interface with the following components:

- Project Structure (Left Panel):** Shows the project hierarchy. The `src/main/java` directory is expanded, revealing the package `com.oth.rabbitmq`. Inside this package, there are sub-packages `config`, `consumer`, `entity`, and `publisher`, each containing a class file (e.g., `MessagingConfiguration`, `EmployeeConsumer`, `EmployeePublisher`). The `RabbitmqApplication` class is also visible. The `resources` directory contains `static` and `templates` folders, and an `application.properties` file. The `test` directory is also shown.
- pom.xml File (Right Panel):** The `pom.xml` file is open, showing the following XML structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.oth</groupId>
  <artifactId>rabbitmq</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>rabbitmq</name>
  <description>Demo project for Spring Boot with rabbit-MQ</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.amqp</groupId>
      <artifactId>spring-rabbit-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```
- Bottom Panel:** Shows the status bar with the message "Build completed successfully in 2 sec, 791 ms (19 minutes ago)".

Pour les dépendances, ici j'ai gardé aussi celles pour effectuer des test, mais vous pouvez utiliser que ces 3 :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

Créer une classe Util dans le package : com.oth.rabbitmq.config

Util

```
package com.oth.rabbitmq.config;

import java.util.Arrays;
import java.util.List;

public class Util {

    public static final List<String> QUEUES = Arrays.asList("QUEUE1", "QUEUE2", "QUEUE3", "QUEUE4", "QUEUE5");
    public static final List<String> EXCHANGES = Arrays.asList("TOPIC", "DELAYED_MESSAGE");
    public static final List<String> ROUTING_KEYS = Arrays.asList("key1", "key2");
}
```

Créer une classe de configuration dans le package : com.oth.rabbitmq.config

MessagingConfiguration

Part 1

```
package com.oth.rabbitmq.config;
import org.springframework.amqp.core.*;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class MessagingConfiguration {
    // QUEUES
    @Bean
    public Queue queue1(){
        return new Queue(Util.QUEUES.get(0));
    }
    @Bean
    public Queue queue2(){
        return new Queue(Util.QUEUES.get(1));
    }
    // -- topic exchange exemple
    @Bean
    public TopicExchange topicExchange(){
        return new TopicExchange(Util.EXCHANGES.get(0));
    }
    // -- delayed message exchange
    @Bean
    CustomExchange delayedExchange(){
        Map<String, Object> args = new HashMap<>();
        args.put("x-delayed-type", "direct");
        return new CustomExchange(Util.EXCHANGES.get(1), "x-delayed-message",true, false,args);
    }
}
```

Part 2

```
//Binding 2 : between queue1 and x-delayed-message
@Bean
public Binding bindingQueue2AndDelayedMessageExchange(){
    return
        BindingBuilder.bind(queue2()).to(delayedExchange()).with(Util.ROUTING_KEYS.get(1)).noargs();
}

@Bean
public MessageConverter messageConverter(){
    return new Jackson2JsonMessageConverter();
}

@Bean
public AmqpTemplate template(ConnectionFactory
    connectionFactory){
    final RabbitTemplate rabbitTemplate = new
        RabbitTemplate(connectionFactory);
    rabbitTemplate.setMessageConverter(messageConverter());
    return rabbitTemplate;
}
}
```

Créer 2 classes des entités dans le package : com.oth.rabbitmq.entity

Employee et EmployeeStatus

Employee

```
package com.oth.rabbitmq.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

import java.math.BigDecimal;

@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Employee {

    private String id;
    private String firstName;
    private String lastName;
    private BigDecimal salary;
}
```

EmployeeStatus

```
package com.oth.rabbitmq.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class EmployeeStatus {

    private Employee employee;
    private String status;
    private String message;
}
```

Créer une classe du publisher (Producer) dans le package : com.oth.rabbitmq.publisher

EmployeePublisher

Part 1

```
package com.oth.rabbitmq.publisher;
import com.oth.rabbitmq.config.Util;
import com.oth.rabbitmq.entity.Employee;
import com.oth.rabbitmq.entity.EmployeeStatus;
import org.springframework.amqp.AmqpException;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessagePostProcessor;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.UUID;

@RestController
@RequestMapping("employee")
public class EmployeePublisher {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @PostMapping("publish")
    public String publishEmployeeTopicExchange(@RequestBody Employee employee){
        employee.setId(UUID.randomUUID().toString());
        EmployeeStatus employeeStatus = new EmployeeStatus(employee, "STARTED", "start process");
        //transfert the employee to service layer ...
        employeeStatus.setStatus("PROGRESS");
        employeeStatus.setMessage("process placed successfully");
        rabbitTemplate.convertAndSend(Util.EXCHANGES.get(0), Util.ROUTING_KEYS.get(0), employeeStatus);

        return "success with topicExchange!!!";
    }
}
```

Part 2

```
@PostMapping("publish2")
public String publishEmployeeDelayedMessage(@RequestBody Employee employee){
    employee.setId(UUID.randomUUID().toString());
    EmployeeStatus employeeStatus = new EmployeeStatus(employee, "STARTED", "start process");
    //transfert the employee to service layer ...
    employeeStatus.setStatus("PROGRESS");
    employeeStatus.setMessage("process placed successfully");

    rabbitTemplate.convertAndSend(Util.EXCHANGES.get(1), Util.ROUTING_KEYS.get(1), employeeStatus, new MessagePostProcessor() {
        @Override
        public Message postProcessMessage(Message message) throws AmqpException {
            System.out.println(message.getMessageProperties());
            message.getMessageProperties().setHeader("x-delay", 10000);
            return message;
        }
    });
    return "success with DelayedMessageExchange!!!";
}
```


Créer une classe du Consumer dans le package : com.oth.rabbitmq.consumer

EmployeeConsumer

EmployeeConsumer

```
package com.oth.rabbitmq.consumer;

import com.oth.rabbitmq.entity.EmployeeStatus;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class EmployeeConsumer {

    @RabbitListener(queues = "QUEUE1")
    public void consumerMessageFromQueueWithTopicExchange(EmployeeStatus employeeStatus){
        System.out.println("Message received from queue1 With TopicExchange : " + employeeStatus);
    }

    @RabbitListener(queues = "QUEUE2")
    public void consumerMessageFromQueueWithDelayedMessageExchange(EmployeeStatus employeeStatus){
        System.out.println("Message received from queue2 With DelayedMessageExchange : " + employeeStatus);
    }

}
```

Maintenant, vous pouvez tester les 2 en démarrant notre projet :

- 1 - <http://localhost:8080/employee/publish> cela va permet de tester exchange de type topic.
- 2 - <http://localhost:8080/employee/publish2> cela va permet de tester exchange de type x-delayed-message.

Conclusion :

RabbitMQ est très utile si vous travaillez dans une architecture distribuée (micro-service), cela vous aidera à activer la communication asynchrone.

J'ai mis ce projet à votre disposition sur github si souhaitez le cloner :

<https://github.com/othman-bouazzaoui/rabbitmq>

Bon courage 😊

و لا تنسونا من صالح الدعاء

وفقكم الله 😊

good luck 😊

Othman BOUAZZAOUI – Chercheur en IT & Spécialiste Java EE/Angular

Email : oth.bouazzaoui@gmail.com

LinkedIn : [linkedin.com/in/othmanbouazzaoui](https://www.linkedin.com/in/othmanbouazzaoui)

E-mail : oth.bouazzaoui@gmail.com

Mobile : 0690683880