

## Contents

Prerequisite .....	2
Java Client Library – Pub & Sub - 60 Minutes .....	3
Using Spring-Boot(RabbitMQ) -60 Minutes .....	14
Using JSON Message – 60 Minutes .....	30
Pom.xml .....	42

## Prerequisite

All development will be using eclipse IDE. Install Eclipse.

Create a maven Project with the following details (java Quickstart project archetype):

File --> New --> Other --> Maven --> Maven Project --> Select Create a simple project

### New Maven project

Select project name and location



☒ Create a simple project (skip archetype selection)

☒ Use default Workspace location

Location:

Browse...

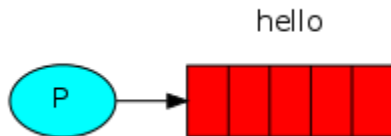
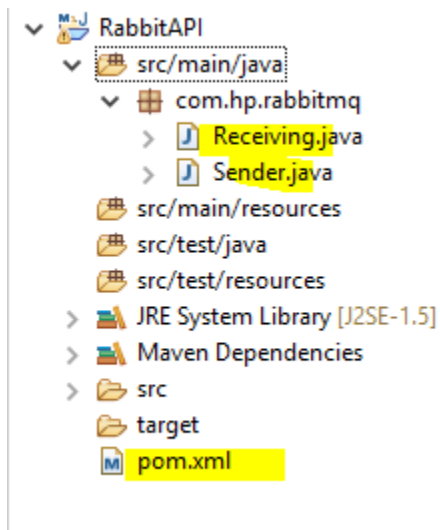
Next. In the next page, you can enter the project details like group Id, Arctifact ID and Version which are mandatory information for creating a maven project. Next you will define library required for running the project in pom.xml.

Update the **pom.xml** of your project with that of the provided given below in pom.xml section

## Java Client Library – Pub & Sub - 60 Minutes

In this lab we will define a java project that will create a direct Queue, hello and publish message to it using Java API.

At the end of this lab, we will have the following project structure



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.

Start the eclipse and enter the following GAV details.

```
<groupId>com.hp.rabbitmq</groupId>
<artifactId>RabbitAPI</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

Update **pom.xml** with the content sepecified in the annexure – pom.xml.

First create a package to store the entire program: com.hp.rabbitmq

Create a java program (Sender.java) as shown below. It will connect to a rabbitMQ host i.e nodeo. Then the program will create a Queue, **hello** and then a message will be published to it. Using the webconsole, you can view the message in the queue.

```
package com.hp.rabbitmq;
```

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

```
public class Sender {
```

```
    private final static String QUEUE_NAME = "hello";
    private final static String QUEUE_HOST = "localhost";
```

```
    /*
```

```
    * You need to substitute the port number of amqp protocol or
```

\* if you are using docker, then the port mapping of the host to the  
 \* above port.  
 \*/

```
private final static int PORT = 17674;
public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(QUEUE_HOST);
    factory.setPort(PORT);
    factory.setUsername("guest");
    factory.setPassword("guest");
    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("UTF-8"));
    System.out.println(" [x] Sent '" + message + "'");

    channel.close();
    connection.close();
}
}
```

## Start the Rabbit MQ i.e Node o

You need to verify amqp port number using the Management UI. Update the port of **amqp** protocol in the above java file. If you are using docker, don't forget to substitute with that of the port mapping of the host. For example, in my case; amqp port 15674 is forward through 17674 of the host.

The screenshot displays the RabbitMQ Management UI. At the top, there are tabs for Overview, Connections, Channels, Exchanges, Queues, and Admin. The 'Overview' tab is selected. Below the tabs, there's a 'Global counts' section with buttons for Connections: 0, Channels: 0, Exchanges: 10, Queues: 6, and Consumers: 0. The 'Nodes' section shows a table with one node, 'rabbit@rabbitmq0', with various statistics like file descriptors, socket descriptors, Erlang processes, memory, and disk space. Below this, there's a 'Churn statistics' section. The 'Ports and contexts' section shows a table of listening ports, where the 'amqp' protocol is bound to port 15674. A red circle highlights the port number 15674. Below this, there's a 'Web contexts' section showing a table with one context, 'RabbitMQ Management', bound to port 15672.

Overview Connections Channels Exchanges Queues Admin User **guest** Log out

Global counts ?

Connections: 0 Channels: 0 Exchanges: 10 Queues: 6 Consumers: 0

▼ Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+/-
rabbit@rabbitmq0	46 32768 available	0 29401 available	413 1048576 available	303 MiB 7.5 GiB high watermark	60 GiB 48 MiB low watermark	16m 11s	basic disc 3 rss	This node All nodes	

► Churn statistics

▼ Ports and contexts

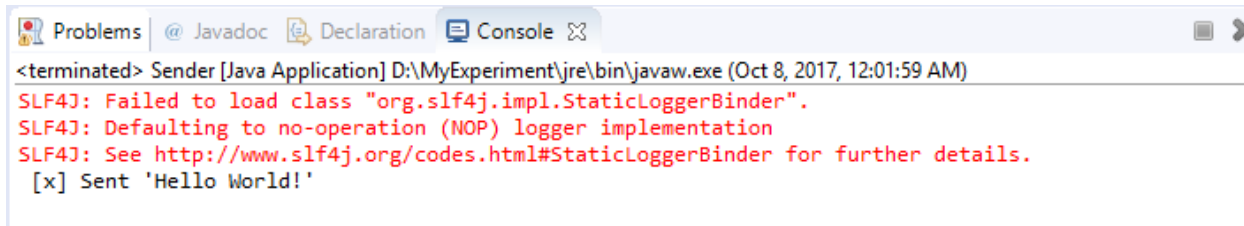
Listening ports

Protocol	Bound to	Port
amqp	::	15674
amqp/ssl	::	5671
clustering	::	25672
http	::	15672

Web contexts

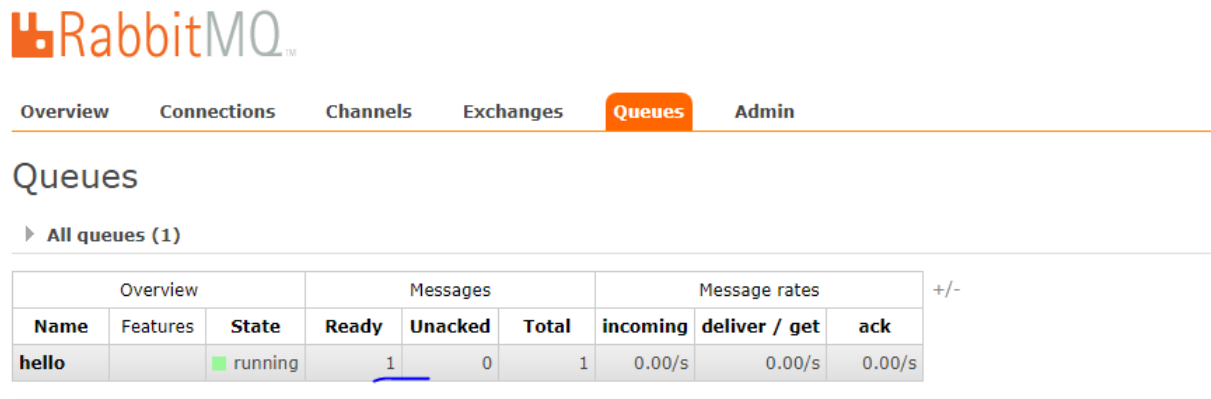
Context	Bound to	Port	SSL	Path
RabbitMQ Management	0.0.0.0	15672	o	/

Execute the main class i.e Sender



```
<terminated> Sender [Java Application] D:\MyExperiment\jre\bin\javaw.exe (Oct 8, 2017, 12:01:59 AM)
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.
[x] Sent 'Hello World!'
```

You can verify the message using web console. You need to logon using guest user and access the Queue, hello which we have declare just now.



RabbitMQ

Overview Connections Channels Exchanges **Queues** Admin

Queues

► All queues (1)

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

As seen above, there will be a single message.

Click on hello -> Get Message to retrieve or consume the message. Leave all the parameter as default.

▼ Get messages

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

Ack Mode:

Encoding:  ?

Messages:

Message 1

The server reported 0 messages remaining.

Exchange	(AMQP default)
Routing Key	hello
Redelivered	0
Properties	
Payload	12 bytes
Encoding: string	Hello World!

If your result is as shown above, then you have pushed message successfully to the Queue, Hello.

Next, let us consume the above message using Java API listener.

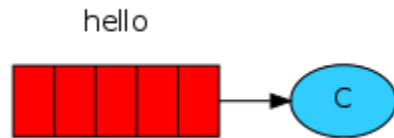


## Subscriber.

In this lab, we will create a subscriber that will consume message from a Queue, hello using Java API.

Our consumer listens 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.

You can use the same java project and create a java class(**Receiver.java**) as shown below:



Most of the code to connect to the broker(Update the broker port accordingly) remains same as in the earlier Publisher project. You can go through the `handleDelivery` method that is invoked when a message is received from the broker.

```
package com.hp.rabbitmq;
```

```
import java.io.IOException;
```

```
import com.rabbitmq.client.AMQP;
```

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

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

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

```
import com.rabbitmq.client.Consumer;
```

```

import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;

public class Receiver {
    private final static String QUEUE_NAME = "hello";
    private final static String QUEUE_HOST = "localhost";
    private final static int PORT = 17674;

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(QUEUE_HOST);
        factory.setPort(PORT);
        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");

        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body)
                throws IOException {

```

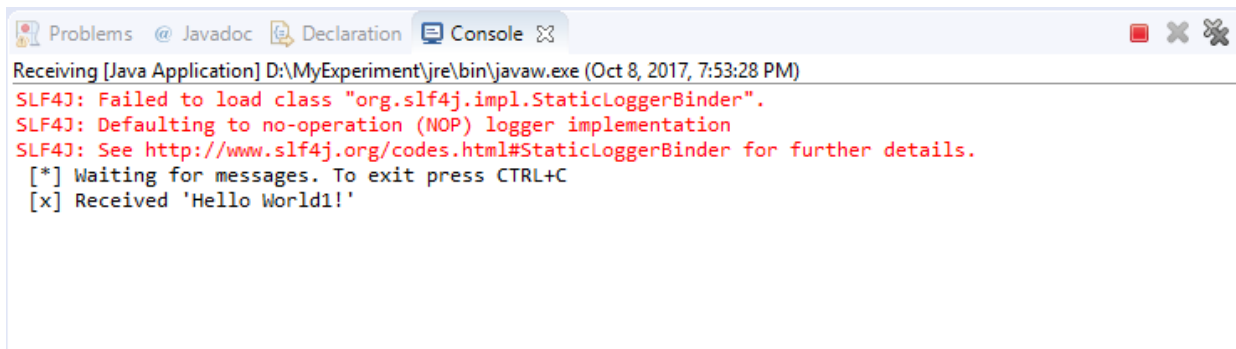
```

        String message = new String(body, "UTF-8");
        System.out.println(" [x] Received '" + message + "'");
    }
};
channel.basicConsume(QUEUE_NAME, true, consumer);
}
}

```

Execute the code.

Result in the consumer terminal.



```

Receiving [Java Application] D:\MyExperiment\jre\bin\javaw.exe (Oct 8, 2017, 7:53:28 PM)
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 World1!'

```

You can view the Queue using the web console. The Message should be o.

Overview Connections Channels Exchanges **Queues** Admin

## Queues

► All queues (1)

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
hello		<span>running</span>	0	0	0	0.00/s	0.00/s	0.00/s	

► Add a new queue

HTTP API | Command Line

----- Lab Ends Here -----



## Using Spring-Boot(RabbitMQ) -60 Minutes

In this lab we will develop a Spring Boot application to interact with RabbitMQ server.

Ports to Connect from Spring boot.

Using Webconsole -> Ports and Contexts.

### ▼ Ports and contexts

Listening ports

Protocol	Bound to	Port
amqp	::	15673
clustering	::	25672
http	::	15672

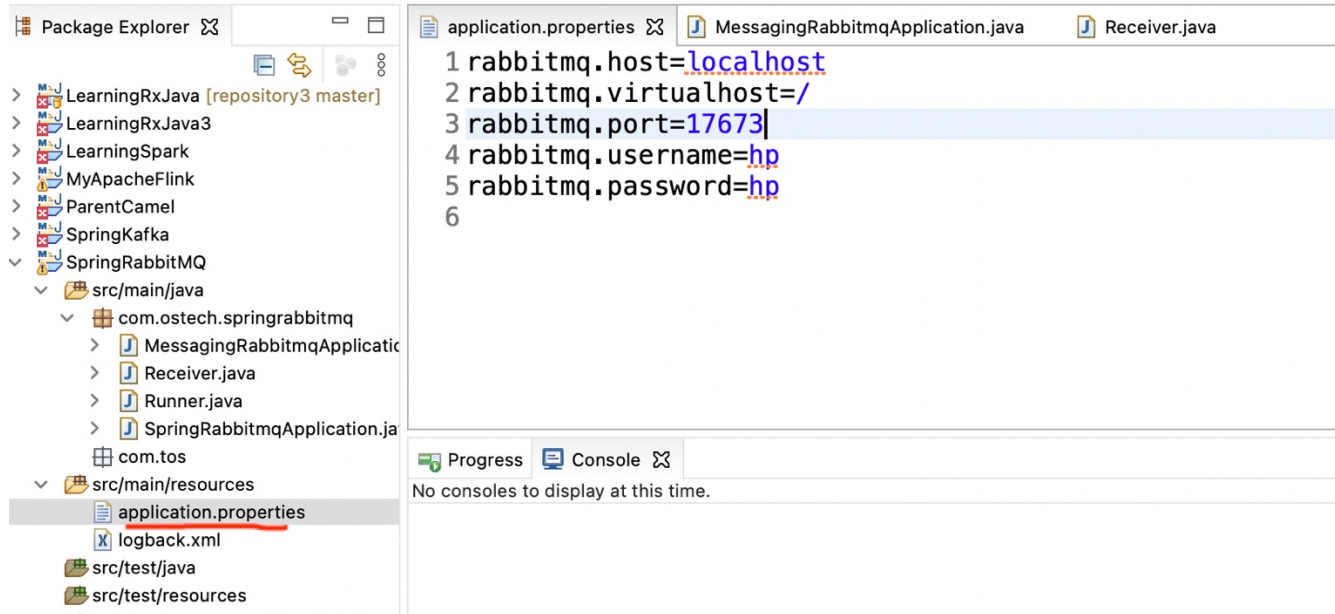
If you are using docker, use the host port mapped with the above container port.

Specify in the rabbitmq config as shown below.

```
[root@rabbitmq0 rabbitmq]# more rabbitmq.conf
# this is a comment
listeners.tcp.default = 15673
[root@rabbitmq0 rabbitmq]# pwd
/etc/rabbitmq
[root@rabbitmq0 rabbitmq]#
```

Specify the connection details in the **application.properties**. Here rabbitmq port is map to 17674 in the docker.

```
rabbitmq.host=localhost  
rabbitmq.virtualhost=/  
rabbitmq.port=17674  
rabbitmq.username=guest  
rabbitmq.password=guest
```



You will build an application that publishes a message by using Spring AMQP's `RabbitTemplate` and subscribes to the message on a POJO by using `MessageListenerAdapter`.

Create a RabbitMQ Message Receiver

With any messaging-based application, you need to create a receiver that responds to published messages. The following listing (from `src/main/java/com.example.messagingrabbitmq/Receiver.java`) shows how to do so:

```
package com.example.messagingrabbitmq;

import java.util.concurrent.CountDownLatch;
import org.springframework.stereotype.Component;

@Component
public class Receiver {

    private CountDownLatch latch = new CountDownLatch(1);

    public void receiveMessage(String message) {
        System.out.println("Received <" + message + ">");
        latch.countDown();
    }

    public CountDownLatch getLatch() {
        return latch;
    }
}
```



```
}
```

```
}
```

The `Receiver` is a POJO that defines a method for receiving messages. When you register it to receive messages, you can name it anything you want.

For convenience, this POJO also has a `CountDownLatch`. This lets it signal that the message has been received.

### Register the Listener and Send a Message

Spring AMQP's `RabbitTemplate` provides everything you need to send and receive messages with RabbitMQ.

However, you need to:

- Configure a message listener container.
- Declare the queue, the exchange, and the binding between them.
- Configure a component to send some messages to test the listener.

Spring Boot automatically creates a connection factory and a `RabbitTemplate`, reducing the amount of code you have to write.

You will use `RabbitTemplate` to send messages, and you will register a `Receiver` with the message listener container to receive messages. The connection factory drives both, letting them connect to the RabbitMQ server. The following listing

(from `src/main/java/com.example.messagingrabbitmq/MessagingRabbitApplication.java`) shows how to create the application class:

```
package com.example.messagingrabbitmq;

import org.springframework.amqp.core.AcknowledgeMode;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
import
org.springframework.amqp.rabbit.connection.CachingConnectionFactory
;
import
org.springframework.amqp.rabbit.connection.ConnectionFactory;
import
org.springframework.amqp.rabbit.listener.SimpleMessageListenerConta
iner;
import
org.springframework.amqp.rabbit.listener.adapter.MessageListenerAda
pter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
```

---

```
public class MessagingRabbitmqApplication {  
  
    static final String topicExchangeName = "spring-boot-exchange";  
  
    static final String queueName = "spring-boot";  
  
    @Value("${rabbitmq.username}")  
    private String username;  
    @Value("${rabbitmq.password}")  
    private String password;  
    @Value("${rabbitmq.host}")  
    private String host;  
  
    @Value("${rabbitmq.port}")  
    private int port;  
  
    @Bean  
    Queue queue() {  
        return new Queue(queueName, false);  
    }  
  
    @Bean  
    TopicExchange exchange() {  
        return new TopicExchange(topicExchangeName);  
    }  
}
```

```

@Bean
Binding binding(Queue queue, TopicExchange exchange) {
    return
BindingBuilder.bind(queue).to(exchange).with("foo.bar.#");
}

```

```

@Bean
public ConnectionFactory connectionFactory() {
    CachingConnectionFactory connectionFactory = new
CachingConnectionFactory();
    // connectionFactory.setVirtualHost(virtualHost);
    connectionFactory.setHost(host);
    connectionFactory.setPort(port);
    connectionFactory.setUsername(username);
    connectionFactory.setPassword(password);

    return connectionFactory;
}
@Bean
SimpleMessageListenerContainer container(ConnectionFactory
connectionFactory,
    MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer();
}

```

```

        container.setConnectionFactory(connectionFactory);
        container.setQueueNames(queueName);
        container.setMessageListener(listenerAdapter);
        return container;
    }

    @Bean
    MessageListenerAdapter listenerAdapter(Receiver receiver) {
        return new MessageListenerAdapter(receiver, "receiveMessage");
    }

    public static void main(String[] args) throws
    InterruptedException {
        SpringApplication.run(MessagingRabbitmqApplication.class,
        args).close();
    }
}

```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation

flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.

- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application.

The bean defined in the `listenerAdapter()` method is registered as a message listener in the container (defined in `container()`). It listens for messages on the `spring-boot` queue. Because the `Receiver` class is a POJO, it needs to be wrapped in the `MessageListenerAdapter`, where you specify that it invokes `receiveMessage`.

JMS queues and AMQP queues have different semantics. For example, JMS sends queued messages to only one consumer. While AMQP queues do the same thing, AMQP producers do not send messages directly to queues. Instead, a message is sent to an exchange, which can go to a single queue or fan out to multiple queues, emulating the concept of JMS topics.

The message listener container and receiver beans are all you need to listen for messages. To send a message, you also need a Rabbit template.

The `queue()` method creates an AMQP queue. The `exchange()` method creates a topic exchange.

The `binding()` method binds these two together, defining the behavior that occurs when `RabbitTemplate` publishes to an exchange.

Spring AMQP requires that the `Queue`, the `TopicExchange`, and the `Binding` be declared as top-level Spring beans in order to be set up properly.

In this case, we use a topic exchange, and the queue is bound with a routing key of `foo.bar.#`, which means that any messages sent with a routing key that begins with `foo.bar.` are routed to the queue.

#### Send a Test Message

In this sample, test messages are sent by a `CommandLineRunner`, which also waits for the latch in the receiver and closes the application context. The following listing

(from `src/main/java/com.example.messagingrabbitmq/Runner.java`) shows how it works:

```
package com.example.messagingrabbitmq;

import java.util.concurrent.TimeUnit;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class Runner implements CommandLineRunner {

    private final RabbitTemplate rabbitTemplate;
    private final Receiver receiver;

    public Runner(Receiver receiver, RabbitTemplate rabbitTemplate) {
        this.receiver = receiver;
    }
}
```

```

    this.rabbitTemplate = rabbitTemplate;
}

@Override
public void run(String... args) throws Exception {
    System.out.println("Sending message...");
    System.out.println(" Host : " +
rabbitTemplate.getConnectionFactory().getHost() + " ");

    rabbitTemplate.convertAndSend(MessagingRabbitmqApplication.topicExchangeName, "foo.bar.baz", "Hello from RabbitMQ!");
    receiver.getLatch().await(10000, TimeUnit.MILLISECONDS);
}
}

```

Notice that the template routes the message to the exchange with a routing key of `foo.bar.baz`, which matches the binding.

In tests, you can mock out the runner so that the receiver can be tested in isolation.

Run the Application

The `main()` method starts that process by creating a Spring application context. This starts the message listener container, which starts listening for messages. There is a `Runner` bean, which is then automatically run. It retrieves



the `RabbitTemplate` from the application context and sends a `Hello from RabbitMQ!` message on the `spring-boot` queue. Finally, it closes the Spring application context, and the application ends.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR. Here, we will run from the IDE itself.

Right click on `MessagingRabbitmqApplication.java` -> Run As -> java Application.

```

20:46:19.266 [main] INFO  c.o.s.MessagingRabbitmqApplication - Started MessagingRabbitmqApplication in
9.932 seconds (JVM running for 10.843)
Sending message...
Host : localhost)
Received <Hello from RabbitMQ!>
20:46:19.316 [main] INFO  o.s.a.r.l.SimpleMessageListenerContainer - Waiting for workers to finish.

```

On the web console.

You should observe the Queue name – `spring-boot`

Reload this page  RabbitMQ 3.12.0 Erlang 25.3.2.2

Overview Connections Channels Exchanges **Queues** Admin

## Queues

▼ All queues (2)

Pagination

Page  of 1 - Filter:  ☐ Regex ?

Overview					Messages			Message rates			+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/	spring-boot	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
/	test	classic	D	idle	0	0	0				

▼ Add a new queue

Virtual host:

Type:

Name:  \*

Durability:

Arguments:  =

Add  ? |  ? |  ?

? |  ? |  ?

? |  ?

|  ?

Make Acknowledgement as Manual in the java file `MessagingRabbitmqApplication` -> `container()` - Method.

```
container.setAcknowledgeMode( AcknowledgeMode.MANUAL);
```

```
@Bean
```

```
SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,
    MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(
        container.setConnectionFactory(connectionFactory);
        container.setQueueNames(queueName);
        container.setAcknowledgeMode( AcknowledgeMode.MANUAL);
        container.setMessageListener(listenerAdapter);
        return container;
    }
```

Now You can verify the message from the web console. Since the consumer has not ack.

Execute the above program again.

On the web console. You should be able to notice a message in the Queue as shown below.



RabbitMQ 3.12.0

Erlang 25.3.2.2

Overview

Connections

Channels

Exchanges

Queues

Admin

## Queues

▼ All queues (2)

Pagination

Page  of 1 - Filter:  ☐ Regex ?

Overview					Messages			Message rates			+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/	<u>spring-boot</u>	classic		idle	1	0	1	0.00/s	0.00/s	0.00/s	
/	test	classic	D	idle	0	0	0				

▼ Add a new queue

Let us verify the message from the web UI.

**Queues -> spring-boot -> Get messages**

## ▼ Get messages

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

Ack Mode:

Encoding:  ?

Messages:

Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange	spring-boot-exchange
Routing Key	foo.bar.baz
Redelivered	•
Properties	priority: 0 delivery_mode: 2 headers: content_encoding: UTF-8 content_type: text/plain
Payload 20 bytes Encoding: string	Hello from RabbitMQ!

You should be able to verify the message as shown above. Clean the Queue before proceeding to next lab.

## Summary

Congratulations! You have just developed a simple publish-and-subscribe application with Spring and RabbitMQ. You can do more with [Spring and RabbitMQ](#) than what is covered here, but this guide should provide a good start.

## Using JSON Message – 60 Minutes

You need to complete the previous lab for completing this lab. In this lab, we will publish a JSON message and then will consume it.

The parts to be define for this lab are same as above with some modifications.

All the necessary classes will be stored in a package: `com.example.json`

First Let us define a class to represent a `User` object. It has three attributes : `id` , `firstName` and `lastName`.

```
package com.example.json;

import java.io.Serializable;

public class User implements Serializable {
    private int id;
    private String firstName;
    private String lastName;

    public User(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

```
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
}
```

Runner.java

Here in the below code, User object is converted into json string before sending to the RabbitMq.

```
User user = new User(12,"Henry","Potsangbam");
Gson gson = new Gson();
String json = gson.toJson(user);
```

Update the above java file with the following content.

```
package com.example.json;
```

```
import java.util.concurrent.TimeUnit;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import com.google.gson.Gson;
```

```
@Component
```

```
public class Runner implements CommandLineRunner {

    private final RabbitTemplate rabbitTemplate;
    private final Receiver receiver;

    public Runner(Receiver receiver, RabbitTemplate rabbitTemplate) {
```



```
    this.receiver = receiver;
    this.rabbitTemplate = rabbitTemplate;
}

@Override
public void run(String... args) throws Exception {
    System.out.println("Sending message...");
    User user = new User(12, "Henry", "Potsangbam");
    Gson gson = new Gson();
    String json = gson.toJson(user);
    rabbitTemplate.setExchange(MessagingJMSRA.topicExchangeName);
    rabbitTemplate.setRoutingKey("foo.bar.baz");
    rabbitTemplate.convertAndSend(json);
    receiver.getLatch().await(10000, TimeUnit.MILLISECONDS);
}
}
```

## Receiver.java

Here, In the Receiver code. We define a Gson object that will convert the message string to an User object.

```
Gson gson = new Gson();  
User user = gson.fromJson(message.toString(), User.class);
```

Update the above class with the following content.

```
package com.example.json;  
  
import java.util.concurrent.CountDownLatch;  
  
import org.springframework.stereotype.Component;  
  
import com.google.gson.Gson;  
  
@Component  
public class Receiver {  
  
    private CountDownLatch latch = new CountDownLatch(1);  
  
    public void receiveMessage(final Object message) {  
        Gson gson = new Gson();  
        User user = gson.fromJson(message.toString(), User.class);  
    }  
}
```

```
        System.out.println("FirstName : " + user.getFirstName() );  
        latch.countDown();  
    }  
  
    public CountDownLatch getLatch() {  
        return latch;  
    }  
  
}
```

Finally, update the following code in MessagingJSONRA.java which is the spring component.

```
package com.example.json;

import org.springframework.amqp.core.AcknowledgeMode;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
import
org.springframework.amqp.rabbit.connection.CachingConnectionFactory
;
import
org.springframework.amqp.rabbit.connection.ConnectionFactory;
import
org.springframework.amqp.rabbit.listener.SimpleMessageListenerConta
iner;
import
org.springframework.amqp.rabbit.listener.adapter.MessageListenerAda
pter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```
@SpringBootApplication
public class MessagingJSONRA {
    static final String topicExchangeName = "spring-boot-exchange";

    static final String queueName = "spring-boot";

    @Value("${rabbitmq.username}")
    private String username;
    @Value("${rabbitmq.password}")
    private String password;
    @Value("${rabbitmq.host}")
    private String host;

    @Value("${rabbitmq.port}")
    private int port;

    @Bean
    Queue queue() {
        return new Queue(queueName, false);
    }

    @Bean
    TopicExchange exchange() {
```

```

    return new TopicExchange(topicExchangeName);
}

@Bean
Binding binding(Queue queue, TopicExchange exchange) {
    return
BindingBuilder.bind(queue).to(exchange).with("foo.bar.#");
}

@Bean
public ConnectionFactory connectionFactory() {
    CachingConnectionFactory connectionFactory = new
CachingConnectionFactory();
    connectionFactory.setHost(host);
    connectionFactory.setPort(port);
    connectionFactory.setUsername(username);
    connectionFactory.setPassword(password);

    return connectionFactory;
}

@Bean
SimpleMessageListenerContainer container(ConnectionFactory
connectionFactory,
    MessageListenerAdapter listenerAdapter ) {

```

```
SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames(queueName);
    container.setAcknowledgeMode(AcknowledgeMode.MANUAL);
    container.setMessageListener(listenerAdapter);
    return container;
}

@Bean
MessageListenerAdapter listenerAdapter(Receiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
}

public static void main(String[] args) throws
InterruptedException {
    SpringApplication.run(MessagingJSONRA.class, args).close();
}
```

Execute the `MessagingJSONRA.java`

- 1) The Message Received in json format.
- 2) Extracted the firstName attribute of the json message.

```
18:59:52.387 [main] INFO com.example.json.MessagingJSONRA - Started MessagingJSONRA in 8.195 seconds
(JVM running for 9.062)
Sending message...
Receiving : {"id":12,"firstName":"Henry","lastName":"Potsangbam"}
FirstName : Henry
18:59:52.574 [container-2] INFO o.s.a.r.l.SimpleMessageListenerContainer - Waiting for workers to
finish.
```

You can verify from the web console too.

Queues -> [spring-boot](#) -> Get messages



▼ **Get messages**

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

Ack Mode:

Encoding:  ?

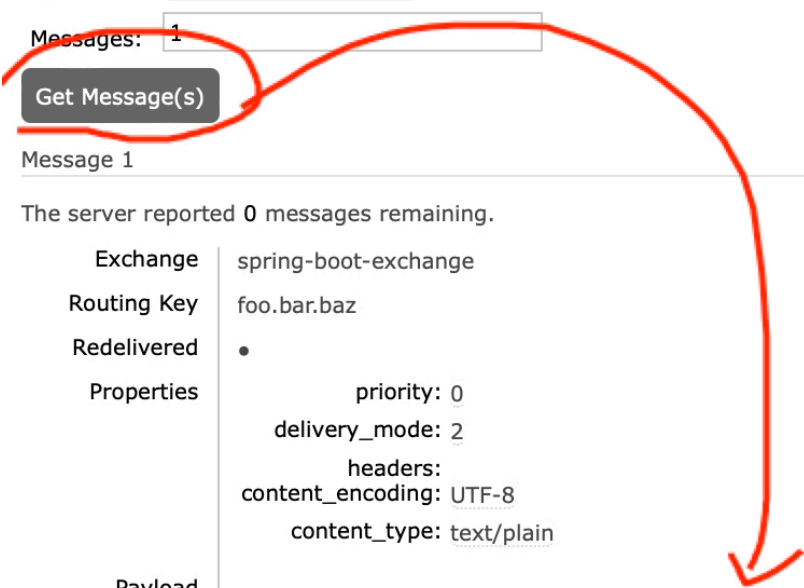
Messages:

**Get Message(s)**

Message 1

The server reported 0 messages remaining.

Exchange	spring-boot-exchange
Routing Key	foo.bar.baz
Redelivered	•
Properties	<div>priority: 0</div> <div>delivery_mode: 2</div> <div>headers:</div> <div>content_encoding: UTF-8</div> <div>content_type: text/plain</div>
Payload	
53 bytes	
Encoding: string	<pre>{"id":12,"firstName":"Henry","lastName":"Potsangbam"}</pre>



----- lab ends here -----

**Pom.xml**

This pom is configured for Java Client and Spring boot integration.

```
<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.ostech</groupId>
  <artifactId>LearningRabbitmq</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.0</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>
  <properties>
    <java.version>11</java.version>
  </properties>
  <dependencies>
    <dependency>
```

```
<groupId>com.rabbitmq</groupId>
<artifactId>amqp-client</artifactId>
<version>5.18.0</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</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>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
</dependency>
<dependency>
  <groupId>com.googlecode.json-simple</groupId>
```

```
        <artifactId>json-simple</artifactId>
        <version>1.1.1</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.6.2</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>
```