



# INDEX

## Spring Boot Messaging -----

1. Messaging	<a href="#"><u>04</u></a>
2. Messaging/ Message Queue using JMS	<a href="#"><u>08</u></a>
a. Procedure to keep Active MQ as MOM software	<a href="#"><u>12</u></a>
b. Procedure to develop JMS PTP (Queue) application using ActiveMQ	<a href="#"><u>14</u></a>
c. Developing JMS Pub-Sub Model app using ActiveMQ	<a href="#"><u>20</u></a>
d. Developing ActiveMQ PTP application to send object as message	<a href="#"><u>25</u></a>
e. Limitation of JMS	<a href="#"><u>30</u></a>
3. Apache Kafka	<a href="#"><u>31</u></a>
a. Apache Kafka Work Flow Diagram	<a href="#"><u>35</u></a>
b. Arranging Apache Kafka Software	<a href="#"><u>37</u></a>
c. Developing Kafka Procedure as Kafka client in Legacy style	<a href="#"><u>40</u></a>
d. Developing Kafka consumer as Kafka client in Legacy style	<a href="#"><u>44</u></a>
4. Spring Boot + Apache Kafka API	<a href="#"><u>48</u></a>

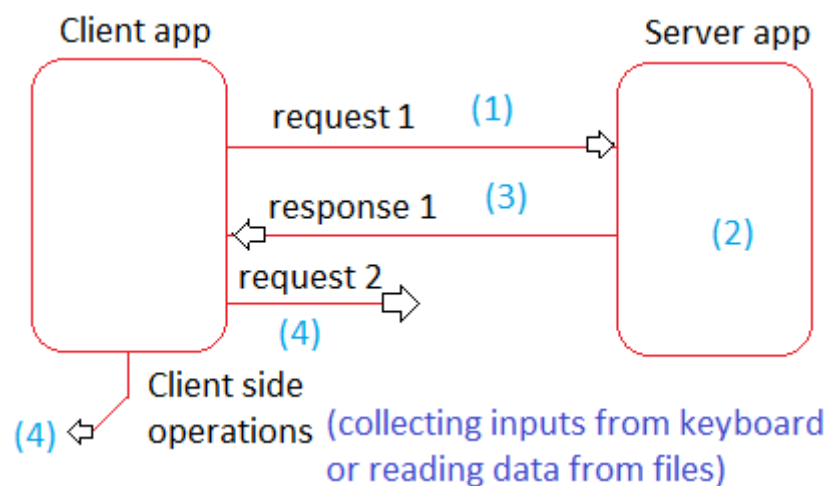
# Spring Boot Messaging

## Messaging

- ✚ Messaging means communication using Messages.
- ✚ The Client - Server Communication is of two types
  - a. Synchronous communication
  - b. Asynchronous Communication

### Synchronous communication

- In this communication, the client app can send next request to server app only after getting the response for already given request i.e., the client app should wait for given request related response from server app in order to make it next request to server app or to perform some client-side operations.



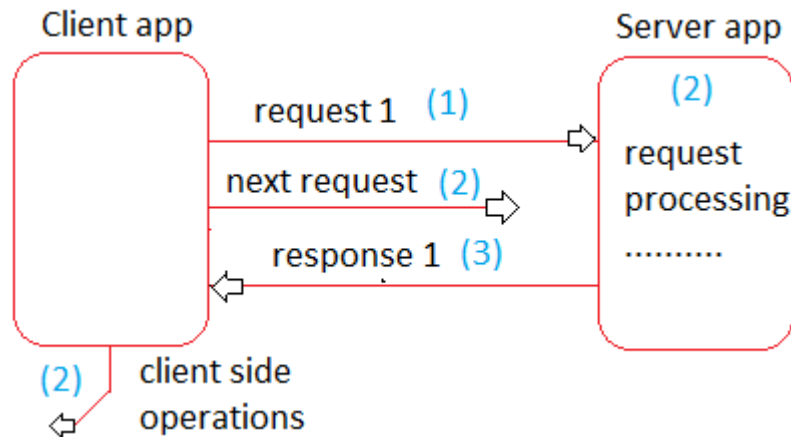
- Here the Client app is blocked to generate next request or to perform client-side operations until given request related response comes back from server app.
- In synchronous communication both client app and server app must be active at a time.

**Note:** By default, all Client - server communications are synchronous communications.

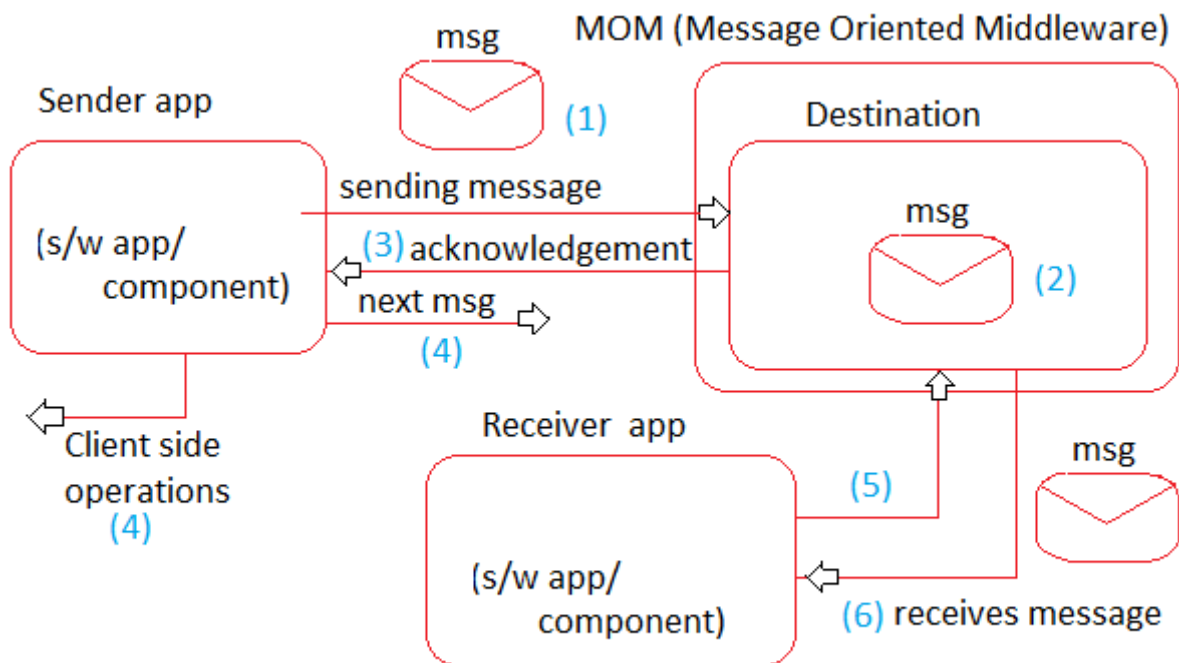
### Asynchronous Communication

- Here Client app is free (not blocked) to generate the next request or to perform client-side operations without waiting for the given request related response from server app.
- In web applications we can use AJAX (Asynchronous Java Script and XML) to get asynchronous communication b/w browser (client) and

web application (server) (now AJAX is part of UI Technologies).



- We can use "Messaging concept" (Messages based communication) to get Asynchronous communication b/w two software apps/ components.



(Messages based Asynchronous Communication)

- MOM is a software that acts as middle man for both sender and receiver and it contains memory called Destination to hold messages sent by sender app and to deliver the same messages to the receiver app.

**Note:**

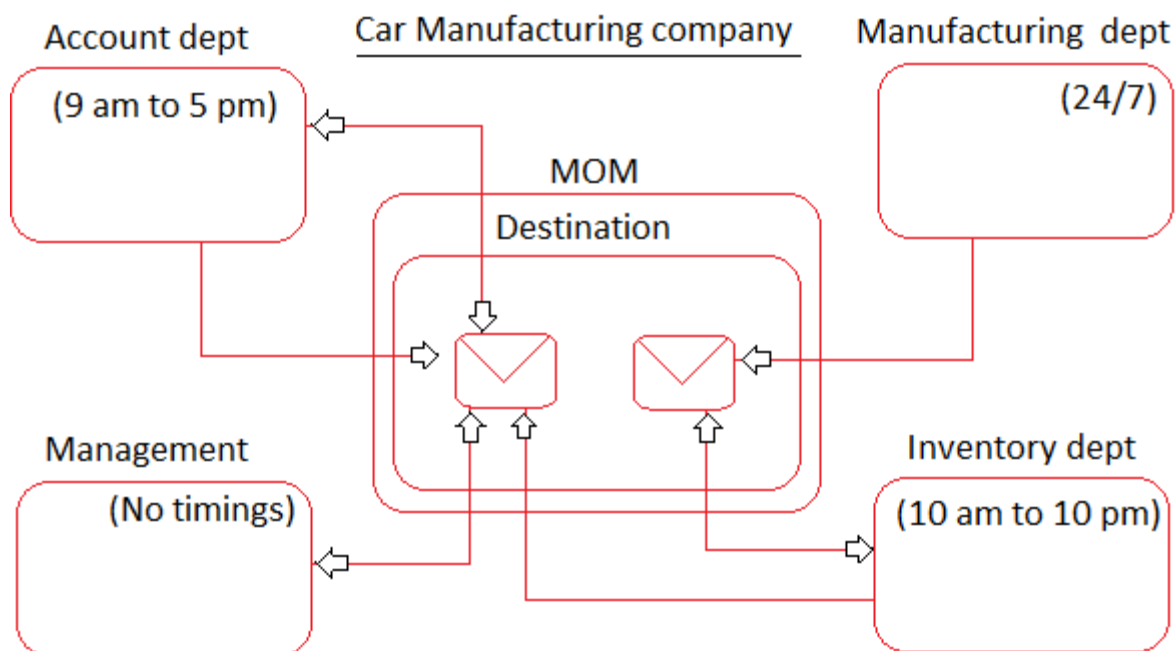
- ✓ In Messaging the messages will go from one to app to another app in continuous flow like stream.
- ✓ SMS messaging, WhatsApp messaging, mailing does not come under

messages based communication because that deals with person to person communication.

- ✓ The actual messaging takes place b/w two software app or components.

#### Use-cases for Messaging:

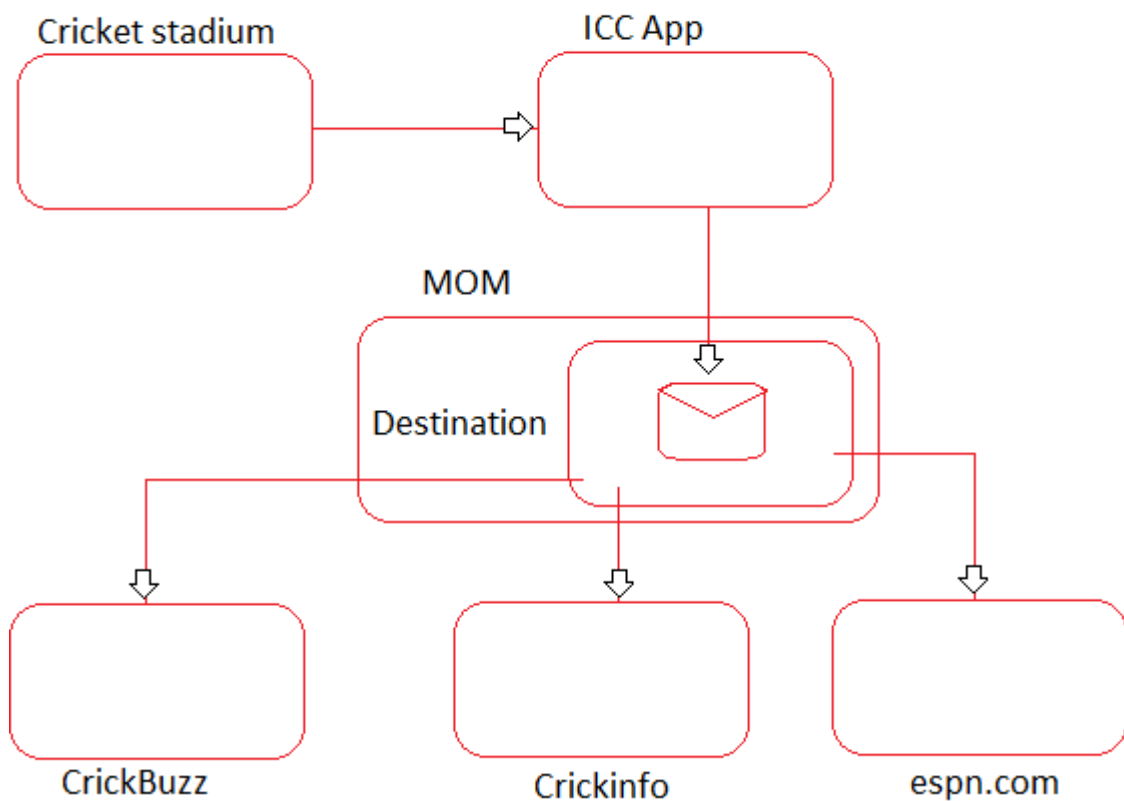
- Data streaming based on scheduled/ continuous flow.  
E.g.: Uber cab availability status, Goods delivery app store availability status, Live train running status, Live cricket score status and etc.
- Web activities and search results  
E.g.: The advertisement agency app gathers google search queries continuously to aggregate search queries.
- Log aggregation  
E.g.: Collecting log messages generated by production environment app and aggregating those messages.



- Since all departments related apps/ modules cannot be active at a time, so synchronous communication among departments is not possible. So, prefer messages based Asynchronous communication.

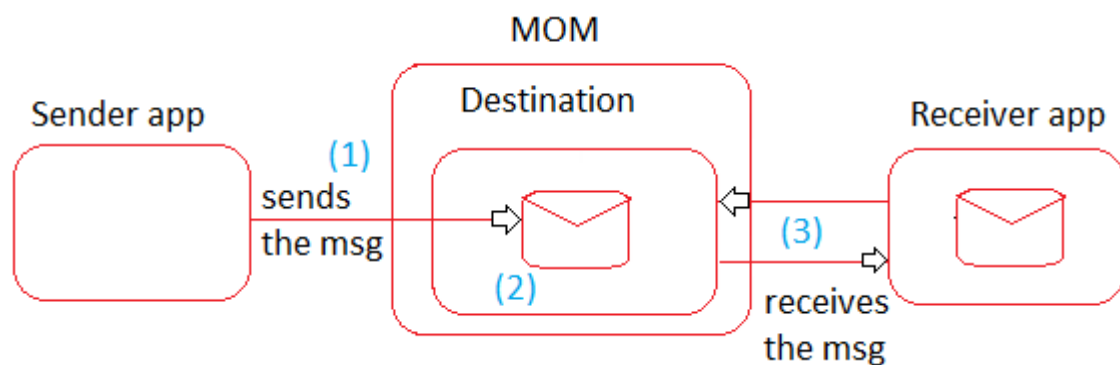
- ✚ This Messaging based communication can be done in two ways
  - as Point to Point (PTP) communication
  - as Publisher and Subscriber (Pub - Sub) communication

### ICC cricket Score Live Stream



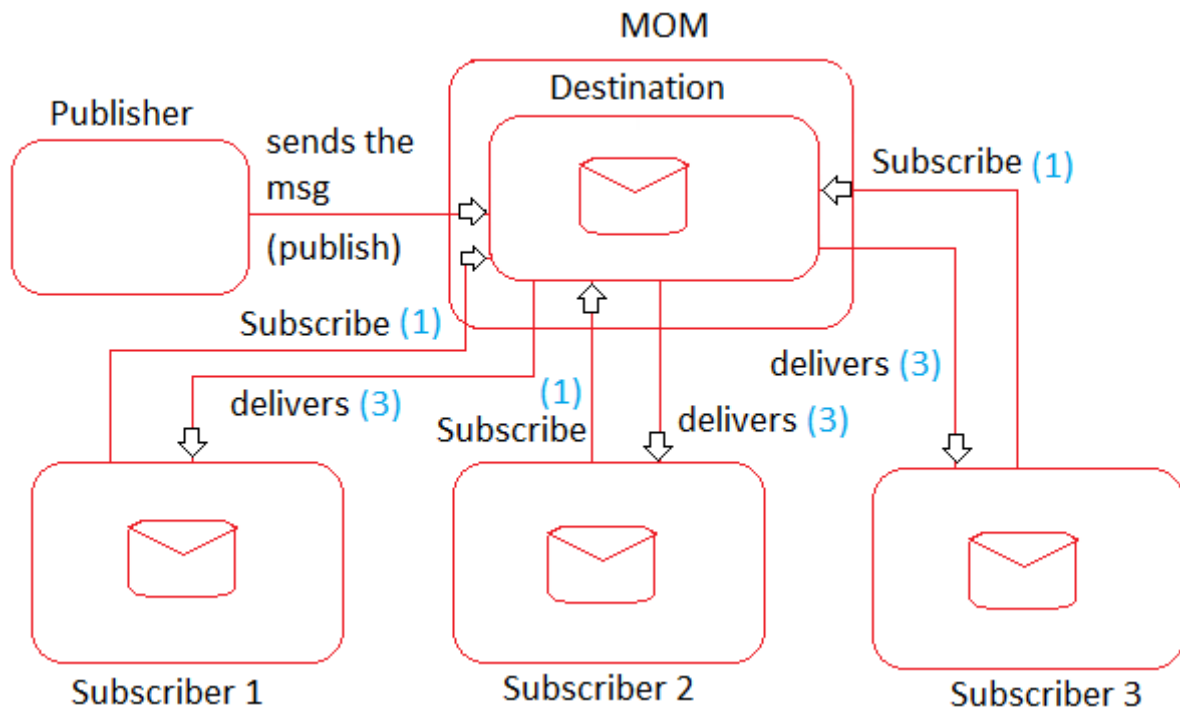
#### Point to Point (PTP) communication:

- Here each message send to destination by sender will have only one consumer/ receiver i.e., once consumer consumes the messages the message will be removed from the destination.



#### Publisher and subscriber (Pub - Sub) communication:

- Here each message sent by the publisher will go to multiple subscribers who have done their subscription to destination before publisher publishes the message.
- Each message can have 0 or more subscribers.

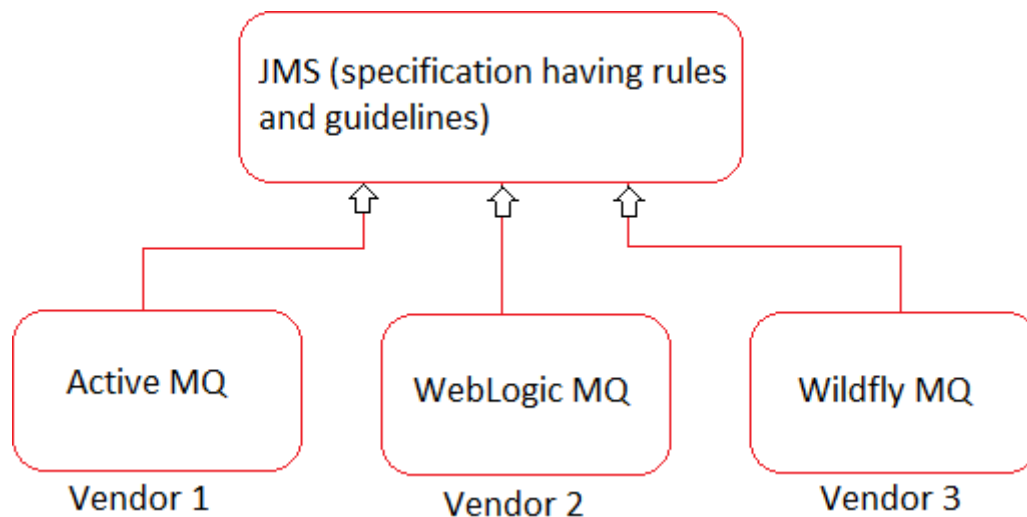


- By default, all client app-server app communication is synchronous communication.  
E.g., browser to web application, Rest client to Rest service, MS client to MS, MS to MS.
- In web application to get asynchronous communication b/w browser and web application take the support of AJAX.
- Between two Java apps or two MS or Rest client and Rest service if you are looking for Messages based asynchronous communication then for Messaging/ Message Queues with the support of MOM software.
- Messaging/ Message Queue can be implemented using two types of protocols
  - a. Using Basic MQ Protocol (BMQP):  
For this we need to use JMS
  - b. Using Advanced MQ Protocol (AMQP):  
For this we need to use Rabbit MQ, Apache Kafka and etc.

## Messaging/ Message Queue using JMS

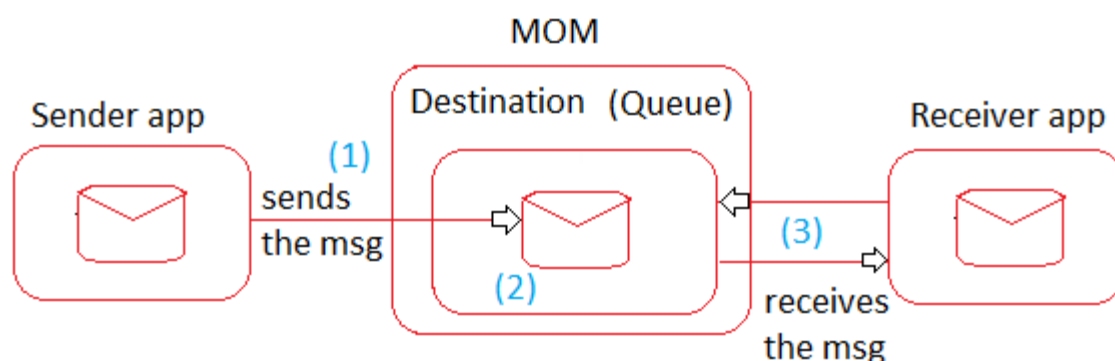
- ✚ JMS stands for Java/ Jakarta Messaging Service.
- ✚ Java Mail API is different from JMS.
- ✚ JMS is the software specification given by Sun Microsystems in JEE module having set of rules and guidelines to provide messages-based communication b/w two Java components or apps.

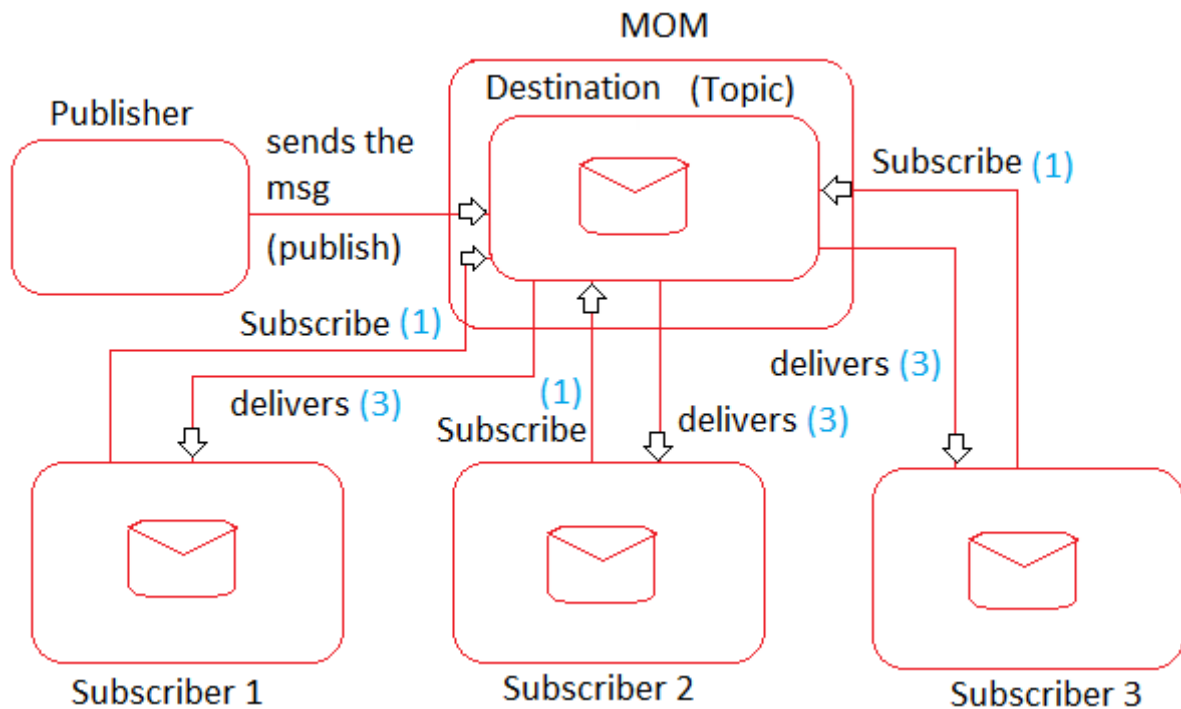




- ✚ Each implementation software of JMS given by different vendors is called one MOM software.
- ✚ Every application server s/w like WebLogic, Wildfly and etc. given one built-in MOM software.
- ✚ Tomcat is not providing any MOM software so, if your application is using tomcat server then prefer working with Active MQ as the MOM software.
- ✚ JMS supports both PTP and Pub to Sub models of messaging.
- ✚ The Middle man software between sender and receiver/ consumer who takes messages given by sender and holds them for receiver to come and consume is called MOM.
- ✚ MOM contains destinations as the logical memories to receive and hold the messages.
- ✚ In PTP model this destination is called "Queue" and in pub-sub model this destination is called "Topic".

**Note:** If someone says Queue means he want Point to Point communication and if someone says Topic means he want Publisher to Subscriber communication.





- JMS provides API representing rules and guidelines in the form of interfaces, classes placed in `javax.jms` or `jakarta.jms` and its sub packages (It is like JDBC API).
- Active MQ, WebLogic MQ, Wildfly MQ and etc. are MOM softwares which are internally providing implementation classes for JMS API interfaces (these are like JDBC drivers).
- Programmers takes this implement classes through JMS interfaces to create objects and to consume services.
- The sender and receiver apps will be developed using JMS API referring one or another MOM implementation softwares (these are like Java apps using JDBC drivers through JDBC API)

Sample reference code to understand API, implementation software and App development using API:

```

interface Operation1 {
    public void process ();
}

interface Operation2 {
    public String process (String msg);
}

```

It is like JMS API given by Sun Microsystem (JMS specification)

```
public class Operation1Impl implements Operation1
    public void process () {
        .....
    }
}
```

```
public class Operation2Impl implements Operation2
    public void process (String msg) {
        .....
    }
}
```

Implementation classes provide by the vendor.  
(It is like Active MQ, WebLogic MQ and etc. MOM softwares give by different vendors)

```
Operation1 op1=new Operation1Impl ();
op1.process();
Operation2 op2=new Operation2Impl ();
String result: op2.process("hello");
```

It Like JMS application developed the Programmer.

**Note:** The interface that contains only one abstract method directly or indirectly is called functional Interface.

### Functional interface

```
@FunctionalInterface
interface Operation1 {
    public void process (String msg);
}
```

### Implementation 1: Normal implementation class

```
public class Operation1Impl implements Operation1 {
    public void process (String msg) {
        .....
    }
}
```

### Implementation 2: Anonymous inner class

```
Operation1 op1 = new Operation1() {
    public void process (String msg) {
        .....
        .....
    }
};
```

Here Anonymous inner class is created implementing Operation1 interface and process (-) is implemented in that class.

Implementation 3: Lambda based anonymous inner class

Operation1 op1 = (msg) -> {.....}

Anonymous inner class Implementation class object + method implementation, this also called as inline implementation.

The Spring JMS API is providing one Functional Interface called "MessageCreator" as shown below,

@FunctionalInterface

```
public interface MessageCreator {  
    Message createMessage (Session session);  
}
```

Implementation 1: Using anonymous inner class

```
MessageCreator msgCreator = new MessageCreator () {  
    public Message createMessage (Session ses) {  
        .....  
        ..... // logic  
        return message obj;  
    }  
};
```

Implementation 2: Using Lambda anonymous inner class

```
MessageCreator msgCreator = (ses)-> {.....  
    .....  
    return message obj;  
};
```

(or)

```
MessageCreator msgCreator = ses-> message obj;
```

**Note:** Spring Boot JMS/ Spring JSMS provides abstraction on plain JMS to simplify the message-based communication with the support of "JmsTemplate" (template method design pattern).

## Procedure to keep ActiveMQ as MOM software

**Step 1:** Download ActiveMQ from internet check the JDK compatibility.

[ActiveMQ 5.16.5](#) and [ActiveMQ website link](#)

## ActiveMQ 5.16.5 (May 2nd, 2022)

[Release Notes](#) | [Release Page](#) | [Documentation](#) | Java compatibility: **8+**

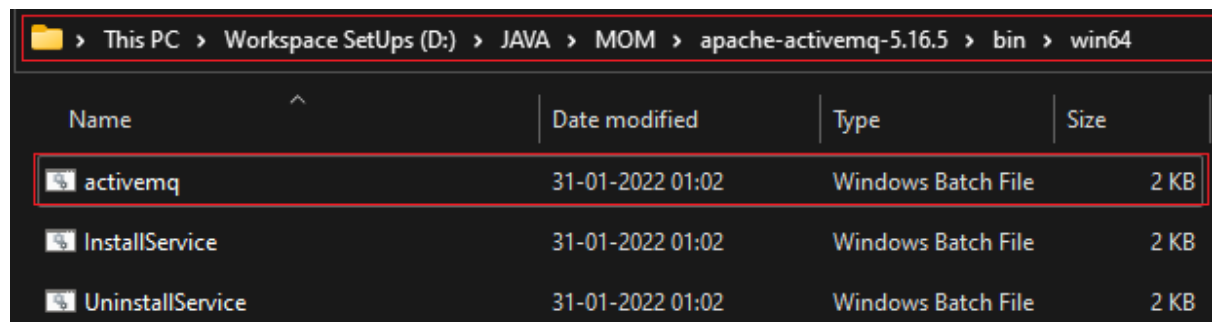
**NOTE:** This is the last planned 5.16.x release. Users should upgrade to the current 5.17.x stream for ongoing releases.

Windows	<a href="#">apache-activemq-5.16.5-bin.zip</a>	SHA512	GPG Signature
Unix/Linux/Cygwin	<a href="#">apache-activemq-5.16.5-bin.tar.gz</a>	SHA512	GPG Signature
Source Code Distribution:	<a href="#">activemq-parent-5.16.5-source-release.zip</a>	SHA512	GPG Signature

**Step 2:** Extract the ZIP file to a folder.

**Step 3:** Start the ActiveMQ software

D:\JAVA\MOM\apache-activemq-5.16.5\bin\win64\activemq.bat

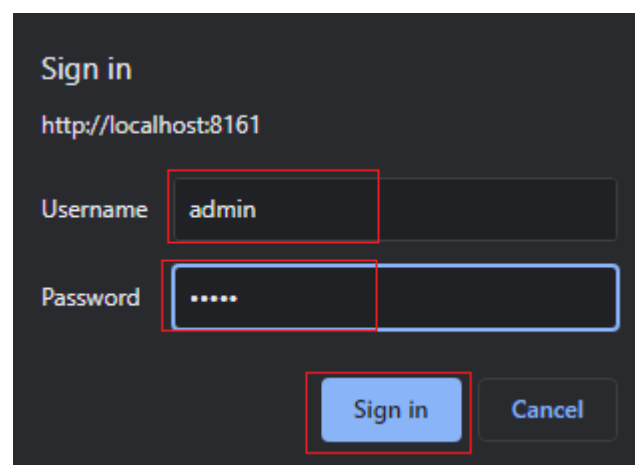


**Step 4:** Open admin console page of ActiveMQ software

<http://localhost:8161>

Username: admin

Password: admin



**Step 5:** Observer Topic and Queue section in admin console page. For that click on Manage ActiveMQ broker then you can get the all required things.

## Welcome to the Apache ActiveMQ!

What do you want to do next?

- Manage ActiveMQ broker
- See some Web demos (demos not included in default configuration)

[Home](#) | [Queues](#) | [Topics](#) | [Subscribers](#) | [Connections](#) | [Network](#) | [Scheduled](#) | [Send](#)

### Procedure to develop JMS PTP (Queue) application using ActiveMQ

- ✚ In any JMS App (Producer/ Sender or Receiver/ Subscriber/ consumer) once we spring-boot-starter-activemq starter as dependency we get JmsTemplate class object through AutoConfiguration that can be injected to Sender/ Receiver App.

#### For Producer App

**Step 1:** Create Spring Boot project add the following starter and choose Packaging type as Jar.

X Spring for Apache ActiveMQ 5

**Step 2:** Add the following properties in application.properties file.

#MOM Connective details

spring.activemq.broker-url=tcp://localhost:61616

spring.activemq.user=admin

spring.activemq.password=admin

#Enable PTP communication

#true enables Pub-sub model (default) and false enables PTP model

spring.jms.pub-sub-domain=false

**Step 3:** Develop runner class as the Message sender having sending logic.

**Note:** JmsTemplate class is having send (-,-) method taking destination (queue/ topic) logical name (generally new name) and MessageCreator (I) (Functional interface).

```
public void send(Destination destination, MessageCreator messageCreator)
                throws JmsException
```

For MessageCreator we can pass either anonymous inner class object or Lambda style inner class object.

**Step 4:** Make sure that Active MQ started.

**Step 5:** Run the Sender app and Observe ActiveMQ Server console Queue section.

### Queues:

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
testmq1	1	0	1	0	<a href="#">Browse</a> <a href="#">Active Consumers</a> <a href="#">Active Producers</a> <a href="#">atom</a> <a href="#">rss</a>	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a> <a href="#">Pause</a>

### For Consumer/ Receiver App

**Step 1:** Create Spring Boot project add the following starter and choose Packaging type as Jar.

X Spring for Apache ActiveMQ 5

**Step 2:** Add the following properties in application.properties file.

#MOM Connective details

spring.activemq.broker-url=[tcp://localhost:61616](#)

spring.activemq.user=[admin](#)

spring.activemq.password=[admin](#)

#Enable PTP communication

#true enables Pub-sub model (default) and false enables PTP model

spring.jms.pub-sub-domain=[false](#)

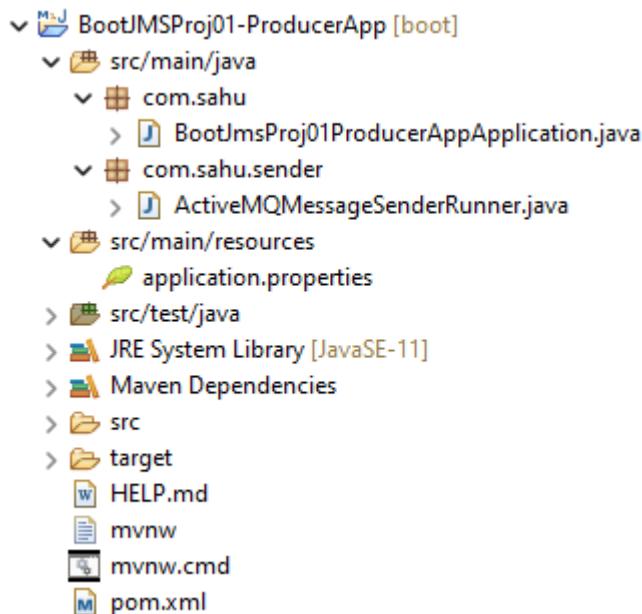
**Step 3:** Develop the Java class having @JmsListener method to consume the message.

**Step 4:** Run the application see the console and check the ActiveMQ console.

### Queues:

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
testmq1	0	1	1	1	<a href="#">Browse</a> <a href="#">Active Consumers</a> <a href="#">Active Producers</a> <a href="#">atom</a> <a href="#">rss</a>	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a> <a href="#">Pause</a>

## Directory Structure of BootJMSProj01-ProducerApp:



- Develop the above directory structure using Spring Starter project option and choose packaging as Jar and create the package, class also.
- Use the following starter during project creation.

X Spring for Apache ActiveMQ 5

- Then place the following code with in their respective files.

### application.properties

```
#MOM Connective details
#8161 for admin console and 61616 actual MOM service
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Enable PTP communication
#true enables Pub-sub model (default) and false enables PTP model
spring.jms.pub-sub-domain=false
```

### ActiveMQMessageSenderRunner.java

```
package com.sahu.sender;

import java.util.Date;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
```



```

import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;


















@Component
public class ActiveMQMessageSenderRunner implements
CommandLineRunner {

    @Autowired
    private JmsTemplate template;

    @Override
    public void run(String... args) throws Exception {
        /*template.send("testmq1", new MessageCreator() {
            @Override
            public Message createMessage(Session session) throws
JMSEException {
                Message message =
session.createTextMessage("From sender at : "+new Date());
                return message;
            }
        });*/
        template.send("testmq1", ses-> ses.createTextMessage("From
sender at : "+new Date()));
        System.out.println("Message Sent");
    }
}

```

### Directory Structure of BootJMSProj01-ReceiverApp

- ▼  BootJMSProj01-ReceiverApp [boot]
    - ▼  src/main/java
      - ▼  com.sahu
        - >  BootJmsProj01ReceiverAppApplication.java
      - ▼  com.sahu.receiver
        - >  JMSMessageConsumer.java
    - ▼  src/main/resources
      -  application.properties
    - >  src/test/java
    - >  JRE System Library [JavaSE-11]
    - >  Maven Dependencies
    - >  src
    - >  target
    - >  HELP.md
-  mvnw  
 mvnw.cmd  
 pom.xml

- Develop the above directory structure using Spring Starter project option and choose packaging as Jar and create the package, class also.
- Use the following starter during project creation.

**X** Spring for Apache ActiveMQ 5

- Then place the following code with in their respective files.

### application.properties

```
#MOM Connective details
#8161 for admin console and 61616 actual MOM service
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Enable PTP communication
#true enables Pub-sub model (default) and false enables PTP model
spring.jms.pub-sub-domain=false
```

### JMSMessageConsumer.java

```
package com.sahu.receiver;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class JMSMessageConsumer {

    @JmsListener(destination = "testmq1")
    public void readMessage(String text) {
        System.out.println("Received message is : "+text);
    }
}
```

### Key points about PTP model messaging:

- a. The destination name is "Queue" (FIFO rule).
- b. Both Sender and receiver need not be active at a time.
- c. One Message will have only one Receiver/ Consumer.
- d. The Queue destination can send message to receiver app who connected to the destination before sender sends the message will also the receive the message when the sender sends the message.

- e. If multiple consumers are waiting for a message sent by the Sender, then only first receiver receives the message.
- f. The queue destination delivers the message to Receiver app and deletes the message.

**Use case:** Our car factory use case needs this model (PTP) communication.

### How to make Sender app of PTP Model sending message continuously to MOM software

- We can take the support of scheduling concept for that we need to add `@EnableScheduling` annotation on the top of main class and `@Scheduled` annotation on the top of business method of Sender app.

**Note:** `@Scheduled` can't be applied on the method with args so, make sure that your business method is designed having no args/ params.



#### [BootJmsProj01ProducerAppApplication.java](#)

```
@SpringBootApplication
@EnableScheduling
public class BootJmsProj01ProducerAppApplication {

    public static void main(String[] args) {

        SpringApplication.run(BootJmsProj01ProducerAppApplication.class,
args);
    }

}
```

▼  com.sahu.sender  
>  ActiveMQMessageSender.java

- Rename the class `ActiveMQMessageSenderRunner.java` to `ActiveMQMessageSender.java` and place the following code.

#### [BootJmsProj01ProducerAppApplication.java](#)

```
package com.sahu.sender;

import java.util.Date;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class ActiveMQMessageSender {

    @Autowired
    private JmsTemplate template;

    @Scheduled(cron = "*/10 * * * *")
    public void sendMessage() {
        template.send("testmq1", ses-> ses.createTextMessage("From
sender at : "+new Date()));
        System.out.println("Message Sent");
    }
}

```

**Note:** While developing Sender and Receiver apps placing @EnableJms on the top of main class is optional.

## Developing JMS Pub-Sub Model app using ActiveMQ

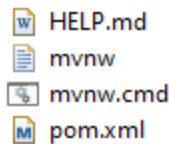
- All are same in both Sender/ Publisher app and also, in Receiver/ Subscriber app but change spring.jms.pub-sub-domain property value to true.

### Directory Structure of BootJMSProj02-PublisherApp-PUB-SUB:

```

v BootJMSProj02-PublisherApp-PUB-SUB [boot]
  v src/main/java
    v com.sahu
      > BootJmsProj01ProducerAppApplication.java
    v com.sahu.publisher
      > ActiveMQMessagePublisher.java
  v src/main/resources
    application.properties
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Maven Dependencies
  > src
  > target

```



- Develop the above directory structure using Spring Starter project option and choose packaging as Jar and create the package, class also.
- Use the following starter during project creation.  
`X Spring for Apache ActiveMQ 5`
- Then place the following code with in their respective files.

### application.properties

```
#MOM Connective details
#8161 for admin console and 61616 actual MOM service
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Enable PTP communication
#true enables Pub-sub model (default) and false enables PTP model
spring.jms.pub-sub-domain=true
```

### ActiveMQMessagePublisher.java

```
package com.sahu.publisher;

import java.util.Date;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class ActiveMQMessagePublisher implements CommandLineRunner
{

    @Autowired
    private JmsTemplate template;

    @Override
```

```

    public void run(String... args) throws Exception {
        template.send("topic1", ses-> ses.createTextMessage("From
sender at : "+new Date()));
        System.out.println("Message Sent");
    }
}

```

### Directory Structure of BootJMSProj02-SubscriberApp-PUB-SUB:

```

v BootJMSProj02-SubscriberApp-PUB-SUB [boot]
v src/main/java
  v com.sahu
    > BootJmsProj01ReceiverAppApplication.java
  v com.sahu.subscriber
    > JMSMessageSubscriber.java
v src/main/resources
  application.properties
> src/test/java
> JRE System Library [JavaSE-11]
> Maven Dependencies
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml

```

- Develop the above directory structure using Spring Starter project option and choose packaging as Jar and create the package, class also.
- Use the following starter during project creation.  
X Spring for Apache ActiveMQ 5
- Then place the following code with in their respective files.

### application.properties

```

#MOM Connective details
#8161 for admin console and 61616 actual MOM service
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Enable PTP communication
#true enables Pub-sub model (default) and false enables PTP model
spring.jms.pub-sub-domain=true

```

### JMSMessageSubscriber.java

```
package com.sahu.subscriber;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class JMSMessageSubscriber {

    @JmsListener(destination = "topic1")
    public void readMessage(String text) {
        System.out.println("Received message is : "+text);
    }

}
```

#### Order of execution:

- Run subscriber app for multiple times (more than 1 time) (To give the feel more subscribers)
- Run publisher app.
- Check the console windows of subscriber's application.

#### Admin page of Active MQ go to Topic tab:

Name ↑	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
topic1	2	2	2	Send To Active Subscribers Active Producers Delete

#### Key points on Pub-Sub model:

- a. One message published by publisher can be consumed by more than one subscriber
- b. The subscribers must do their subscription before publisher publishes the message.
- c. The message published by publisher will be duplicated and will be delivered to multiple subscribers.
- d. The Publisher and Subscribers all must be in active mode at a time.
- e. In this model the Destination name is Topic.
- f. Once the subscriber consumes the message. The messages from Topic destination of MOM s/w will not be deleted.

Q. Can we send java object data over the network?

**Ans.** Yes, by making the object as Serializable object.

**Note:** Serializing object means the converting object data into stream of bits - bytes that are required to send data over the network or to write to destination file.

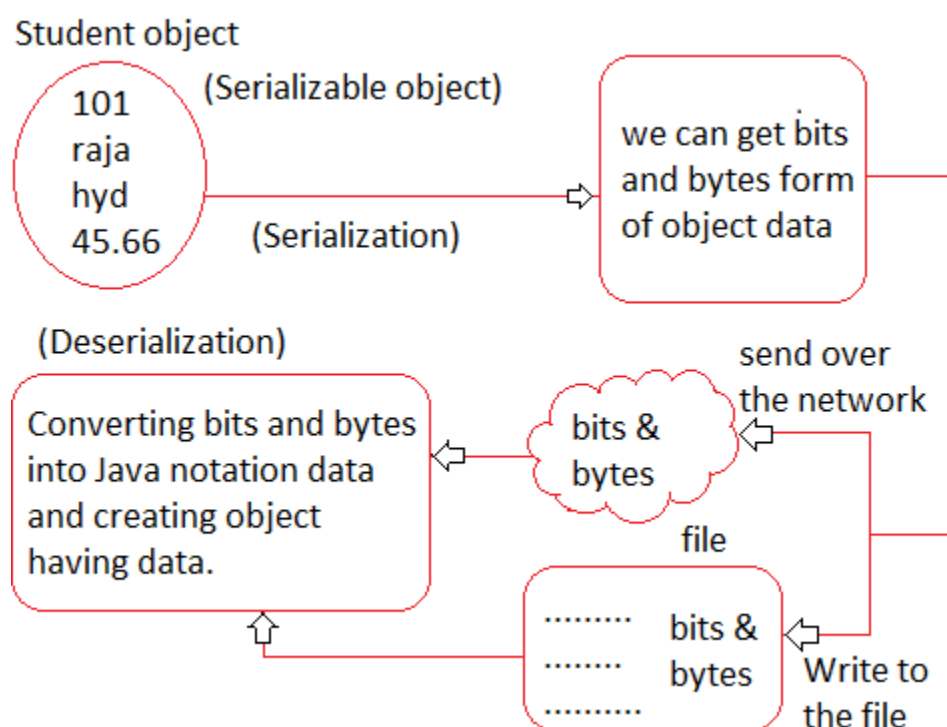
**Q. How does normal object become Serializable object?**

**Ans.** By making the class of the object as the Serializable class (class must implement `java.io.Serializable (I)`).

**Q. `java.io.Serializable (I)` is empty interface (marker interface) then how does it makes its implementation class object as Serializable object?**

**Ans.** Markers Interface does nothing directly, by seeing the marker interface implementation the underlying JVM/ Server/ Framework/ Container provides special runtime capabilities to implementation class objects they create.

- If JVM is create object for normal class that object data is in Java format and cannot be converted to stream of bits and bytes i.e., we can't send/ write normal object over the network or to a file.
- If JVM is creating object for Serializable class (class implementing `java.io.Serializable (I)`) then the JVM provides capability to the object to convert its data as stream of bits and bytes when needed to send the object's data over the network or write to a file.





- Most of marker interfaces are empty interfaces but we cannot say every empty interface is marker interface.
- We can develop our own custom marker interface but we need to create custom container to provide runtime capabilities to the implementation class objects of custom marker interface.

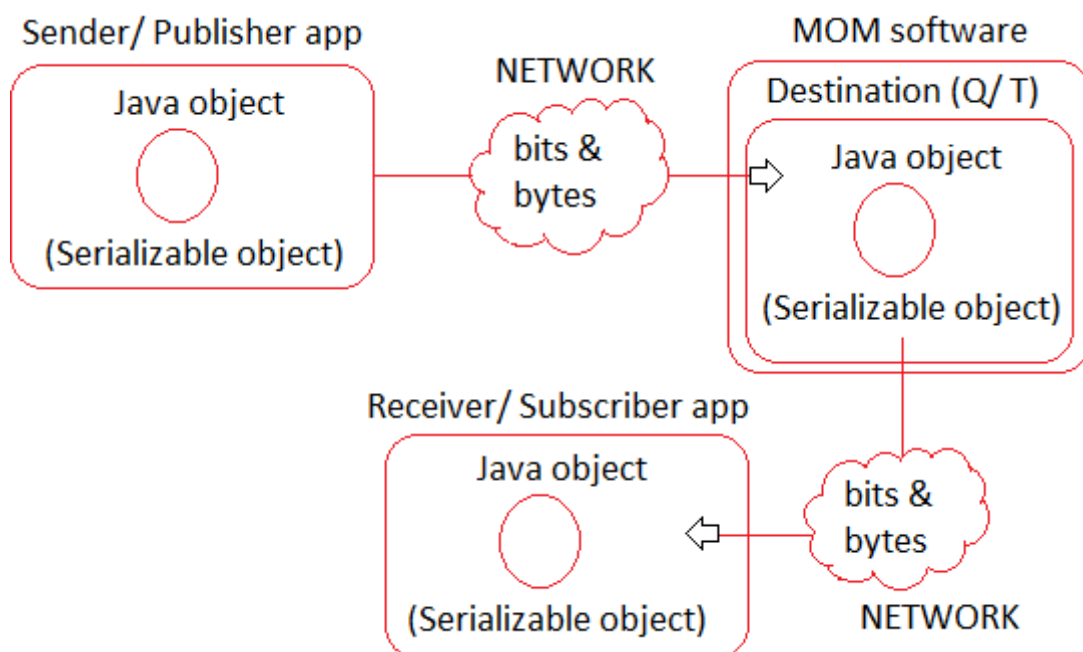
Other marker interfaces are:

- java.io.Serializable (I)
- java.lang.Cloneable (I)
- java.rmi.Remote (I)
- Repository (I) of Spring Data JPA
- java.lang.Runnable (not empty)
- and etc.

**Note:** To decide whether interface is marker or not do not take emptiness as the criteria take whether the underlying JVM/ container/ server/ Framework providing special runtime capabilities to the implementation class object or not.

### Developing ActiveMQ PTP application to send object as message

- ✚ Since Sender and Receiver apps can be there in two different machines of a network or there is possibility of running MOM software on different machine so we need to take the object as Serializable object, Sender app/ Publisher app.



## Sender app

Step 1: Create Spring Boot project adding starter dependencies.

- X Lombok
- X Spring for Apache ActiveMQ 5

Step 2: Develop the Java bean class as the Model class.

Step 3: Develop the sender app by enabling @Scheduling on the top of main class and a class for sending message with @Scheduled time.

Step 4: Add properties in application.properties file.

Step 5: Make sure ActiveMQ software (MOM software) is in running mode  
D:\JAVA\MOM\apache-activemq-5.16.5\bin\win64\activemq.bat file

Step 6: Run the Send app.

## Receiver app

Step 1: Create Spring Boot project adding starter dependencies.

- X Lombok
- X Spring for Apache ActiveMQ 5

Step 2: Develop the Java bean class as the Model class.

Step 3: Add properties in application.properties file.

Step 4: Develop the Receiver app as JMS Listener.

Step 5: Run the Send app.

## Directory Structure of BootJMSProj03-SendingObject-SenderApp:

- ▼ BootJMSProj03-SendingObject-SenderApp [boot]
  - ▼ src/main/java
    - ▼ com.sahu
      - > BootJmsProj03SendingObjectSenderAppApplication.java
    - ▼ com.sahu.model
      - > ActorInfo.java
    - ▼ com.sahu.sender
      - > ObjectMessageSender.java
  - ▼ src/main/resources
    - application.properties
  - > src/test/java
  - > JRE System Library [JavaSE-11]
  - > Maven Dependencies

```

> src
> target
HELP.md
mvnw
mvnw.cmd
pom.xml

```

- Develop the above directory structure using Spring Starter project option and choose packaging as Jar and create the package, class also.
- Use the following starter during project creation.  
X Lombok  
X Spring for Apache ActiveMQ 5
- Then place the following code with in their respective files.

### ActorInfo.java

```

package com.sahu.model;

import java.io.Serializable;

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

@Data
@NoArgsConstructor
@AllArgsConstructor
public class ActorInfo implements Serializable {
    private Integer actorId;
    private String actorName;
    private String actorAddress;
}

```

### BootJmsProj03SendingObjectSenderAppApplication.java

```

@SpringBootApplication
@EnableScheduling
public class BootJmsProj03SendingObjectSenderAppApplication {
    public static void main(String[] args) {
        SpringApplication.run(BootJmsProj03SendingObjectSenderAppApplication.class, args);
    }
}

```

### ObjectMessageSender.java

```
package com.sahu.sender;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import com.sahu.model.ActorInfo;

@Component
public class ObjectMessageSender {

    @Autowired
    private JmsTemplate jmsTemplate;

    @Scheduled(cron = "0/20 * * * * *")
    public void sendObjectDataAsMessage() {
        //Prepare object
        ActorInfo actor = new ActorInfo(1001, "Ranveer", "Mumbai");
        //Send object as Message
        jmsTemplate.convertAndSend("obj_mq1", actor);
        System.out.println("Object is send as message");
    }
}
```

### application.properties

```
#MOM Connective details
#8161 for admin console and 61616 actual MOM service
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Enable PTP communication
#true enables Pub-sub model (default) and false enables PTP model
spring.jms.pub-sub-domain=false

#Make all packages as the trusted packages (especially model class package)
#To send/ receive model class object data as message to MOM software
```

```
spring.activemq.packages.trust-all=true
#(or)
#spring.activemq.packages.trusted=com.sahu.model
```

### Directory Structure of BootJMSProj03-ReceivingObject-ReceiverApp:

```
▼ BootJMSProj03-ReceivingObject-ReceiverApp [boot]
  ▼ src/main/java
    ▼ com.sahu
      > BootJmsProj03ReceivingObjectReceiverAppApplication.java
    ▼ com.sahu.model
      > ActorInfo.java
    ▼ com.sahu.receiver
      > ObjectMessageReceiver.java
  ▼ src/main/resources
    application.properties
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Maven Dependencies
  > src
  > target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option and choose packaging as Jar and create the package, class also.
- Use the following starter during project creation.
  - X Lombok
  - X Spring for Apache ActiveMQ 5
- Copy ActorInfo.java and application.properties file from previous project.
- Then place the following code with in their respective files.

### ObjectMessageReceiver.java

```
package com.sahu.receiver;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import com.sahu.model.ActorInfo;

@Component
public class ObjectMessageReceiver {
```

```

    @JmsListener(destination = "obj_mq1")
    public void consumeObjectDataAsMessage(ActorInfo actor) {
        System.out.println("Received object : "+actor);
    }
}

```

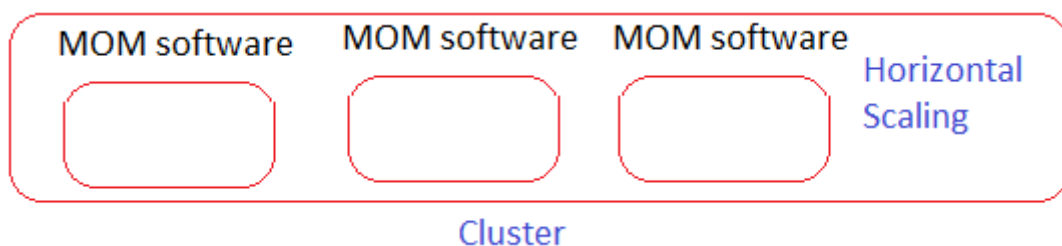
Run the application according to above steps.

**Note:** spring.activemq.packages.trust-all=true, If we do not place this entry in application.properties file then there is a possibility of getting the following error,

This class is not trusted to be serialized as ObjectMessage payload. Please take a look at <https://activemq.apache.org/objectmessage.html> for more information on how to configure trusted classes.

### Limitation of JMS

- JMS is Java language dependent technology i.e.; we need to develop both sender and receiver app in same Java language, (JMS can't be used outside of Java domain).
- JMS based MOM softwares can receive and send messages only by using protocol TCP i.e., we can't use other than protocol TCP like HTTP, SMTP and etc.
- There is a possibility of losing data/ message if the MOM software is down or MOM software is not responding while publisher or sender is sending the messages.
- If the Message is very big/ large scale then MOM software behaves very slow (i.e., gives the performance issue).
- There is no ability of creating multiple instances of single MOM software if multiple senders/ publishers are sending messages simultaneously then performance of single copy MOM software may not suitable for industry needs. (It indirectly says JMS style MOM software does not horizontal scaling, it supports only vertical scaling)



MOM software



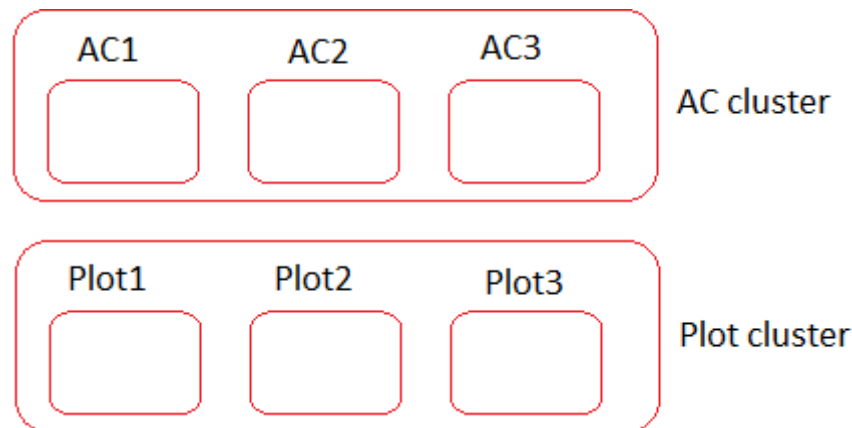
Vertical Scaling

(Increase its capacity by adding more RAMs, CPUs, HDD to computer)

**Conclusion:** Use JMS style messaging only when number of messages are less and number of senders/ publishers are less towards sending messages.

## Apache Kafka

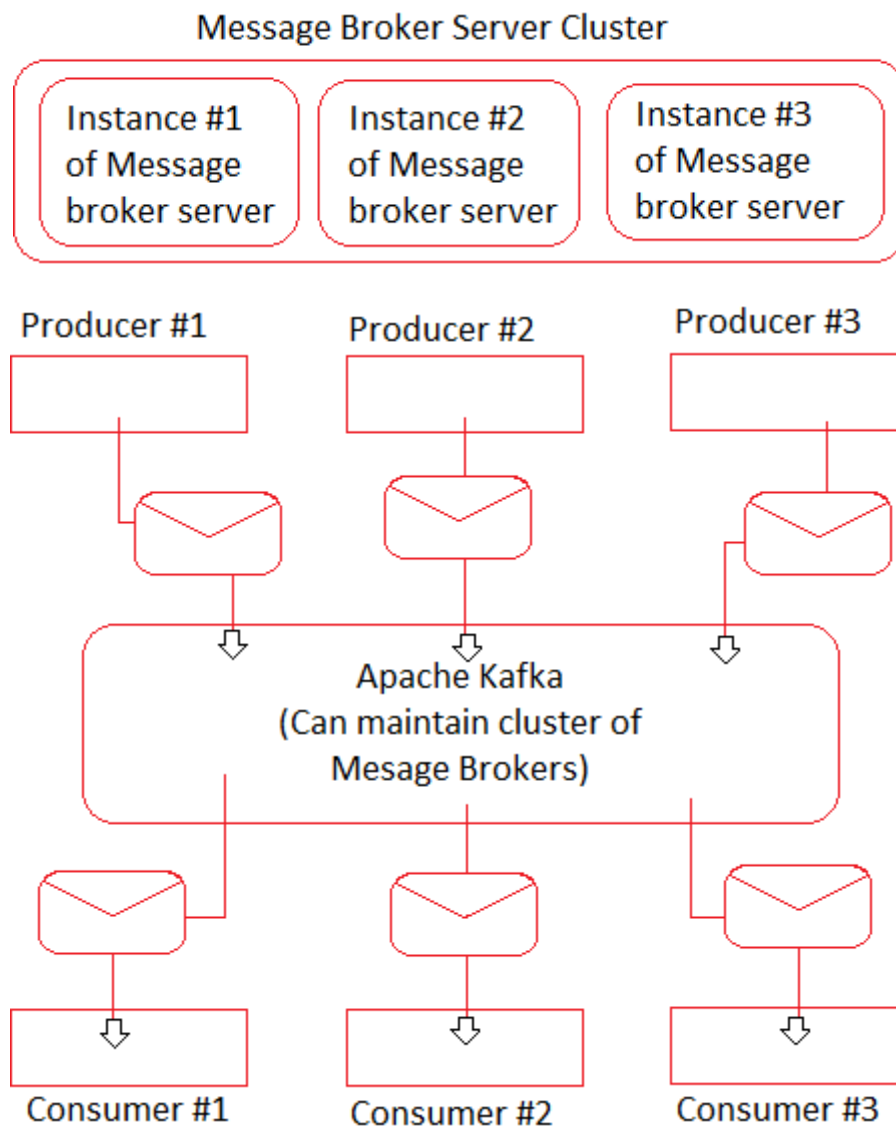
- ✚ It is Java based messaging technique which can be implemented in different languages.
- ✚ It supports to use different technologies and concepts for messaging activity like Hadoop, Spark, Scala and etc.
- ✚ Kafka integration is possible with multiple technologies of Java and non-java environment.
- ✚ zApache Kafka is all about Store + Process + Integrate (Transfer).
- ✚ Kafka is basically used to transfer (integrate) data/ messages between multiple complex apps/ systems using Cluster design.
- ✚ Cluster: set of similar items is called cluster.



- ✚ Kafka integration/ interaction with non-java applications is possible with the support of REST calls.
- ✚ Kafka is protocol independent we can write code to send messages among the applications using HTTP, TCP, FTP and etc. protocols.
- ✚ Kafka allows to take multiple message broker s/w (MOM s/w) at a time as cluster having support for Horizontal scaling.

Horizontal scaling giving the following advantages:

- Fast data transfer for largest data set/ messages.
- No data loss even one message broker s/w or MOM s/w is down because other broker s/w or MOM s/w takes care of messages.
- It takes the support of Apache Zookeeper to handle load balancing among the multiple instances of message broker s/w or MOM s/w.
- Apache Zookeeper is like Netflix Eureka server.



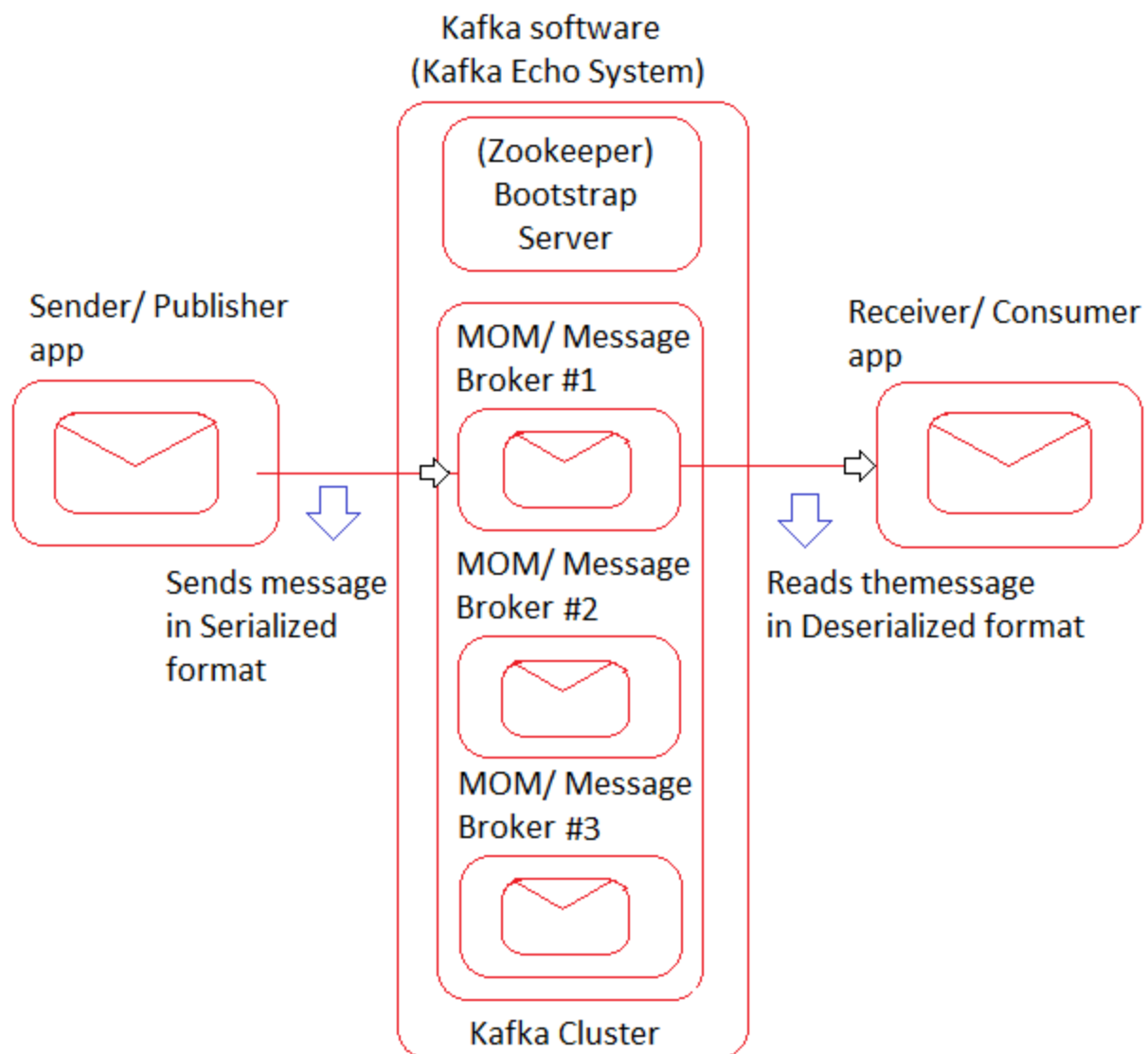
### Kafka Message Broker/ Broker server/ MOM Server

- It behaves like MOM/ Broker to receive messages/ data from sender, to hold them as needed and to deliver to Consumer.
- When Kafka s/w is started one instance of Message broker will be created automatically and it can be increased as needed.
- Collection message broker instances together is called Kafka message broker cluster and we can add any number of instances in one cluster



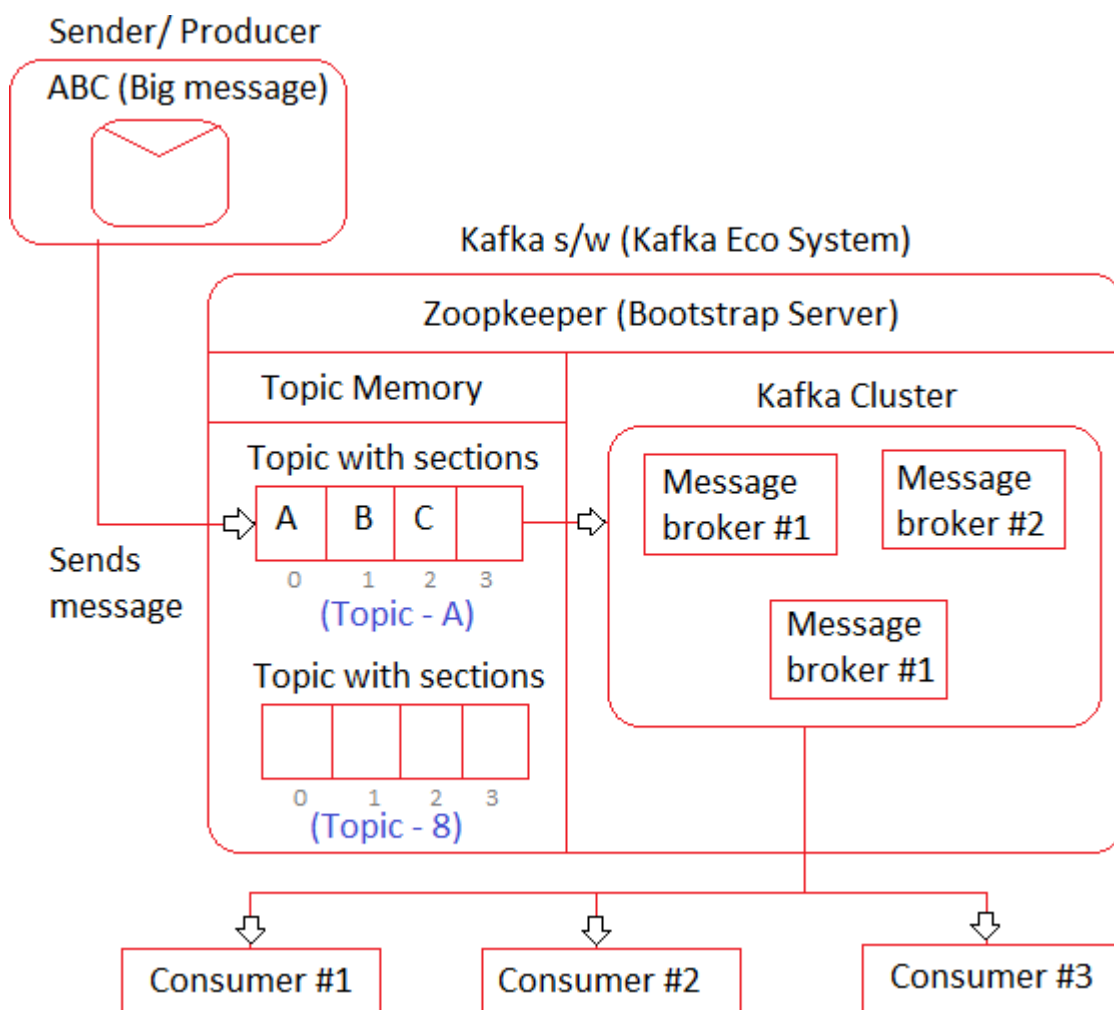
i.e., there is no limit to add instances.

- Apache Zookeeper is responsible to manage message broker instances of cluster by applying Load Balance support and it is also called Bootstrap server because it is responsible to create cluster having single instance and increasing the instances as needed.
- Apache Kafka Eco System = (Zookeeper) Bootstrap server + Message broker cluster.
- The Apache Kafka message broker gets message from sender/ publisher app in serialized format and delivers to consumer/ subscriber in deserialized format.



- Kafka does not support PTP model messaging i.e., the message broker/ MOM can't have Queue as the Destination.
- Kafka supports only Pub-Sub mode messaging i.e., the message broker/ MOM can have only Topic as the destination.

- To send message only to one consumer we take the support of topic destination.
- For one message one topic destination (memory) is created/ allocated, that stores given data/ message in partitions (huge message will be divided into parts/ packets).
- Every partition of the message is identified with an index like 0, 1, 2, 3.... these indexes are technically called as offsets.
- In Kafka messaging, for one consumer one message broker instance will be allocated at a time. So, to send 1 message to 5 consumers we need 1 Sender, 1 message, 1 Topic with sections/ partitions, 5 message broker instances, 5 consumers.
- At a time, the message broker can read one partition data, takes the data, replicates data (cloned data) and sends to consumer i.e., each large message/ data will be stored in multiple partitions of topic and the message broker reads data from all partitions and sends to its respective consumer.



Topic Sections = Topic with partitions

- Each message broker instance is capable of reading message from multiple sections/ partitions of Topic destinations, replicates the message parts -- merges the message parts -- sends the message to respective consumer.

## Apache Kafka Terminologies

**Message Broker/ MOM:** The mediator/ middle man that will transfer message to consumer by reading message/ data from Topic/ Topic partitions.

**Kafka Cluster:** Collection/ group of message broker instances which are created based on consumers count is called Kafka cluster.

**Topic:** The memory that holds message sent by the Producer in parts/ sections.

**Producer:** The app/ component that sends the message to Topic memory in Serialized format.

**Consumer:** The app/ component that reads the message from Topic memory through Message broker in Deserialized format.

**Offset/ Index:** The id or index given to partition message/ data in Topic.

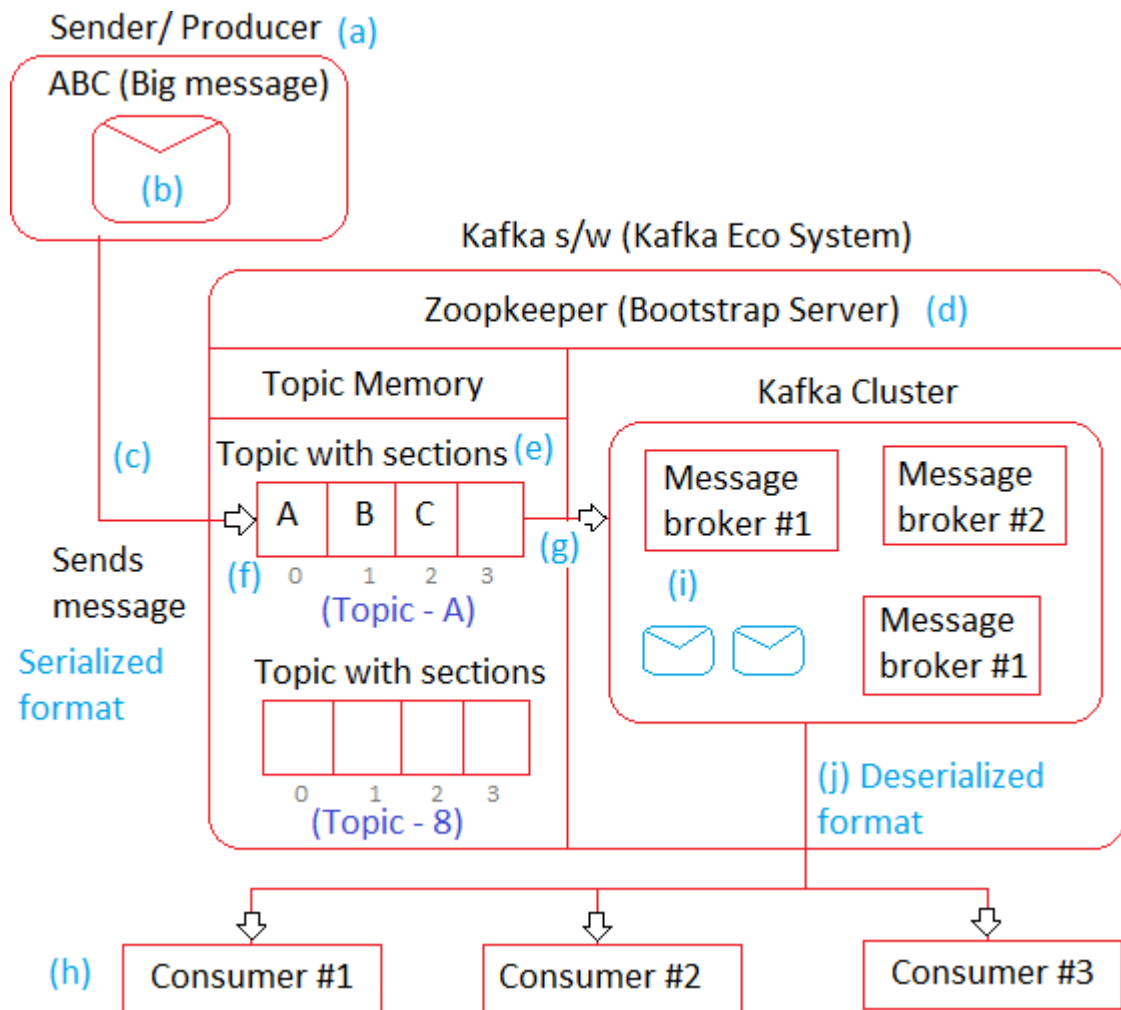
**Zookeeper/ Bootstrap Server:** Handles Message broker cluster and Topic memory, while managing the Message broker cluster it will do Load Balancing.

**Replica-Factor:** The number of cloned/ duplicate messages that should be created in order to send the messages to Consumers through message brokers, generally it is decided based consumer count. If there are 10 consumers then Replica-factor is 1/10 (for 1 message 10 cloned copies are required).

## Apache Kafka Work Flow Diagram

- a. Define one Producer app.
- b. Create either static message or dynamic message that can be given at run time.
- c. Send message from Producer in the form of key-value pair format here "key" is topic name (with the given name topic is available then uses it, otherwise new topic will be created), "value" is the message/ data.

**Note:** Producer app sends this data/ message in Serialized format.



- d. Start Zookeeper (Bootstrap server).
- e. Topic section will be created in Topic memory to read the message sent by the Producer app.

**Note:** New Topic section will be created (if not already available) otherwise it will use the existing Topic section.

- f. Topic section reads the message and stores the message in different partitions of Topic which will be created as needed.

**Note:** These partitions are identified with indexes/ offsets.

- g. Creates link between Topic section/ Topic and Message broker.
- h. Define one or more Consumer apps as needed by specifying topic name.

**Note:**

- ✓ Based on the given Topic name in consumer apps the link

between Topic, message broker and Consumer app will be created as needed.

- ✓ The Zookeeper takes the responsibility of creating message broker instances as cluster based on the Consumers count and takes care of LB -Load Balancing.
- i. The Message broker(s) reads the data/ messages from Topic partitions and creates cloned/ replicate copies of these partition messages.
- j. The message broker(s) sends the cloned partition messages to one or more consumer(s) part by part.

## Arranging Apache Kafka Software

**Step 1:** Go to this [\[Link\]](#).

Then Click on the Scala Binary downloads then click the suggestion link, now your download will start then Extract the ZIP file to a particular location.

### 3.1.1

- Released May 13, 2022
- [Release Notes](#)
- Source download: [kafka-3.1.1-src.tgz \(asc, sha512\)](#)
- Binary downloads:
  - Scala 2.12 - [kafka\\_2.12-3.1.1.tgz \(asc, sha512\)](#)
  - Scala 2.13 - [kafka\\_2.13-3.1.1.tgz \(asc, sha512\)](#)

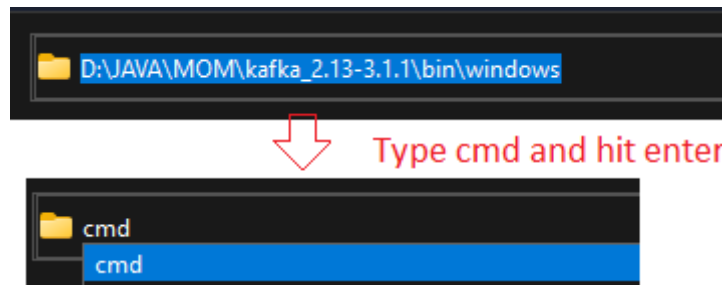


We suggest the following site for your download:

[https://dlcdn.apache.org/kafka/3.1.1/kafka\\_2.13-3.1.1.tgz](https://dlcdn.apache.org/kafka/3.1.1/kafka_2.13-3.1.1.tgz)

**Step 2:** To start the Zookeeper server, go to your <Kafka Home>\bin\windows

D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows and open CMD (command prompt there).



Type cmd and hit enter

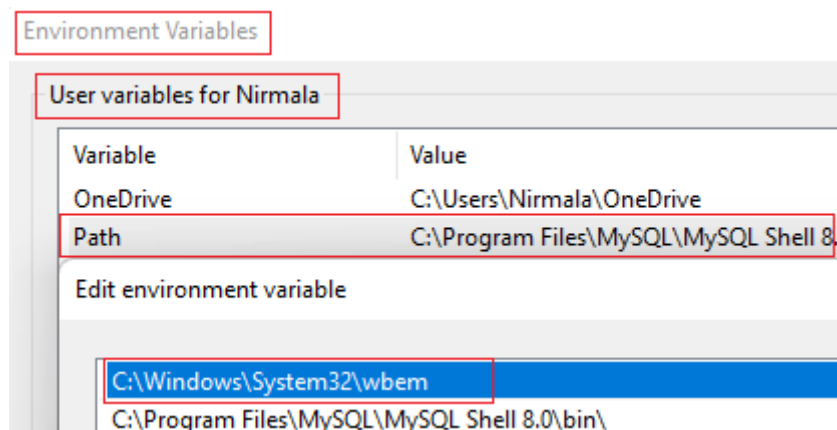
- Run the below command in command prompt.

`D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> zookeeper-server-start.bat D:\JAVA\MOM\kafka_2.13-3.1.1\config\zookeeper.properties`

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>zookeeper-server-start.bat
D:\JAVA\MOM\kafka_2.13-3.1.1\config\zookeeper.properties
```

### Step 3: Start Apache Kafka setup.

- Make sure that “C:\Windows\System32\wbem” should added to PATH environment variable of User or System variable other wise to add Go to the This PC – Properties – Advanced system settings – Environment variables... – User variables – click New... if PATH variable is not available otherwise choose path and click on Edit... – Then give the value “C:\Windows\System32\wbem” then click on OK – OK – OK.



- Then open a new command prompt in same location `D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows` and run the below command.

`D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-server-start D:\JAVA\MOM\kafka_2.13-3.1.1\config\server.properties`

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-server-start D:\JAVA\MOM\kafka_2.13-3.1.1\config\server.properties
```

**Step 4:** Create new topic specifying replication-factor (number of copies), partitions count (offset count). Open a new command prompt in same location D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows and run the below command.

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic sahu-tpc
```

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic sahu-tpc
Created topic sahu-tpc.
```

**Step 5:** Create a producer and link with message broker with the support of bootstrap-server. Open a new command prompt in same location D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows and run the below command.

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-console-producer --bootstrap-server localhost:9092 --topic sahu-tpc hello
```

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-console-producer --bootstrap-server localhost:9092 --topic sahu-tpc
```

**Step 6:** Create a consumer and link with message broker. Open a new command prompt in same location D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows and run the below command.

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-console-consumer --bootstrap-server localhost:9092 --topic sahu-tpc
```

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic sahu-tpc
```

After that give message from Producer and check in consumer

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-console-producer --bootstrap-server localhost:9092 --topic sahu-tpc hello
>hello
>hii
>are you redy to work
>_
```

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-console-consumer.bat
--bootstrap-server localhost:9092 --topic sahu-tpc
hello
hii
are you redy to work
```

## Developing Kafka Producer as Kafka client in Legacy style

- Both producer/ sender and consumer/ receiver apps that are taking Kafka software are called Kafka client apps.

### Producer/ Sender App Development (Legacy Style - Spring style)

- First, we need to establish a link between Producer app and Kafka setup (mainly bootstrap server) by using the following details/ properties

```
bootstrap-server = localhost:9092
key-serializer = pkg.StringSerializer
value-serializer = pkg.StringSerializer
```

(Kafka Bootstrap server default port is 9092) we generally keep this info in Properties class object if we develop application in legacy style (Non-Spring Boot Style).

**Note:** Any type of data/ message given by Producer app will be converted to String message/ data using StringSerializer class that is specified above. If we give Java object as message/ data then it will be converted into JSON String content.

- Take the support of "ProducerRecord" class to specify message object details and topic name.

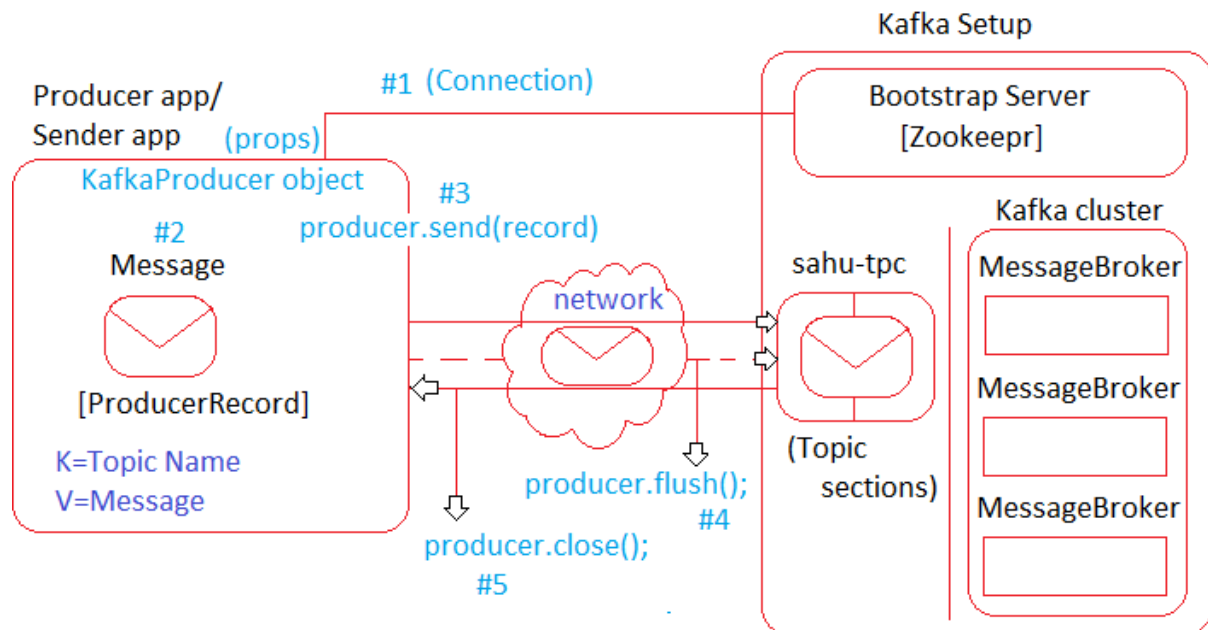
```
ProducerRecord<String, String> record = new ProducerRecord
    ("topic name", message object);
(ProducerRecord class object indirectly representing message/ data to
send from producer app to topic of Kafka setup).
```

- To send message (ProducerRecord) created in the Producer app we need to use KafkaProducer<String, String> class as shown below

```
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
producer.send(record); //puts data in connection stream in queue (line)
```



```
producer.flush(); //sends to data Topic based on the given topic name
producer.close(); // Closes the link b/w producer and Kafka setup
```



## Code development and execution

**Step 1:** Create maven project (Non-Spring Boot starter project) using maven-archetype-quickstart. For that

- Click on File menu, choose Maven project – then click on Next – Select an Archetype “maven-archetype-quickstart” – Next – Fill the below details then Finish.

Group Id:	com.sahu
Artifact Id:	KafkaProj01-Producer
Version:	0.0.1-SNAPSHOT
Package:	com.sahu.producer

- Add the following dependencies from mvnrepository.com
  - [Apache Kafka](#)
  - [SLF4J Simple Binding](#)
  - [Jackson Dataformat XML](#)

**Step 2:** Develop the producer application.

**Step 3:** Execute things in the following order

- Start zookeeper

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> zookeeper-server-  
start.bat D:\JAVA\MOM\kafka_2.13-3.1.1\config\zookeeper.properties
```

b. Start Kafka setup

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-server-start  
D:\JAVA\MOM\kafka_2.13-3.1.1\config\server.properties
```

c. Create topic

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-topics --create --  
bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --  
topic sahu-tpc-may
```

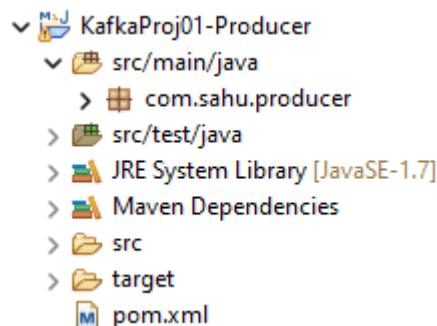
d. Start the consumer app

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows> kafka-console-consumer  
--bootstrap-server localhost:9092 --topic sahu-tpc-may
```

e. Run the Producer application – Message Sent.

```
D:\JAVA\MOM\kafka_2.13-3.1.1\bin\windows>kafka-console-consumer  
--bootstrap-server localhost:9092 --topic sahu-tpc-may  
Welcome to Apache Kafkas Message
```

### Directory Structure of BootJMSProj03-ReceivingObject-ReceiverApp:



- Develop the above directory structure using Maven project option and create the package, class also.
- Add the following dependencies in pom.xml from mvnrepository.com.
  - [Apache Kafka](#)
  - [SLF4J Simple Provider](#)
  - [Jackson Dataformat XML](#)
- If you want to change the JDK version then you do it in pom.xml file.
- Then place the following code with in their respective files.
- Run the application by following above execution order.

### MessageProducer.java

```
package com.sahu.producer;

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

public class MessageProducer {

    public static void main(String[] args) {
        //Create connection properties as K=V in java.util.Properties
        class object
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

        //Create KafkaProducer object
        KafkaProducer<String, String> producer = new
KafkaProducer<String, String>(props);
        //Create ProducerRecord representing the message
        String message = "Welcome to Apache Kafkas Message";
        String topicName = "sahu-tpc-may";
        ProducerRecord<String, String> record = new
ProducerRecord<String, String>(topicName, message);
        //Send message (record)
        producer.send(record);
        //flush the message
        producer.flush();
        //close the connection with bootstrap server
        producer.close();
        System.out.println("Message sent.");
    }
}
```

## Developing Kafka consumer as Kafka client in Legacy style

- ✚ We need to create communication link b/w Kafka setup and consumer(s) by supplying multiple details connection details as key-value pairs in the form of Properties object.

bootstrap-server = localhost:9092

key-deserializer = StringDeserializer (given by Kafka API)

value-deserializer = StringDeserializer

groupid = grp-listeners [anything can be given here] [optional]

- ✚ If multiple consumers are taken reading same message from topic through message broker, then grouping multiple consumers to single group by providing group id, makes replicate/ cloning operation on messages faster.
- ✚ Create KafkaConsumer class object specifying the above connection properties to get link/ connection between consumer and Kafka setup (Indirectly with Bootstrap server (zookeeper) that manages Topic memories and message brokers).

```
KafkaConsumer<String, String> consumer = new  
KafkaConsumer<>(props);
```

- ✚ Props is java.util.Properties class obj having the above connection properties.
- ✚ Link Consumer with MessageBroker by specifying the topic name (Here MessageBroker will be created dynamically on 1 per Consumer basis)

```
consumer.subscribe(Arrays.asList("sahu-tpc-may")); // we given multiple  
topic names to read messages from multiple topic sections.
```

- ✚ Do polling (pinging the message broker continuously having certain time gap) with Message broker to check message(s) are available or not and read message.

### Expected process:

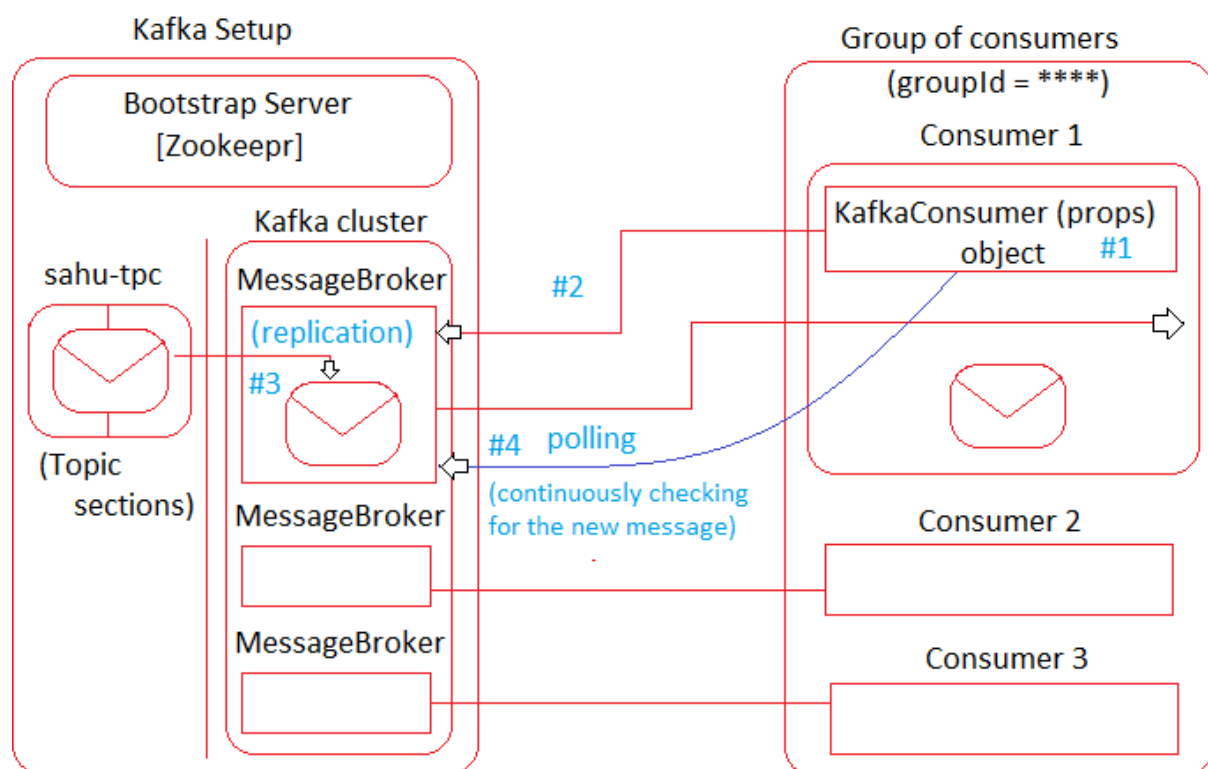
- On arrival of message to topic memory, the Message broker reads the messages and sends the message to all the subscribed consumers automatically.
- But in legacy style Kafka consumer development that is not happening. So, polling has to be done.

### Actual process:

- Consumer pings the Message Broker having scheduling (having certain continuous time gap) for new message, if new message came to topic memory, then the message broker gives that message to consumer (this process is called polling).

```
while(true) {  
    ConsumerRecords<String, String> records=  
        consumer.poll(Duration.ofMillis(1000));  
    for (ConsumerRecord<String, String> record: records) {  
        System.out.println("message is: "+record.value());  
    }  
}
```

- Each consumed message will be represented by ConsumerRecord object multiple consumed messages will be represented by ConsumerRecord object.



### Code and execution

**Step 1:** Create maven project adding the following dependencies

- [Apache Kafka](#)
- [SLF4J Simple Binding](#)

- [Jackson Dataformat XML](#)

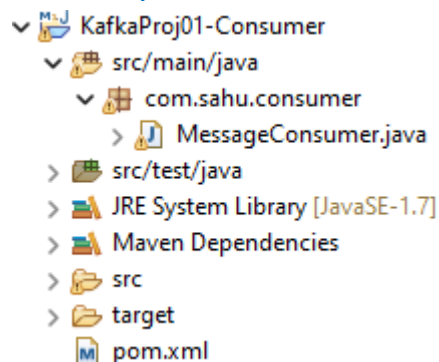
Step 2: Develop the consumer application.

Step 3: Run The services in the following order

- Start zookeeper  
D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows> [zookeeper-server-start.bat D:\JAVA\MOM\kafka\\_2.13-3.1.1\config\zookeeper.properties](#)
- Start Kafka setup  
D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows> [kafka-server-start D:\JAVA\MOM\kafka\\_2.13-3.1.1\config\server.properties](#)
- Since topic name “sahu-tpc-may” is already created so we need not to create again.
- Run the consumer application using Eclipse
- Run the Producer application using Eclipse (Previous application).  
Consumer console:

Message is : Welcome to Apache Kafkas Message

Directory Structure of BootJMSProj03-ReceivingObject-ReceiverApp:



- Develop the above directory structure using Maven project option and create the package, class also.
- Add the following dependencies in pom.xml from mvnrepository.com.
  - [Apache Kafka](#)
  - [SLF4J Simple Binding](#)
  - [Jackson Dataformat XML](#)
- If you want to change the JDK version then you do it in pom.xml file.
- Then place the following code with in their respective files.

- Run the application by following above execution order.

### MessageConsumer.java

```
package com.sahu.consumer;

import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

public class MessageConsumer {

    public static void main(String[] args) {
        // Create connection properties as K=V in java.util.Properties
class object
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
"grp1_consumer");

        // Create KafkaProducer object
        KafkaConsumer<String, String> consumer = new
KafkaConsumer<String, String>(props);
        // Subscribe to topic destination through Message broker
        consumer.subscribe(Arrays.asList("sahu-tpc-may"));
        //Perform polling to check and read the message
        while (true) {
            //Polling and get consumer records (message)
```

```

        ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(2000));
        //Read and display messages
        for(ConsumerRecord<String, String> record : records) {
            System.out.println("Message is : "+record.value());
        }
    }
}

```

## Spring Boot + Apache Kafka API

- Spring Boot gives built-in support for Apache Kafka. It even given certain objects automatically through AutoConfiguration process.
- If we add spring-boot-starter-apache-kafka dependency to application then we get KafkaTemplate<K, V> class object through autoconfiguration which internally takes care of creating KafkaProducer, ProducerRecord objects that are required to send messages/ data.

### In Producer class or component

@Autowired

private KafkaTemplate<String, String> template;

- we can place @KafkaListener (topicName="....", groupId="....") annotation on the method Listener class to make Message broker to collect the message received to topic section i.e., it internally takes care of creating KafkaConsumer<String, String> object and ConsumerRecord object.

### In Listener class

@KafkaListener (topicName="sahu-tpc-fri", groupId="sahu-grp1")

public void readMessage(String msg){

.....

.....//logic to read the message.

}

- We need to add @EnableKafka on the top of starter class (main class).
- We can get message from end users as request parameter through RestController/ Microservice of Spring MVC/ Spring Rest app, so we



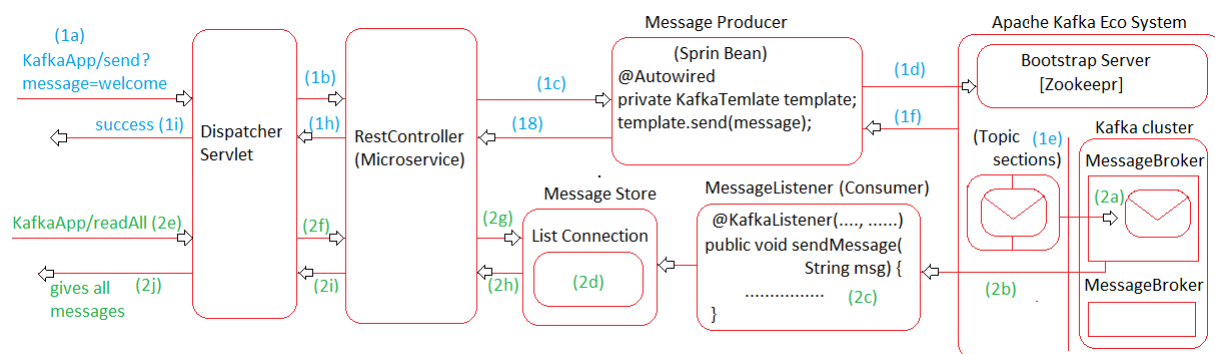
need to make the Producer app taking request through RestController to send Kafka setup.

- Similarly, we need to read the message from Kafka setup by using Consumer or MessageListener to send end users through RestController.

<http://localhost:4041/KafkaApp/send?message=welcome>

<http://localhost:4041/KafkaApp/send?message=quotation value 2000>

<http://localhost:4041/KafkaApp/send?message=how are u>



Full and clear diagram [\[Link\]](#)

- (1a) – (1i) – 1<sup>st</sup> request response.
- (2a) – (2j) – 2<sup>nd</sup> request response.

## Order of development

- MessageProducer having injection KafkaTemplate object.
- RestController with handler method with "/send" request path having injection of MessageProducer object.
- MessageStore.
- MessageListener injected with MessageStore.
- Above RestController with another handler method with "/readAll" request path having injection of MessageStore object.

**Step 1:** Create Spring Boot starter project by adding the following starter dependencies

```

X Spring Boot DevTools
X Lombok
X Spring for Apache Kafka
X Spring Web
  
```

**Step 2:** Add @Enablekafka on the top of main class/ starter class.

**Step 3:** Add the following properties in application.properties.

#Server configuration

```
server.port=4041
#context path
spring.mvc.servlet.path=/RestKafkaApp
#Topic name
app.topic.name=sahu-tpc-fri
#Producer properties
spring.kafka.producer.bootstrap-servers=localhost:9092
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-
serializer=org.apache.kafka.common.serialization.StringSerializer
#Consumer properties
spring.kafka.consumer.bootstrap-servers=localhost:9092
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

**Step 4:** Create producer class as MessageProducer.java.

**Step 5:** Create RestController having handler methods.

**Step 6:** Create store helper class as MessageStore.java.

**Step 7:** Create consumer class as MessageConsumer.java.

#### Execution Order:

- a. Start zookeeper  
D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows> [zookeeper-server-start.bat D:\JAVA\MOM\kafka\\_2.13-3.1.1\config\zookeeper.properties](#)
- b. Start Kafka setup  
D:\JAVA\MOM\kafka\_2.13-3.1.1\bin\windows> [kafka-server-start D:\JAVA\MOM\kafka\\_2.13-3.1.1\config\server.properties](#)

**Note:** No need of creating topic separately, @EnableKakfa will take care of creating topic dynamically.

- c. Run the application as Spring Boot app or on server and give requests  
<http://localhost:4041/RestKafkaApp/send?message=hello>

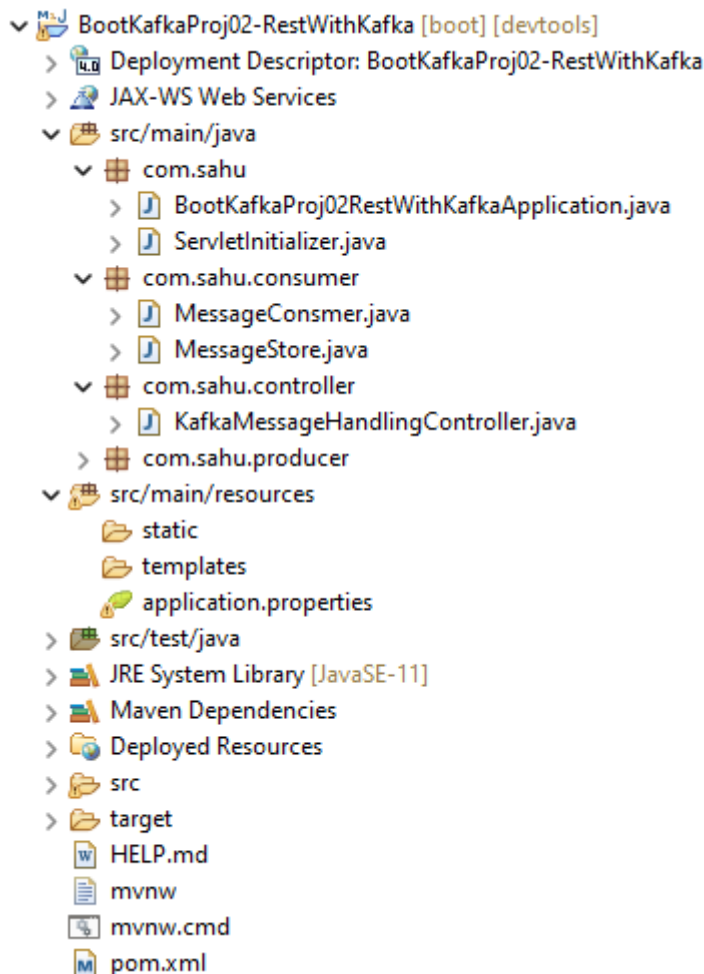
<http://localhost:4041/RestKafkaApp/send?message=Welcome to Kafka>  
<http://localhost:4041/RestKafkaApp/send?message=Raja>

<http://localhost:4041/RestkafkaApp/readAll>

Output:

[hello, Welcome to Kafka, Raja]

### Directory Structure of BootJMSProj03-ReceivingObject-ReceiverApp:



- Develop the above directory structure using Spring Starter project option and choose packaging type as war and create the packages, classes also.
- Use the following starter during project creation.
  - X Spring Boot DevTools
  - X Lombok
  - X Spring for Apache Kafka
  - X Spring Web
- Then place the following code with in their respective files.
- Execute by following above order.

### BootKafkaProj02RestWithKafkaApplication.java

```
@SpringBootApplication
@EnableKafka
public class BootKafkaProj02RestWithKafkaApplication {

    public static void main(String[] args) {

        SpringApplication.run(BootKafkaProj02RestWithKafkaApplication.class, args);
    }

}
```

### application.properties

```
#Server configuration
server.port=4041

#context path
spring.mvc.servlet.path=/RestKafkaApp

#Topic name
#Not a Spring property, custom property
app.topic.name=sahu-tpc-fri

#Producer properties
spring.kafka.producer.bootstrap-servers=localhost:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

#Consumer properties
spring.kafka.consumer.bootstrap-servers=localhost:9092
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

### MessageProducer.java

```
package com.sahu.producer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component("msgProducer")
public class MessageProducer {

    @Autowired
    private KafkaTemplate<String, String> template;

    @Value("${app.topic.name}")
    private String topicName;

    public String sendMessage(String message) {
        template.send(topicName, message);
        return "Message delivered";
    }
}
```

### MessageConsmer.java

```
package com.sahu.consumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
public class MessageConsmer {

    @Autowired
    private MessageStore messageStore;

    @KafkaListener(topics = "${app.topic.name}", groupId="grp1")
    public void readMessage(String message) {
        //add messages to store
    }
}
```

```
        //add messages to store
        messageStore.addMessage(message);
    }
}
```

### MessageStore.java

```
package com.sahu.consumer;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Component;

@Component
public class MessageStore {

    private List<String> listMessage = new ArrayList<>();

    public void addMessage(String message) {
        listMessage.add(message);
    }

    public String getAllMessage() {
        return listMessage.toString();
    }

}
```

### KafkaMessageHandlingController.java

```
package com.sahu.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.sahu.consumer.MessageStore;
import com.sahu.producer.MessageProducer;
```

```
@RestController
public class KafkaMessageHandlingController {

    @Autowired
    private MessageProducer producer;

    @Autowired
    private MessageStore messageStore;

    @GetMapping("/send")
    public String sendMessage(@RequestParam("message") String
message) {
        String status = producer.sendMessage(message);
        return "<h1>"+status+"<h1>";
    }

    @GetMapping("/readAll")
    public String fetachAllMessage() {
        return "<h1>"+messageStore.getAllMessage()+"</h1>";
    }

}
```

----- The END -----