receiver to simplify the setup.

1. **Go to Spring Initializr:** Open your web browser and navigate to https://start.spring.io/.
2. **Configure Your Project:**
   - **Project:** Maven Project
   - **Language:** Java
   - **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   - **Group:** com.example.jms
   - **Artifact:** spring-jms-demo
   - **Name:** spring-jms-demo
   - **Description:** Spring JMS Demo with ActiveMQ
   - **Package Name:** com.example.jms
   - **Packaging:** Jar
   - **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   - **Spring Web:** For a simple REST controller to trigger message sending.
   - **Spring for Apache ActiveMQ 5:** This is the crucial dependency for JMS integration with ActiveMQ.
   - **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).

STEP 3: Configure JMS Connection in application.properties

We need to tell Spring Boot how to connect to our ActiveMQ broker.

1. **Open application.properties:**
   - Navigate to src/main/resources/ and open application.properties.
2. **Add ActiveMQ Connection Properties:**
   - Add the following lines:

```
# application.properties
# ActiveMQ broker URL (default for ActiveMQ)
spring.activemq.broker-url=tcp://localhost:61616

# Enable JMS listener auto-startup
spring.jms.listener.auto-startup=true

# Enable sending messages to non-existent destinations (for demo simplicity)
spring.jms.template.explicit-qos-enabled=true
spring.jms.template.delivery-persistent=true
```

- tcp://localhost:61616 is the default TCP port for ActiveMQ.
- spring.jms.listener.auto-startup=true ensures our message listeners start automatically.

STEP 4: Create a Message Sender (Producer)

We'll create a service that uses JmsTemplate to send messages to a queue.

- Create a new package com.example.jms.service in src/main/java/.
- Create a Java class named MessageSender.java inside this package:

```
// src/main/java/com/example/jms/service/MessageSender.java
package com.example.jms.service;


import org.springframework.jms.core.JmsTemplate; // Core class for sending JMS
messages
import org.springframework.stereotype.Service;


/**
 * Service responsible for sending messages to JMS destinations.
 */
@Service
public class MessageSender {


    private final JmsTemplate jmsTemplate;


    // Spring automatically injects JmsTemplate
    public MessageSender(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    /**
     * Sends a simple text message to a specified queue.
     * @param destination The name of the queue.
     * @param message The text content of the message.
     */
    public void sendMessage(String destination, String message) {
        System.out.println("Sending message to queue '" + destination + "': " +
message);
        jmsTemplate.convertAndSend(destination, message); // convertAndSend
handles serialization
    }

    /**
     * Sends a message and waits for a reply (Request-Reply Messaging).
     * @param requestDestination The queue to send the request to.
     * @param replyDestination The queue to expect the reply from.
     * @param requestMessage The request message content.
     * @return The reply message content.
     */
    public String sendAndReceive(String requestDestination, String replyDestination,
String requestMessage) {
        System.out.println("Sending request to '" + requestDestination + "': " +
requestMessage);
        System.out.println("Expecting reply on '" + replyDestination + "'");

        // Send the message and specify the reply-to destination
        // The convertAndSend method with MessagePostProcessor allows setting
JMS headers
```

```
        jmsTemplate.convertAndSend(requestDestination, requestMessage, message
    -> {

    message.setJMSReplyTo(jmsTemplate.getConnectionFactory().createContext().cre
    ateQueue(replyDestination));
            return message;
        });

        // Receive the reply message from the reply destination
        // receiveAndConvert blocks until a message is available or timeout occurs
        Object reply = jmsTemplate.receiveAndConvert(replyDestination);

        if (reply != null) {
            System.out.println("Received reply from '" + replyDestination + "': " + reply);
            return reply.toString();
        } else {
            System.out.println("No reply received from '" + replyDestination + "'.");
            return "No reply";
        }
    }
}
```

STEP 5: Create a Message Receiver (Consumer)

We'll create a component that listens for messages from a queue.

- Create a new package com.example.jms.consumer in src/main/java/.
- Create a Java class named MessageReceiver.java inside this package:

```
// src/main/java/com/example/jms/consumer/MessageReceiver.java
package com.example.jms.consumer;

import org.springframework.jms.annotation.JmsListener; // Annotation for message-
driven POJOs
import org.springframework.stereotype.Component;
```

```
/**
 * Component responsible for receiving messages from JMS destinations.
 */
@Component
public class MessageReceiver {

   /**
    * Listens for messages on the "myQueue" destination.
    * This method acts as a Message-Driven POJO (MDP).
    * @param message The received text message.
    */
   @JmsListener(destination = "myQueue") // Listens to "myQueue"
   public void receiveMessage(String message) {
      System.out.println("Received message from 'myQueue': " + message);
   }

   /**
    * Listens for request messages on "requestQueue" and sends a reply to the
JMSReplyTo destination.
    * This demonstrates the Request-Reply Messaging pattern.
    * @param requestMessage The received request message.
    * @return The reply message.
    */
   @JmsListener(destination = "requestQueue") // Listens to "requestQueue"
   public String handleRequest(String requestMessage) {
      System.out.println("Received request from 'requestQueue': " +
requestMessage);
      String replyMessage = "Reply to: " + requestMessage.toUpperCase();
      System.out.println("Sending reply: " + replyMessage);
      return replyMessage; // Spring automatically sends this as a reply to
JMSReplyTo
   }
}
```

STEP 6: Create a REST Controller to Trigger Messaging

We'll create a simple REST controller to easily trigger sending messages via HTTP requests.

- Create a new package com.example.jms.controller in src/main/java/.
- Create a Java class named JmsController.java inside this package:

```
// src/main/java/com/example/jms/controller/JmsController.java
package com.example.jms.controller;

import com.example.jms.service.MessageSender;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/**
 * REST Controller to trigger JMS message sending.
 */
@RestController
public class JmsController {

    private final MessageSender messageSender;

    public JmsController(MessageSender messageSender) {
        this.messageSender = messageSender;
    }

    /**
     * Endpoint to send a message to "myQueue".
     * Example: GET http://localhost:8080/send?message=HelloJMS
     * @param message The message content to send.
     * @return A confirmation message.
     */
    @GetMapping("/send")
    public ResponseEntity<String> sendMessage(@RequestParam String message)
{
        messageSender.sendMessage("myQueue", message);
        return ResponseEntity.ok("Message sent to myQueue: " + message);
    }

    /**
     * Endpoint to send a request message to "requestQueue" and receive a reply
from "replyQueue".
     * Example: GET http://localhost:8080/request-reply?message=Ping
     * @param message The request message content.
     * @return The reply received from the consumer.
     */
    @GetMapping("/request-reply")
    public ResponseEntity<String> sendRequestAndReceiveReply(@RequestParam
String message) {
        String reply = messageSender.sendAndReceive("requestQueue",
"replyQueue", message);
        return ResponseEntity.ok("Request sent, Reply received: " + reply);
    }
}
```

STEP 7: Main Spring Boot Application Class

This is the standard Spring Boot application entry point. We need to enable JMS listening.

- Open your main application class (e.g., SpringJmsDemoApplication.java in com.example.jms.app).

```
// src/main/java/com/example/jms/app/SpringJmsDemoApplication.java
package com.example.jms;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jms.annotation.EnableJms; // Enables JMS listener
annotation processing

/**
 * Main Spring Boot application class for JMS demo.
 * @EnableJms: Activates detection of @JmsListener annotations.
 */
@SpringBootApplication
@EnableJms // This annotation is crucial for @JmsListener to work
public class SpringJmsDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringJmsDemoApplication.class, args);
    }
}
```

STEP 8: Run the Application and Test JMS Messaging

1. **Ensure ActiveMQ is Running:**
   - Double-check that your Apache ActiveMQ broker is running (from STEP 1). You

should be able to access its web console at http://localhost:8161/.

2. **Run the Spring Boot Application:**
   - Open your main application class (SpringJmsDemoApplication.java).
   - Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
   - You should see Spring Boot starting on port 8080.
3. **Test Sending and Receiving Messages:**
   - **Open your IDE's console for the spring-jms-demo application.** This is where you'll see both sent and received messages.
   - **Test Sending Messages (Point-to-Point):**
     - Open your web browser and go to: http://localhost:8080/send?message=HelloFromWeb
     - **Observe:**
       - In your application's console, you should see:
         ```
         Sending message to queue 'myQueue': HelloFromWeb
         Received message from 'myQueue': HelloFromWeb
         ```
       - In the ActiveMQ web console (http://localhost:8161/ -> Queues tab), you should see myQueue appear (if it didn't exist before) and its message count briefly increment and then decrement (as the message is consumed).
   - **Test Request-Reply Messaging:**
     - Open your web browser and go to: http://localhost:8080/request-reply?message=RequestForReply
     - **Observe:**
       - In your application's console, you should see:
         ```
         Sending request to 'requestQueue': RequestForReply
         Expecting reply on 'replyQueue'
         Received request from 'requestQueue': RequestForReply
         Sending reply: REPLY TO: REQUESTFORREPLY
         Received reply from 'replyQueue': REPLY TO: REQUESTFORREPLY
         ```
       - In the ActiveMQ web console, you should see requestQueue and

replyQueue appear, with messages flowing through them.

You have successfully implemented messaging with Spring JMS and Apache ActiveMQ! You've learned how to:

- Set up and run a local ActiveMQ broker.
- Configure Spring Boot to connect to ActiveMQ.
- Send messages using JmsTemplate.
- Receive messages using @JmsListener (Message-Driven POJOs).
- Implement a request-reply messaging pattern.

This activity provides a practical understanding of how Spring simplifies asynchronous communication using JMS.

Activity 10.1: Spring JavaMail “Sending Emails in Spring Boot”

This activity will guide you through integrating email functionality into a Spring Boot application using Spring's JavaMail support. You will learn how to send simple text emails, rich HTML emails with attachments, and implement asynchronous email sending, along with basic error handling.

STEP 1: Project Setup (Spring Boot Application)

We'll use Spring Initializr to set up a new Spring Boot project with the necessary mail dependencies.

1. **Go to Spring Initializr:** Open your web browser and navigate to https://start.spring.io/.
2. **Configure Your Project:**
    - **Project:** Maven Project
    - **Language:** Java
    - **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
    - **Group:** com.example.mail
    - **Artifact:** spring-mail-demo
    - **Name:** spring-mail-demo
    - **Description:** Spring JavaMail Demo
    - **Package Name:** com.example.mail
    - **Packaging:** Jar
    - **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
    - **Spring Web:** For a simple REST controller to trigger email sending.
    - **Java Mail Sender (Spring Boot Starter Mail):** This is the core dependency for Spring Mail integration.
    - **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE