http://www.devglan.com/spring-boot/spring-boot-jms-activemq-example

Spring Boot Jms ActiveMQ Example

By [Dhiraj Ray](https://plus.google.com/+DhirajRay1" \t "_blank), 13 April,2017

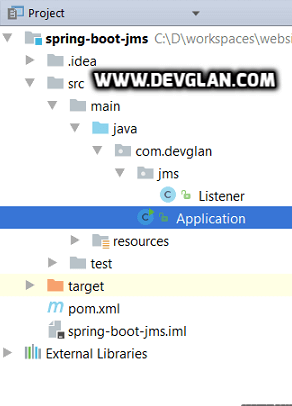
This article is about spring boot and JMS integration. Here we will be using spring boot features to implement asynchronous messaging service. We wil be using activemq as a message broker and look into different support provided by spring boot while integrating JMS with activemq. The application that we will be creating here will be a simple spring jms activemq example with a sample JMS producer and consumer.

If you are looking for spring 4 and JMS integration with activemq then follow - [Spring JMS ActiveMq Exmple](http://www.devglan.com/spring-mvc/spring-jms-activemq-integration-example)

Spring Boot ActiveMQ Support

Spring boot automatically configures connectionfactory if it detects ActiveMQ on the classpath. In this case, it also makes use of embedded broker if does not find any ActiveMQ custom configurations in application.properties. Here in this example we will have activemq configurations outside the application but we will also look into ways to integrate embedded activemq with spring boot by defining our JMS configurations in application.properties. So let us get started with defining the project structure first.

Project Structure

[](http://imgur.com/NZ5wLmH)

Maven Dependencies

spring-boot-starter-activemq : It provides all the required dependencies to integrate JMS and activemq with spring boot.

activemq-broker : This provides embedded activemq in spring boot application. But since, we will be configuring our activemq outside the application we have commented it for time being.

spring-boot-maven-plugin : It will collect all the jar files present in the classpath and create a single executable jar.

pom.xml

<groupId>com.devglan</groupId>

<artifactId>spring-boot-jms</artifactId>

<version>0.1.0-SNAPSHOT</version>

<properties> <java.version>1.8</java.version> </properties>

<parent> <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.1.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-activemq</artifactId>

</dependency>

<dependency>

<groupId>org.apache.activemq</groupId>

<artifactId>activemq-broker</artifactId>

</dependency>

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

**Other Interesting Posts** [Spring JMS ActiveMQ Example](http://www.devglan.com/spring-mvc/spring-jms-activemq-integration-example) [Spring Boot Actuator Complete Guide](http://www.devglan.com/spring-boot/spring-boot-actuator-tutorial-guide) [Spring Boot Security Hibernate Example](http://www.devglan.com/spring-security/spring-boot-security-hibernate-login-example) [Spring Mvc Anglar Js Example](http://www.devglan.com/spring-mvc/spring-mvc-angularjs-integration-example) [Working with Spring Boot Jdbc Template](http://www.devglan.com/spring-jdbc/working-with-springboot-jdbctemplate) [Spring Boot Security Password Encoding using Bcrypt Encoder](http://www.devglan.com/spring-security/spring-boot-security-password-encoding-bcrypt-encoder) [Websocket spring Boot Integration Without STOMP](http://www.devglan.com/spring-boot/spring-websocket-integration-example-without-stomp)

Configuring ActiveMQ in Spring Boot

Following is the application.properties which has configurations for activemq. Here we are configuring the tcp broker url along with the username and password required to make activemq connection

**application.properties**

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

Configuring Spring Beans for JMS(Not Required)

While integrating JMS and ActiveMQ with spring boot, no extra configuratins specific to JmsTemplate or containerFactory is required. The following configurations will be implicitly provided by spring boot for us. Hence, the following class is not available in the source. I am including these configurations just to show you guys how Spring Boot helps to get rid of boiler plate codes and configurations.

@Configuration public class JmsConfig {

String BROKER\_URL = "tcp://localhost:61616";

String BROKER\_USERNAME = "admin";

String BROKER\_PASSWORD = "admin";

@Bean public ActiveMQConnectionFactory connectionFactory(){ ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(); connectionFactory.setBrokerURL(BROKER\_URL); connectionFactory.setPassword(BROKER\_USERNAME); connectionFactory.setUserName(BROKER\_PASSWORD); return connectionFactory; }

@Bean public JmsTemplate jmsTemplate(){

JmsTemplate template = new JmsTemplate(); template.setConnectionFactory(connectionFactory());

return template; }

@Bean public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() { DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory(); factory.setConnectionFactory(connectionFactory());

factory.setConcurrency("1-1");

return factory;

} }

JMS Listener

@JmsListener:It marks a method to be the target of a JMS messagelistener on the specified destination. In our case the destination is inbound.queue.This class is responsible to listen messsage from the inbound queue and process the same.

@SendTo annotation will take care of sending the return value of receiveMessage() to the destination defined in @SendTo.

**Listener.java**

package com.devglan.jms; import com.google.gson.Gson; import org.springframework.stereotype.Component; import javax.jms.JMSException; import javax.jms.Message; import javax.jms.TextMessage; import org.springframework.jms.annotation.JmsListener; import java.util.Map; @Component public class Listener { @JmsListener(destination = "inbound.queue") @SendTo("outbound.queue") public String receiveMessage(final Message jsonMessage) throws JMSException { String messageData = null; System.out.println("Received message " + jsonMessage); String response = null; if(jsonMessage instanceof TextMessage) { TextMessage textMessage = (TextMessage)jsonMessage; messageData = textMessage.getText(); Map map = new Gson().fromJson(message, Map.class); response = "Hello " + map.get("name"); } return response; } }

Spring Bean Configuration

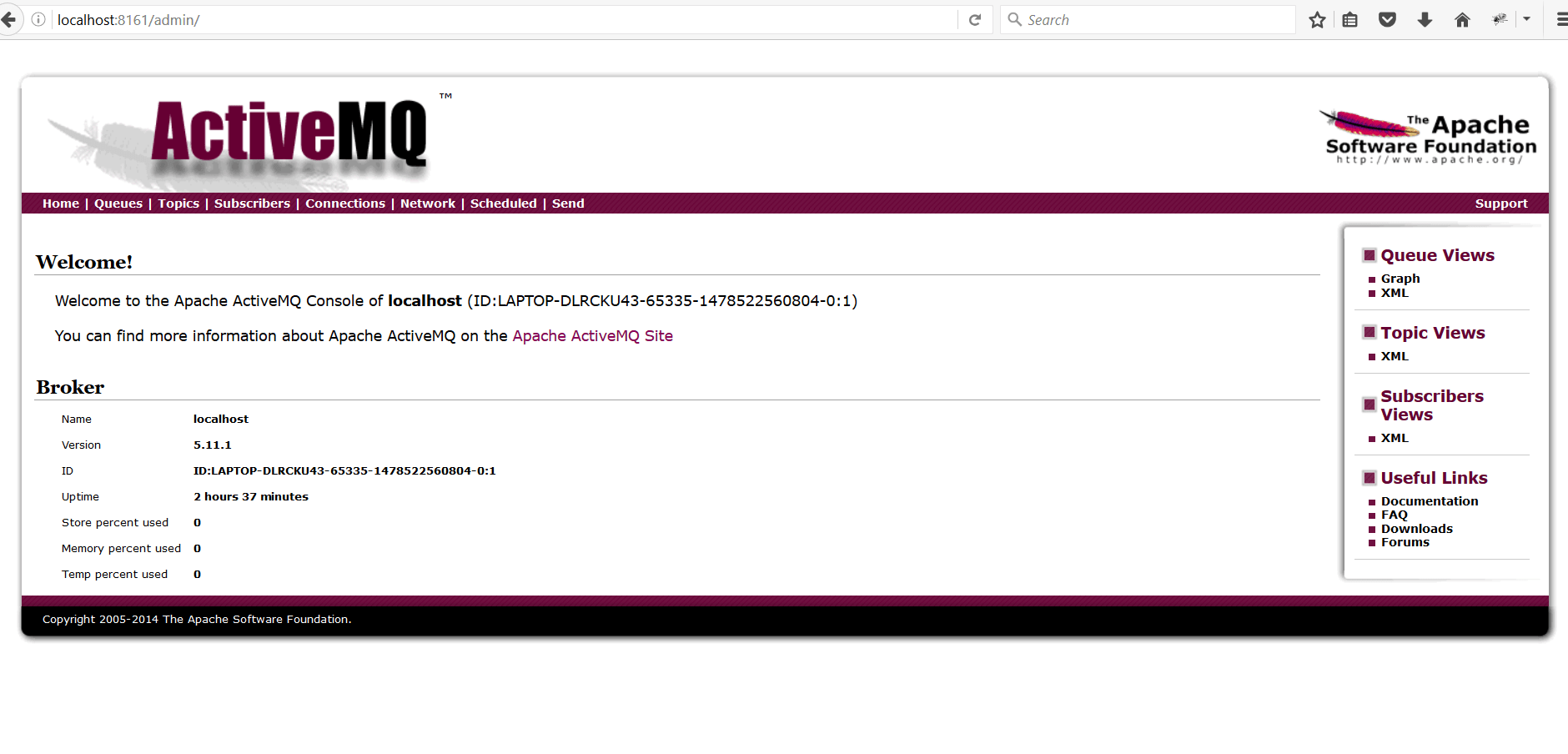
@SpringBootApplication Equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes:

**Application.java**

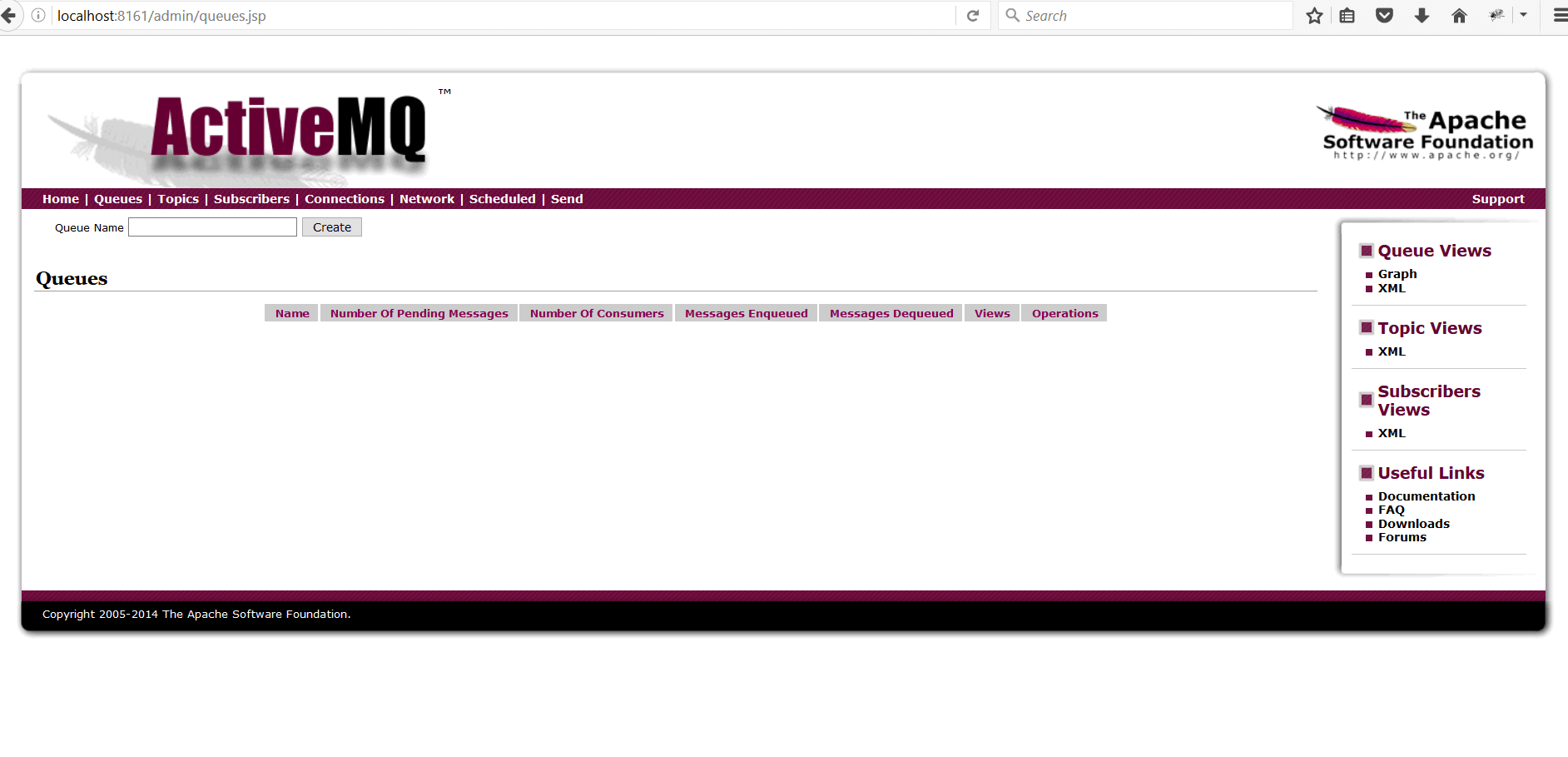
package com.devglan.tiles; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.jms.annotation.EnableJms; @SpringBootApplication @EnableJms public class Application { public static void main(String[] args) { SpringApplication.run(Application.class, args); } }

Installing Apache ActiveMq

ActiveMq by default exposes a broker url tcp://localhost:61616 and an admin console on tcp://localhost:61616 with userId and password as admin and admin. Following are the steps to download and install activemq. 1. Download apache activemq from [here](http://activemq.apache.org/activemq-5111-release.html) as per your operating system. 2. Extract under some folder. In my case it's under java\apache-activemq-5.11.1-bin. 3. Now traverse to java\apache-activemq-5.11.1-bin\bin\win64 and execute the acivemq.bat file. 4. Open the browser and hit - <http://localhost:8161/admin/> 4. Enter userId/password as admin/admin. Then following screen will appear:

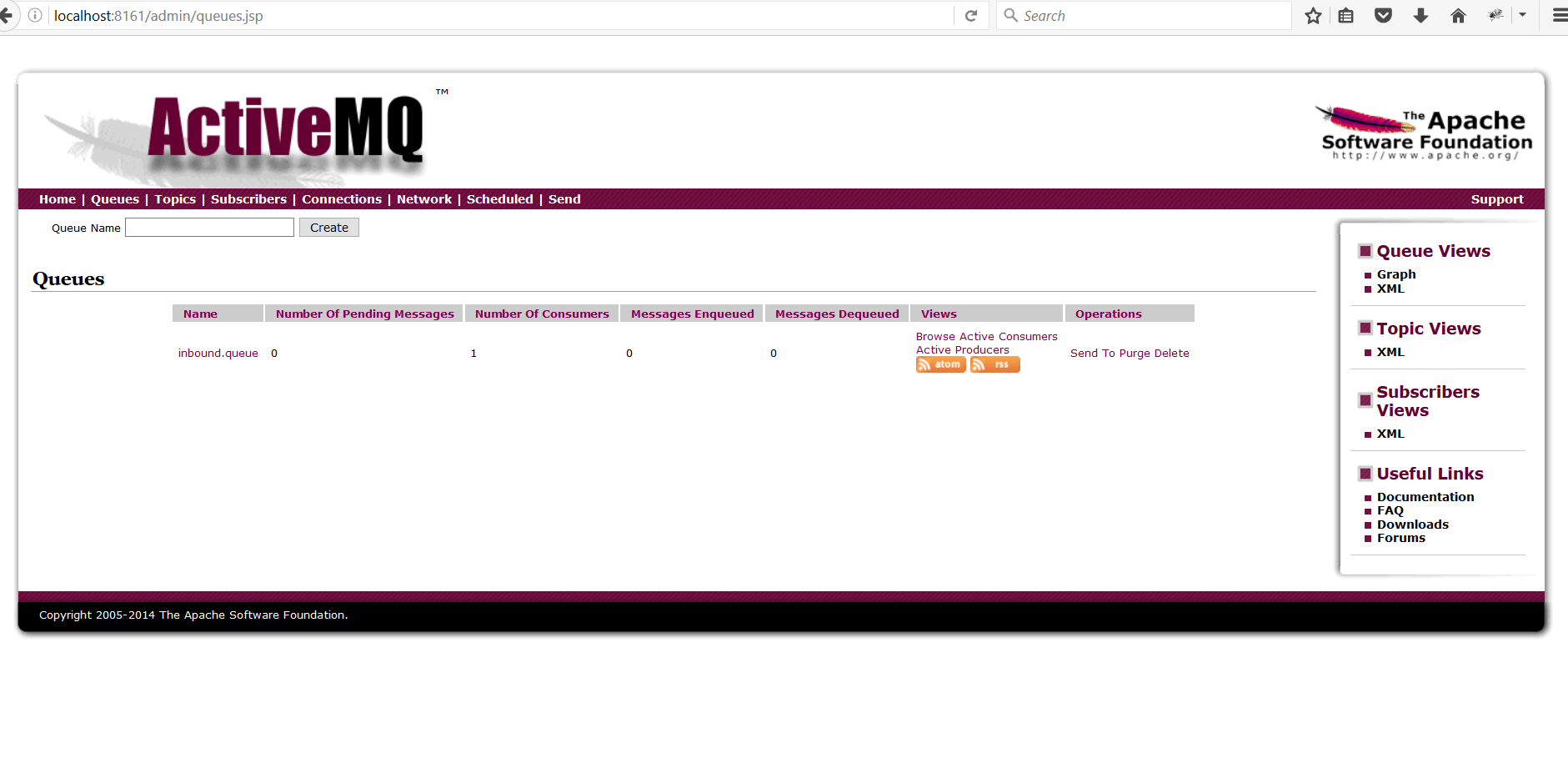


5. Now click on the queues option present in menu bar and you can notice there are no queues available as below image.

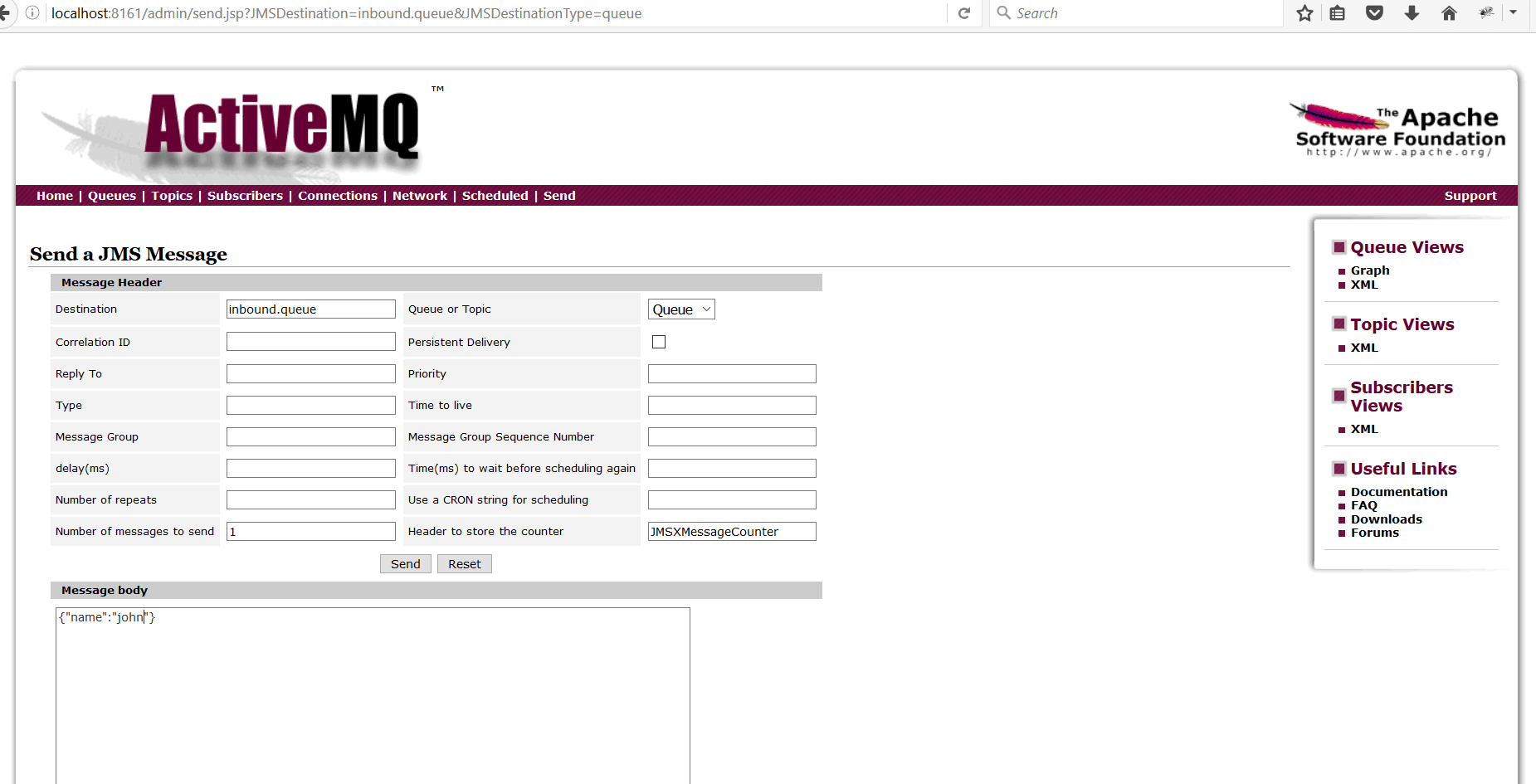


Run Application

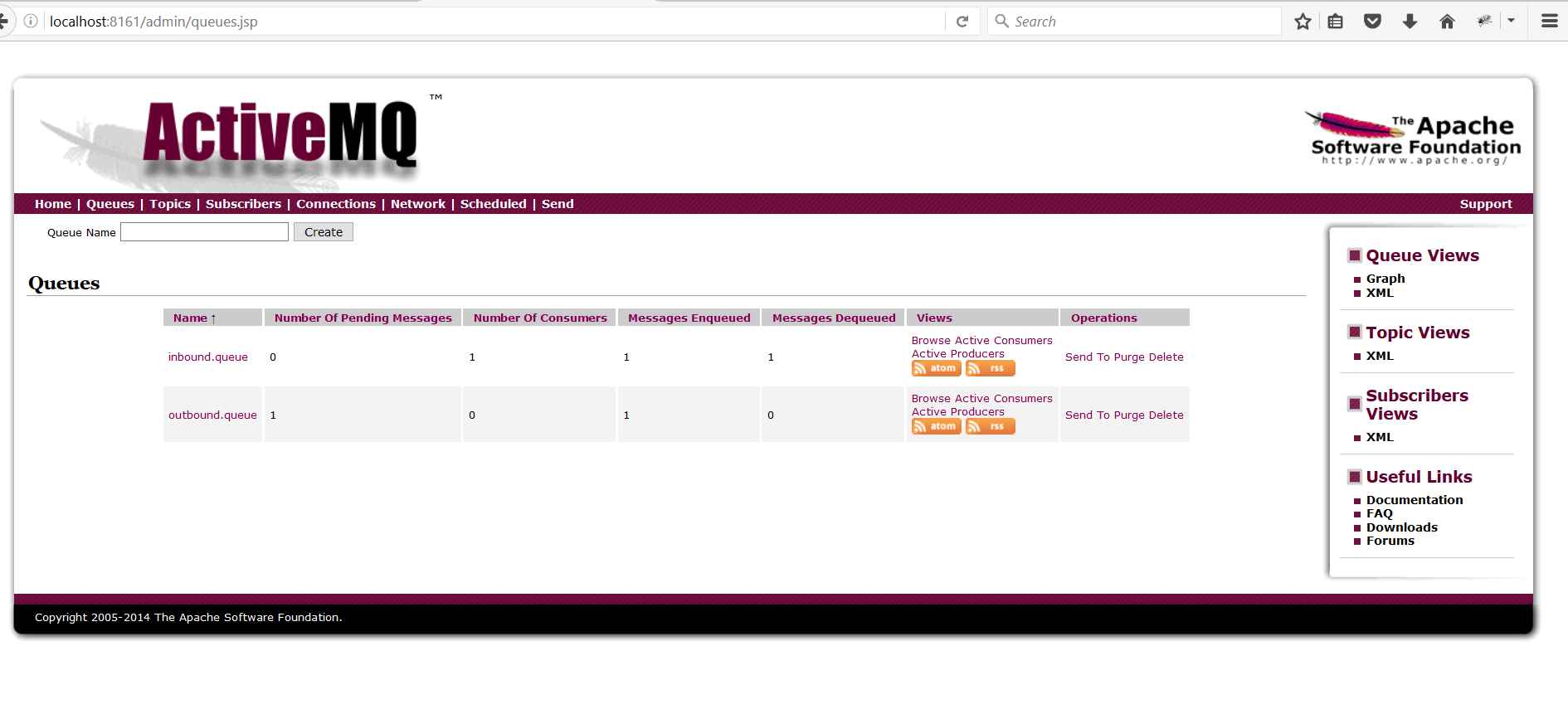
1. Run Application.java as a java applicaton and you can see one queue automatically created with name inbound.queue which we had configured in our Listener.class. Refresh the page after application restart.



2. Our application is now listening to this queue and whenever we push messsage to this queue our Listener should automatically pick this message. Now click on the send To option. Enter a JSON string as {"name":"John"} and click on send.



3. After the messsage is sent, the message is received by Listener.java and the same message will be processed and send back to outbound.queue by Producer.java. Now you can see one more queue created once you click on the queues option from the menu as shown below:



Also, you can check logs in the console.

Conclusion

This demonstrates a simple JMS integration. It is very useful in a mechanism to allow asynchronous request processing. You may wish to implement JMS because the request take a long time to complete or because several parties may be interested in the actual request. Another reason for using it is to allow multiple clients (potentially written in different languages) to access information via JMS.