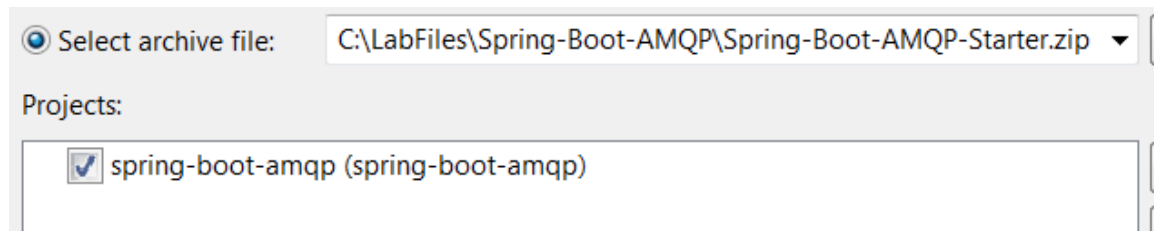


Lab 10 - Use AMQP Messaging with Spring Boot

In this lab you will complete a Spring Boot application that sends and receives messages through a RabbitMQ instance.

Part 1 - Import the Starter Project

- ___1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- ___2. Click the radio button for **Select archive file**:
- ___3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-AMQP\Spring-Boot-AMQP-Starter.zip** and then click **Open**.



- ___4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

Part 2 - Examine the Starter Project

The starter project implements a simple HTML form that accepts inputs that simulate an Order in an online-store system. Most but not all of the components are in place to send that order into an AMQP message queue. We'll fill in the rest of the components in this lab. We'll also add a component that listens on the same message queue and prints out the orders that are received.

Note: In normal application usage, we would have one application sending the messages, and a completely different application receiving the messages. We're only combining the two functions in one application here to simplify the lab setup a little.

There are a few items already setup in the starter project that you should take note of for your own projects

- 'pom.xml' contains a dependency that are particularly important to this project:

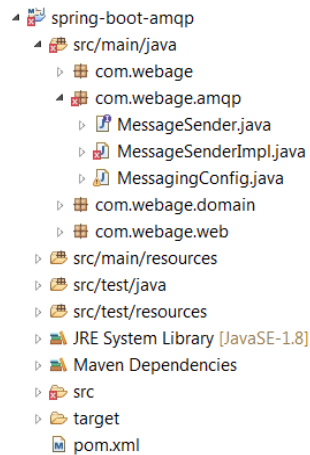
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

- This dependency pulls in the AMQP and RabbitMQ support.
- In 'src/main/resources/templates', there are HTML files that are used to display the data entry and success forms.
- The basic components of the Spring Boot application are already present. There is an 'App.java' file that contains a 'Main' method that can be used to start the system.
- The 'com.webage.amqp' package contains starter point files for the messaging implementation.
- The 'com.webage.web' package contains a request handler class that receives the form submittal and calls an implementation of 'com.webage.amqp.MessageSender' that is autowired by Spring.
- At the root of 'src/main/resources', there is a YAML file, 'application.yml' that contains the connection details (host, port, userid, user secret) for connecting to the RabbitMQ instance.

Part 3 - Complete the MessageConfig

Spring Boot handles the vast majority of the setup automatically, as soon as it finds the AMQP libraries in the classpath. However, there is one piece of setup that we need to provide: We need to have a configuration object that will enable the messaging queue to be created on-demand.

___1. In the **Project Explorer**, locate the 'com.webage.amqp' package node.



___2. Inside the 'com.webage.amqp' package, locate the 'MessagingConfig.java' file. Double-click on the file to open it. You will see that the class is empty.

___3. Add the following declaration to the body of the class:

```
@Bean
public Queue myQueue() {
    return new Queue("myqueue1");
}
```

This declaration declares a desired message queue. Spring AMQP will use this definition to create the queue the first time it's used.

___4. Save and close the file.

Part 4 - Complete the MessageSenderImpl

___1. In the 'com.webage.amqp' package, locate the 'MessageSenderImpl' class, and then double-click on the class to open it.

__2. The class should look something like this:

```
@Component
public class MessageSenderImpl implements MessageSender {

    @Autowired
    private AmqpTemplate amqpTemplate=null;

}
```

The class is requesting injection of an 'AmqpTemplate' object. This object is defined automatically by Spring Boot when we include the starter dependency.

__3. Add the following method into the body of the class:

```
@Override
public void sendOrderMessage(Order toSend) {
    System.out.println("Sending message with order - " + toSend);
    amqpTemplate.convertAndSend("myqueue1", toSend);
}
```

This is the method that the web controller component will call to actually send the message.

__4. Save the file. Bite that the error is gone.

__5. Right click **spring-boot-amqp** and select **Maven → Update Project**.

__6. Make sure **spring-boot-amqp** is selected and click OK.

__7. Right click on the **spring-boot-amqp** project and select **Run as → Maven install** to Run a Maven install and ensure that there are no errors. You may need to run the Maven Install two times to get the Build Success message.

Part 5 - Add a Message Listener

In order to simplify the lab a little bit, we're going to create a message listener in the same project as the message sender. This structure is probably not one that you would use in production, but it's not entirely out of the question, either.

__1. Create a new class called 'MyListener' in the 'com.webage.amqp' package.

__2. Edit the class so the body of it looks like:

```
@Component
public class MyListener {

    @RabbitListener(queues="myqueue1")
    public void onMessage(Order order) {
        System.out.println("Message received - Order Details:" + order);
    }
}
```

__3. Organize imports by pressing **Ctrl-Shift-O**. Select **com.webage.domain.Order**.

__4. Save the file.

The '@RabbitListener' annotation will establish this method as a listener on the designated queue. Spring Boot will setup an additional thread to monitor the queue and then call this method when a message arrives. Here, the message handler method simply prints out the received message to the console.

Part 6 - Verify Rabbit MQ is running

In this part, you will open the Windows Services panel and verify Rabbit MQ is running. This is necessary before testing the application in the next part.

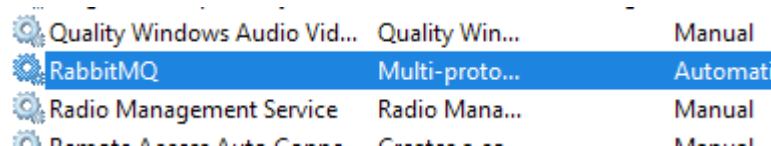
__1. Click the **Windows Start** menu in the lower left corner of your screen.

__2. Click **Administrative Tools**.

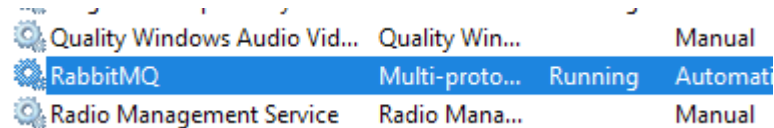
__3. In the *Administrative Tools* window, double click on **Services**.

__4. In the *Services* panel, scroll down until you see **RabbitMQ**.

__5. If the status is not **Running** (e.g., empty), then right click on **RabbitMQ** and choose **Start**.



__6. Verify **RabbitMQ** starts and the status changes to **Running**.

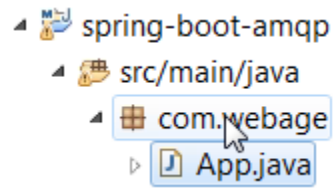


__7. Close the *Services* panel.

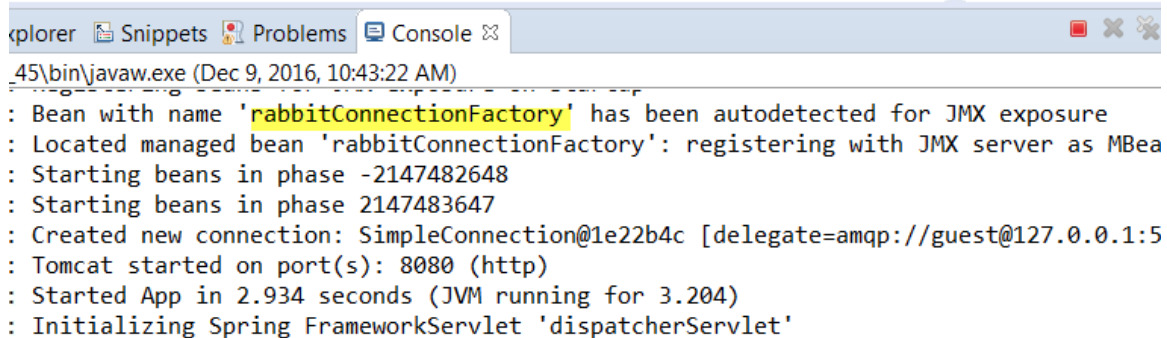
Part 7 - Test

__1. Run 'Maven Install'.

__2. Run the 'App.java' class as a Java application.



__3. Notice that rabbit has been called.

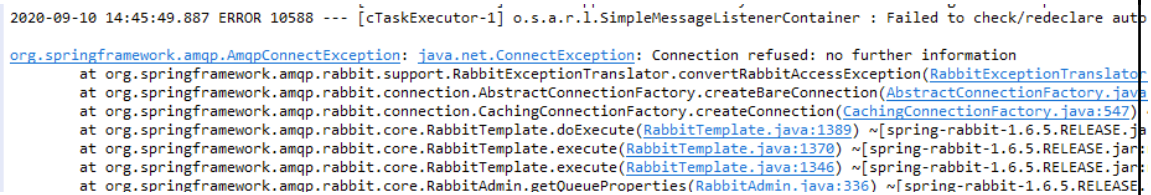


explorer Snippets Problems Console

45\bin\javaw.exe (Dec 9, 2016, 10:43:22 AM)

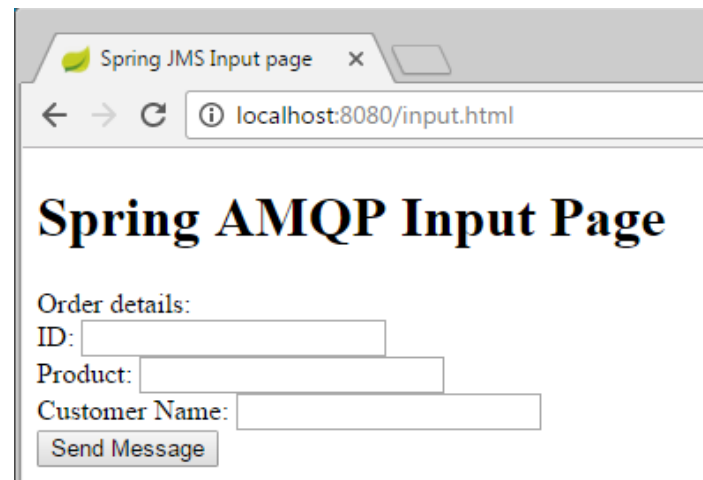
```
: Bean with name 'rabbitConnectionFactory' has been autodetected for JMX exposure
: Located managed bean 'rabbitConnectionFactory': registering with JMX server as MBean
: Starting beans in phase -2147482648
: Starting beans in phase 2147483647
: Created new connection: SimpleConnection@1e22b4c [delegate=amqp://guest@127.0.0.1:5
: Tomcat started on port(s): 8080 (http)
: Started App in 2.934 seconds (JVM running for 3.204)
: Initializing Spring FrameworkServlet 'dispatcherServlet'
```

Troubleshooting: If you see the following error message, then Rabbit MQ is not running. See the previous part of the lab on how to start it. Then re-run the application.



```
2020-09-10 14:45:49.887 ERROR 10588 --- [cTaskExecutor-1] o.s.a.r.l.SimpleMessageListenerContainer : Failed to check/redeclare auto
org.springframework.amqp.AmqpConnectException: java.net.ConnectException: Connection refused: no further information
    at org.springframework.amqp.rabbit.support.RabbitExceptionTranslator.convertRabbitAccessException(RabbitExceptionTranslator.java:47)
    at org.springframework.amqp.rabbit.connection.AbstractConnectionFactory.createBareConnection(AbstractConnectionFactory.java:547)
    at org.springframework.amqp.rabbit.connection.CachingConnectionFactory.createConnection(CachingConnectionFactory.java:336)
    at org.springframework.amqp.rabbit.core.RabbitTemplate.doExecute(RabbitTemplate.java:1389) ~[spring-rabbit-1.6.5.RELEASE.jar:1.6.5.RELEASE]
    at org.springframework.amqp.rabbit.core.RabbitTemplate.execute(RabbitTemplate.java:1370) ~[spring-rabbit-1.6.5.RELEASE.jar:1.6.5.RELEASE]
    at org.springframework.amqp.rabbit.core.RabbitTemplate.execute(RabbitTemplate.java:1346) ~[spring-rabbit-1.6.5.RELEASE.jar:1.6.5.RELEASE]
    at org.springframework.amqp.rabbit.core.RabbitAdmin.getQueueProperties(RabbitAdmin.java:336) ~[spring-rabbit-1.6.5.RELEASE.jar:1.6.5.RELEASE]
```

__4. Open a web browser and navigate to 'localhost:8080/input.html'.



Spring JMS Input page

localhost:8080/input.html

Spring AMQP Input Page

Order details:

ID:

Product:

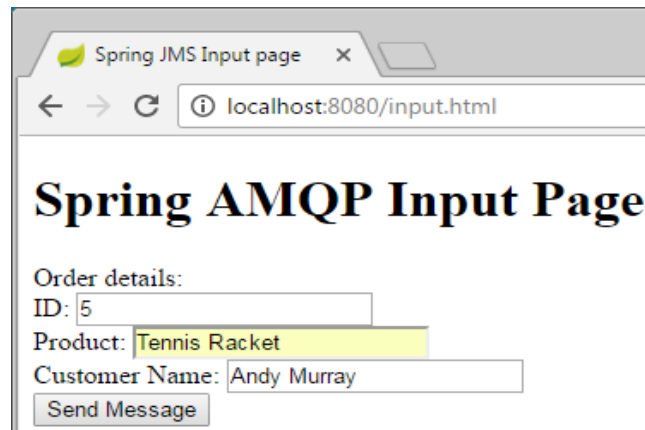
Customer Name:

__5. Enter in the following order details:

ID: 5

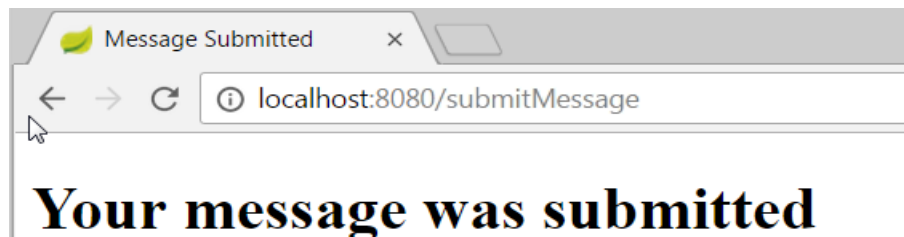
Product: Tennis Racket

Customer Name: Andy Murray



__6. Click on **Send Message**.

__7. You will see this message:



__8. Look in the system console: You should see output that suggest the message was sent and received.

```
2016-12-09 03:17:20.223 INFO 3720 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : F  
2016-12-09 03:17:28.254 INFO 3720 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : F  
getIndex was called  
Order submitted to RequestHandlerServlet: Order #:5, Product:Tennis Racket, Customer:Andy Murray  
Sending message with order - Order #:5, Product:Tennis Racket, Customer:Andy Murray  
Message received - Order Details:Order #:5, Product:Tennis Racket, Customer:Andy Murray
```

__9. Close the browser.

__10. Stop 'App.java' by clicking on the red stop button.

__11. Click **Remove All Terminated Launches**.

__12. Close all open files.

Part 8 - Review

In this lab, we used Spring AMQP to send message into the messaging server. As we saw, Spring Boot handles the vast majority of the configuration automatically, leaving only a minimal amount of work for the programmers to add messaging capability to an application.