Lab Exercise

Service Activators - Lab 7

Throughout this tutorial series, you have seen many service activators without really knowing what a service activator is or does – generically at least. It is time to resolve those questions. Service activators are Spring Integration message endpoints that trigger some service on the arrival of a message into a channel in which the service activator is attached. The service is just a Spring bean – typically nothing more than a plain old Java object – with at least one processing method. The service typically performs some type of business processing based on the incoming message (and its contents), but it doesn't have to. The service activator bean can produce results. Results are placed in an outbound (or reply) message channel.

Service activators are quite simple to create and very versatile components. Because they don't really have any set form or specific type of processing like other SI components such as a transformer, filter, etc. they can be used in any number of situations. In fact, as they see all messages coming through a channel, they could be used as alternates to components like enrichers or transformers. However, they often serve as a Spring Integration conduit to business processing of message information, and as good building blocks and debugging facilities in the construction and maintenance of SI applications. For example, you have already seen, in prior labs, the use of a service activator to print out the contents of messages in the last outbound channel to the Console view. In this lab, you explore how to create and use a business processing service activator.

Specifically, in this lab you will:

- Revisit Spring XML to object message and object to XML transformation.
- Revisit Spring file adapters to read and write XML messages to/from the file system.
- Create a POJO to tally the total revenue from the shipment orders that come through the SI system.
- Configure a service activator to use the POJO to display the running revenue total to the Console view.

Lab solution folder: ExpressSpringIntegration\lab7\lab7-serviceactivators-solution

Scenario – Calculate the shipment order revenue

Already created for you is a rather complex Spring Integration application that uses many of the SI components you have already seen and used. The existing SI application reads XML files into file messages, transforms the file messages to XML string messages, and then transforms string XML messages to Shiporder messages. Shiporder messages are then transformed back to XML string messages, where they are finally transformed to file messages and dumped to the file system. Your task will be to plant a service activator into the middle of this application. The service activator will look at the Shiporder messages and add up the total revenue generated by each order. It will also roll-up all the order totals to provide a grand revenue amount generated by all orders to the Console view in Eclipse.

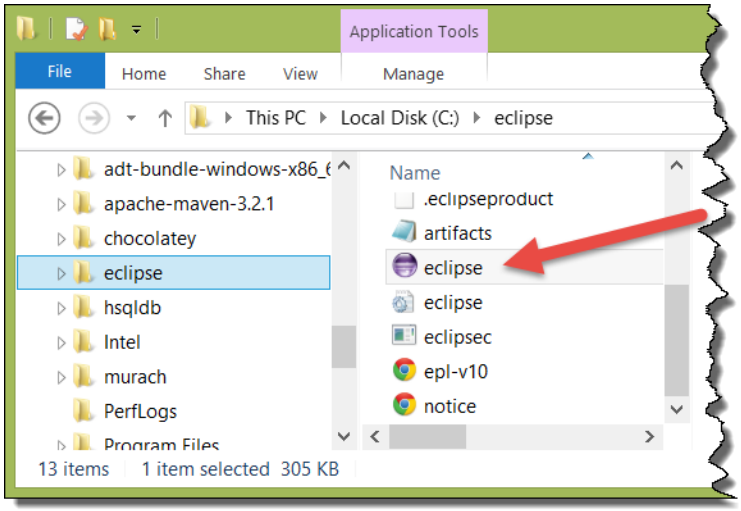
Step 1: Import the Maven Project

A Maven Project containing the base code for a message transforming and revenue calculating application has already been created for you. You will use this project to begin your exploration of enrichers.

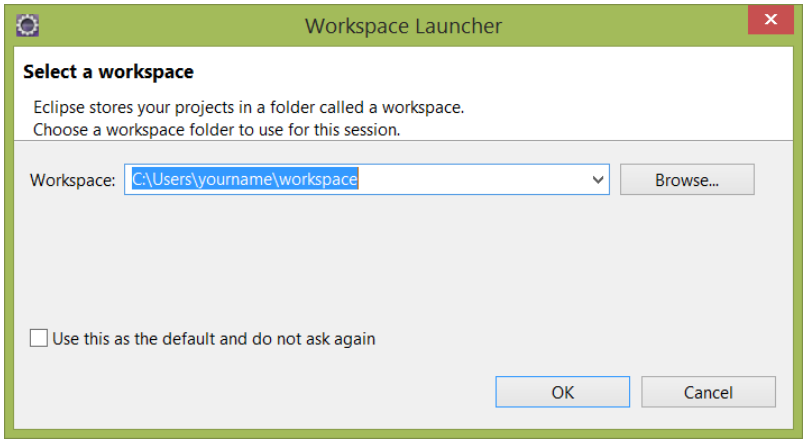
1.1 Start the Eclipse-based IDE. Locate and start Eclipse (or Eclipse-based) IDE.

1.1.1 Locate the Eclipse folder.

1.1.2 Start Eclipse by double clicking on the eclipse.exe icon (as highlighted in the image below).



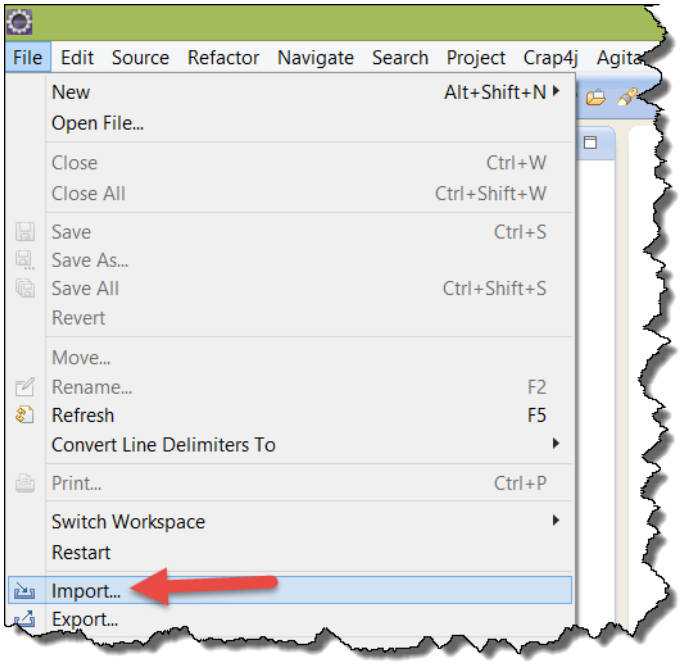
1.1.3 Open your workspace. Type in *C:\Users\<your username >\workspace* and click the *OK* button.



Note: As noted in the past labs, you may use an alternate location for your workspace, but the labs will always reference this location (c:\users\[your username]\workspace) as the default workspace location.

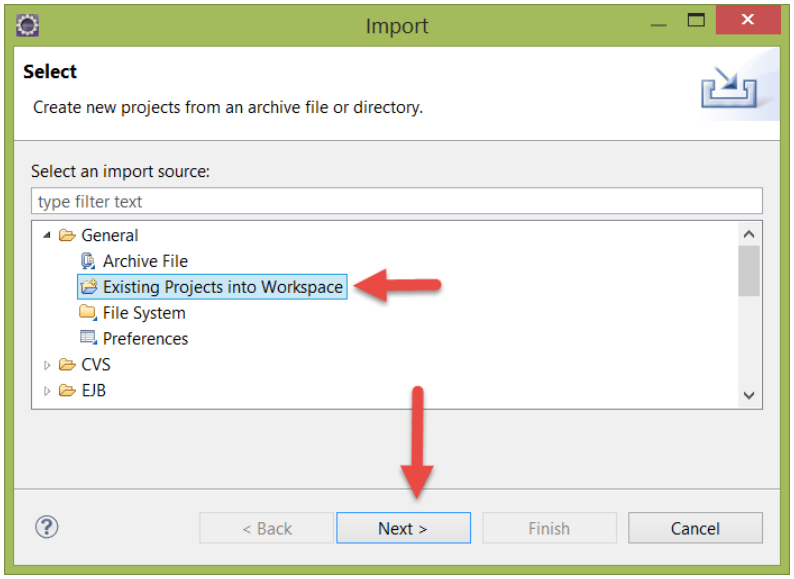
1.2 Import the new Maven Project.

1.2.1 Select File>Import... from the Eclipse menu bar.

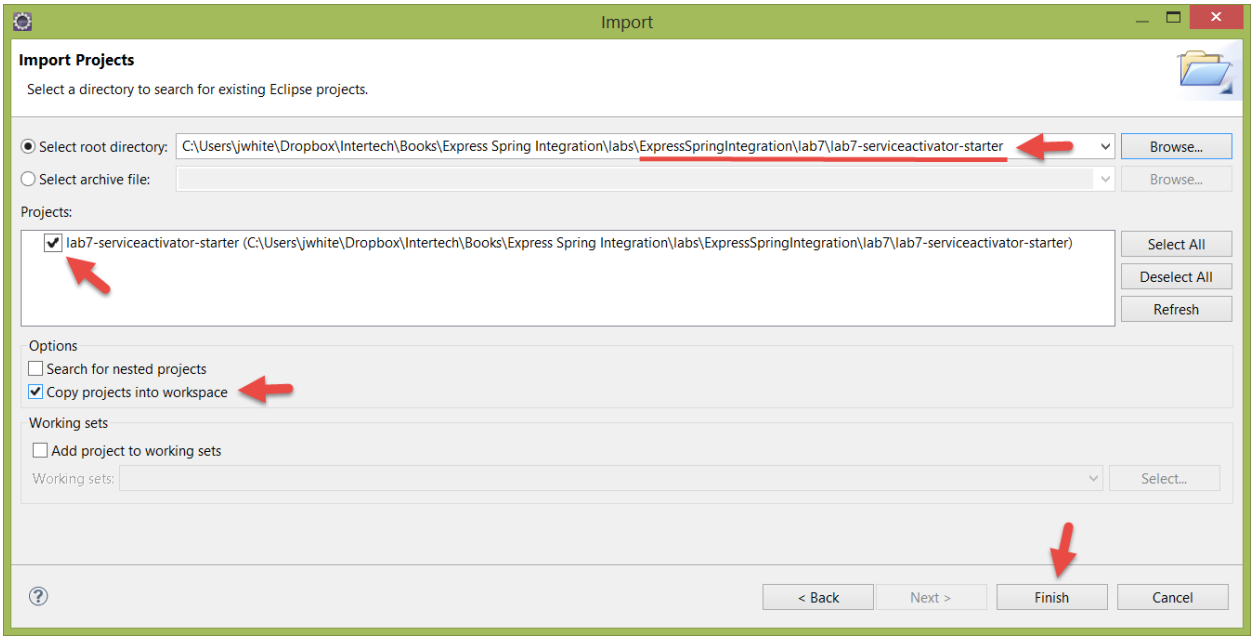


 Rev: 16

1.2.2 Locate and open the General folder in the Import window, and then select *Existing Projects into Workspace Project* from the options. Now push the *Next>* button.

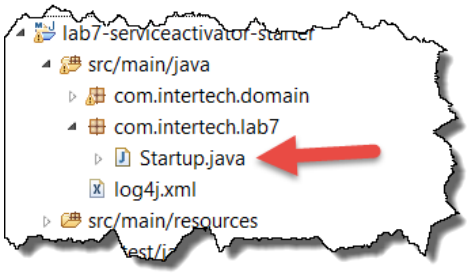


1.2.3 In the "Import" window, use the *Browse...* button to locate the lab7-serviceactivator-starter project folder located in ExpressSpringIntegration\lab7 (this folder is located in the lab downloads). Make sure the project is selected, and the *"Copy projects into workspace"* checkbox is also checked before you hit the *Finish* button (as shown below).

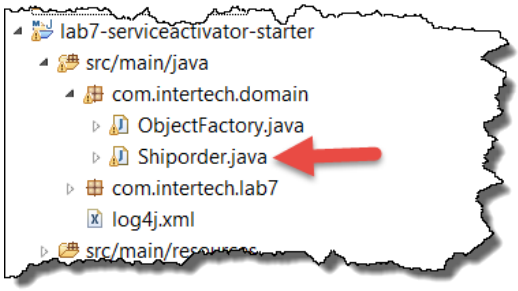


1.3 Explore the project. Examine the project for the Spring Integration components that are already present.

1.3.1 Examine the Startup.java. Expand the src/main/java folder in the Project Explorer view, and open the Startup.java file by double clicking on it. As in past labs, the Startup.java class is used to put the application in an infinite loop so that the SI components can do their work.

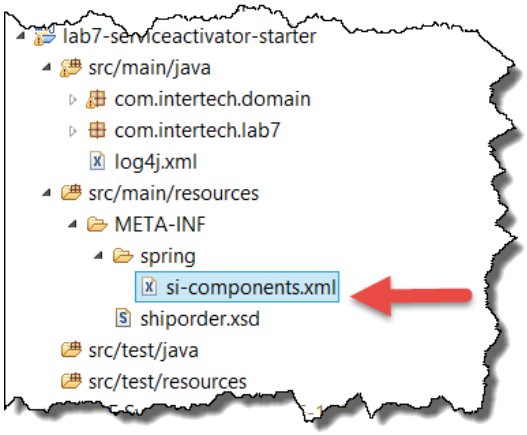


1.3.2 Locate and open the Shiporder.java class found in the com.intertech.domain package in src/main/java folder by double clicking on the file in the Project Explorer.

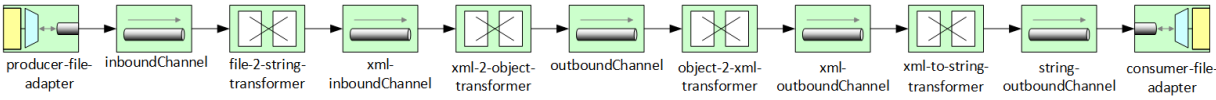


As you learned earlier in this tutorial class, there are three classes defined in this file: Shiporder and two inner classes Item and Shipto. XML messages will again be read and transformed into Shiporder objects with this lab. In particular, the Shiporder objects will be examined by the service activator and the total cost of each of the ship orders will be added up to provide the total revenue generated by the shipments.

1.3.3 Examine the SI configuration. Expand the src/main/resources/META-INF/spring folder in the Project Explorer view, and open si-components.xml file by double clicking on it. The si-components.xml file contains the Spring configuration that includes the definitions for several SI components already.



In particular, note that the application already contains an inbound and outbound file adapters. It also contains a number of transformers. There are transformers for getting files to strings, XML to objects (unmarshaller), objects to XML objects (marshaller), and XML to strings. Of course, there are several channels to connect all the components. Below is the EIP diagram of the application as it currently stands.



This is a complete application. You can test the application at this time if you so desire (go to Step 4 below to see how to prepare the XML messages and execute the application). In running the application, you will find that the application takes XML shipment order messages from c:\\inboundXML and deposits them (unchanged) to c:\\outboundXML by default.

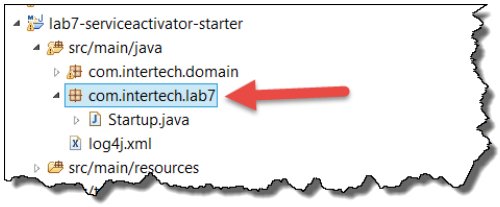
Note: Be aware that at this time, if you run the application, nothing will show up in the Console view. You must watch the c:\\outboundXML folder. Once 3 messages have arrived in the outboundXML folder, you can stop the application.

Step 2: Create a Service Activator

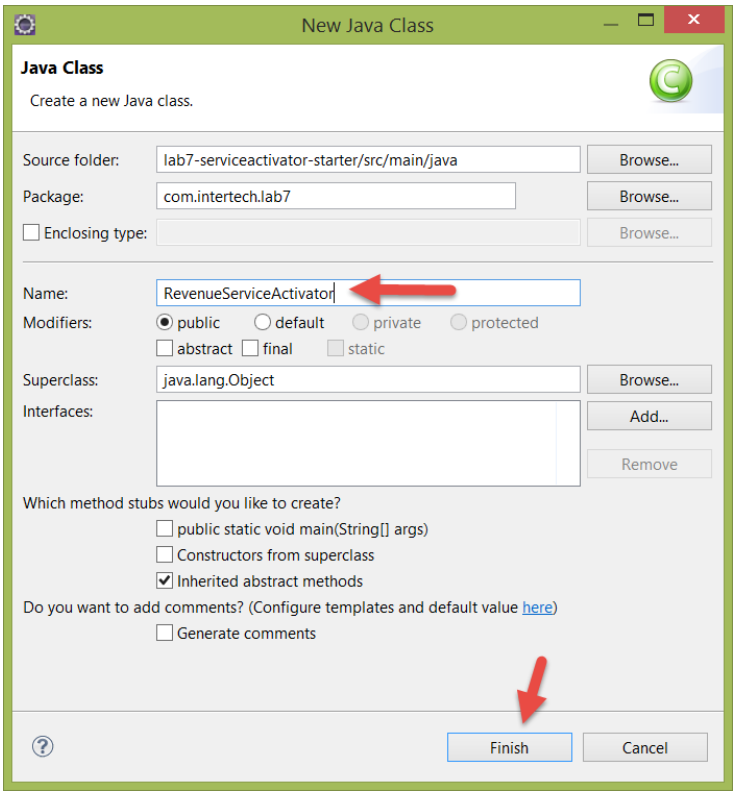
In this step, you create a POJO that calculates the total cost for each shipment order (Shiporder) received through the SI system – adding the total cost to a total revenue number it tracks for all messages received. The POJO represents the business service. You also create a service activator SI component that calls on the POJO each time a message is received in a designated message channel.

2.1 Create the service activator's POJO that provides the business service.

2.1.1 Locate the com.intertech.lab7 package, right click on it and select New > Class from the resulting menu.



2.1.2 In the New Java Class window, enter RevenueServiceActivator as the class name and then press the Finish button.



Note that the class does not have to implement any interfaces or extend any classes – again, it is a simple POJO.

2.2 Code the RevenueServiceActivator class.

2.2.1 Add a revenue holding instance variable to the new RevenueServiceActivator class. This instance variable will hold the total costs (revenue for the receiving company) of all shipment orders passing through the SI application.

```
private double revenue = 0.0;
```

2.2.2 Code the adjustRevenue( ) method. A service activator service bean can have multiple service methods. However, when it does, the configuration of the service activator must indicate the method to call when a message arrives. When a service activator has just one public method – as in this case – Spring Integration knows which method to call and no additional configuration is needed. Enter the following code for the service activator's business processing method.

```
public Message<Shiporder> adjustRevenue(Message<Shiporder> order) {
  System.out.println("Processing order");
  for (Item item : order.getPayload().getItem()) {
    revenue = revenue
        + (item.getPrice().doubleValue() * item.getQuantity()
            .intValue());
    System.out.println("Revenue now up to:  " + revenue);
  }
  System.out.println("Done processing order");
  return order;
}
```

Note that the method gets passed a SI message with a Shiporder payload. It must also return a message containing the Shiporder payload. The return message will be placed in an outbound channel. The method iterates through all items associated to the Shiporder, and calculates the total cost of each order. It then adds the order to the revenue total instance variable.

Note: if you get stuck or feel like not typing in all the code yourself, you will find a working copy of the final RevenueServiceActivator class at ExpressSpringIntegration\lab7\lab7-serviceactivator-solution.

2.2.3 Add the required imports to the RevenueServiceActivator class. You can hit Control-Shift-O to have Eclipse automatically add the right imports. You should find the following imports are required in the class.

```
import org.springframework.messaging.Message;
import com.intertech.domain.Shiporder;
import com.intertech.domain.Shiporder.Item;
```

2.2.4 Save the class and make sure there are not compile errors.

Step 3: Add and Configure the Service Activator Component

With the business service defined, the service activator, Spring bean, and additional message channel need to be added to the Spring Integration configuration.

3.1 Add a message channel for incoming messages. The service activator will need to trigger on the arrival of messages in a channel. The results returned by the service activator will then be published as a message to the outbound channel.

3.1.1 Locate the si-components.xml file in src/main/resource/META-INF/spring and open it by double clicking on the file.

3.1.2 Add a new message channel for incoming Shiporder messages that are to trigger the service activator.

```
<int:channel id="revenueProcessingChannel" />
```

3.1.3 Alter the unmarshalling-transformer to put is outbound messages into the new revenueProcessingChannel versus the outboundChannel as shown below.

```
<int-xml:unmarshalling-transformer
  id="xml-2-object-transformer" input-channel="xml-inboundChannel"
  output-channel="outboundChannel"
  output-channel="revenueProcessingChannel"
unmarshaller="jaxbMarshaller" />
```

3.2 Add the Service Activator.

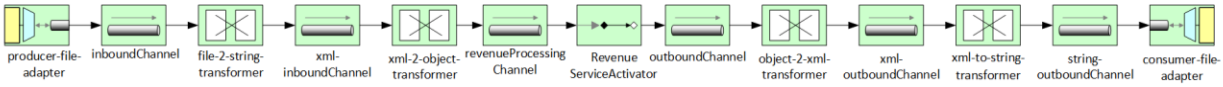
3.2.1 Add the RevenueServiceActivator as a Spring bean.

```
<bean id="revenueServiceBean"
class="com.intertech.lab7.RevenueServiceActivator" />
```

3.2.2 Add the service activator using (by reference) the service bean. The service activator should trigger on messages entering the new revenueProcessingChannel and it should push the results (in message form) to the outboundChannel.

```
<int:service-activator ref="revenueServiceBean"
  input-channel="revenueProcessingChannel"
  output-channel="outboundChannel" />
```

3.2.3 With the completion of the service activator and new channel, the SI application can be represented by the Wolfe and Hohpe EIP model below.



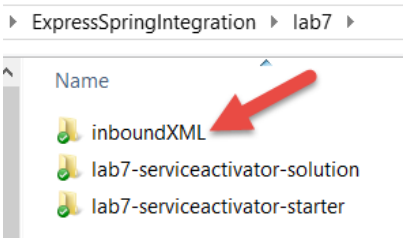
3.3 Save the configuration file and make sure there are no errors in the file.

Step 4: Test the Service Activator

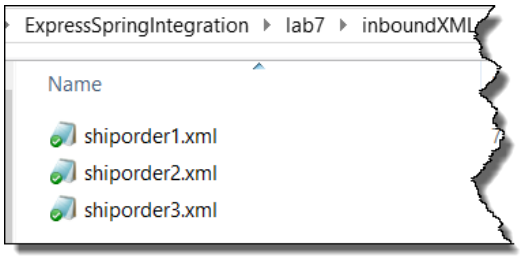
4.1 Add the files to the inbound message folder.

4.1.1 In the si-components.xml, find the producer-file-adapter. Note the location of the directory. It is set, by default, to file:c://inboundXML. This is the location where the XML ship order messages will be taken into the application by the adapter. It is the same folder used in past labs. If it does not already exist, create this message folder - changing the location to suit your needs and your file system (change the producer-file-adapter to reflect your location).

4.1.2 Again, some sample XML ship order messages have been provided to you. Find them in an inboundXML folder in ExpressSpringIntegration\lab7.



Copy the messages from this folder to the c:\\inboundXML folder (or whatever folder you created per 4.1.1. above).



Note: the messages and locations did not change for the last lab. You can use the same messages if they remain from the last lab (lab 6) without the need to copy anew.

4.1.3 Note the outbound channel adapter called consumer-file-adapter in the si-components.xml. In particular, note the location of the directory. It is set, by default, to file:c://outboundXML. It will be automatically created if it does not exist so you do not have to create it (see the auto-create-directory attribute). This is the location where the XML ship order messages will be deposited once they have worked their way through the SI application and service activator. You can look in this directory for the three messages that originated in c:\\inboundXML folder.

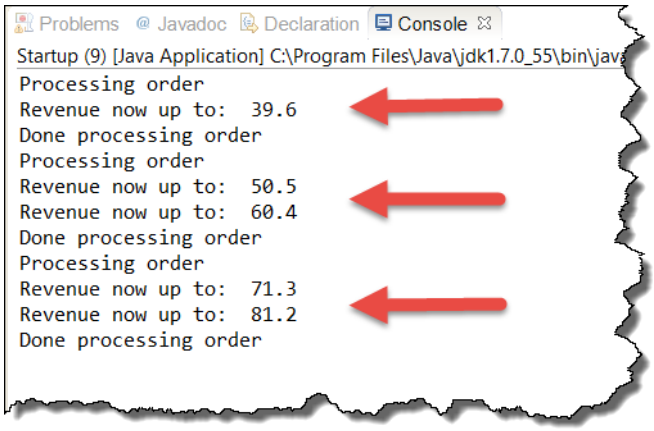
4.2 Test the application. Test the application to see the service activator tally up the order costs as revenue.

4.2.1 Locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

4.2.2 This should cause messages to be ready from c:\\inboundXML and deposited to c:\\outboundXML. Using a Windows Explorer, you can check for new messages in the c:\\outboundXML folder.

Note: depending on the speed of your system, you may have to give the application a couple of seconds to complete its work.

4.3 Check the Console view for results. The RevenueServiceActivator's adjustRevenue( ) method will be called for each message received in the revenueProcessingChannel. For each Shiporder-payload message received, the service activator also writes several lines to the Console view. Namely, it indicates when it has started and stopped processing a Shiporder message and it also indicates new revenue added to the revenue total based on the cost of each item it finds in the Shiporder message.



Note that there are only three shipment order messages to be processed, so you should see three sets of "Processing order" and "Done processing order" messages in the Console view.

4.4 Stop the application. Recall the application is running in an infinite loop to allow for the constant publishing and consuming of messages. Stop the application now.

4.4.1 In the Console view, click on the red square to terminate the Startup application.

4.4.2 The Console view should now indicate that the application is "<terminated>".

Lab Solution

si-components.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-
file="http://www.springframework.org/schema/integration/file"
  xmlns:int-
mail="http://www.springframework.org/schema/integration/mail"
  xmlns:int-
xml="http://www.springframework.org/schema/integration/xml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int-
stream="http://www.springframework.org/schema/integration/stream"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-
integration.xsd
  http://www.springframework.org/schema/integration/stream
http://www.springframework.org/schema/integration/stream/spring-
integration-stream.xsd
  http://www.springframework.org/schema/integration/file
http://www.springframework.org/schema/integration/file/spring-
integration-file.xsd
  http://www.springframework.org/schema/integration/xml
http://www.springframework.org/schema/integration/xml/spring-
integration-xml.xsd">

  <!-- Adapter for reading files -->

  <int-file:inbound-channel-adapter id="producer-file-adapter"
    channel="inboundChannel" directory="file:c://inboundXML"
    prevent-duplicates="true">
    <int:poller fixed-rate="5000" />
  </int-file:inbound-channel-adapter>

  <int:channel id="inboundChannel" />

  <int-file:file-to-string-transformer
    id="file-2-string-transformer" input-channel="inboundChannel"
    output-channel="xml-inboundChannel" charset="UTF-8" />

  <int:channel id="xml-inboundChannel" />

  <int-xml:unmarshalling-transformer
    id="xml-2-object-transformer" input-channel="xml-inboundChannel"
    output-channel="revenueProcessingChannel"
unmarshaller="jaxbMarshaller" />

  <bean id="jaxbMarshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="contextPath" value="com.intertech.domain" />
```



```
  </bean>

  <int:channel id="revenueProcessingChannel" />

  <int:service-activator ref="revenueServiceBean"
    input-channel="revenueProcessingChannel" output-
channel="outboundChannel" />
  <bean id="revenueServiceBean"
class="com.intertech.lab7.RevenueServiceActivator" />

  <int:channel id="outboundChannel" />

  <int-xml:marshalling-transformer id="object-2-xml-transformer"
    input-channel="outboundChannel" output-channel="xml-
outboundChannel"
    marshaller="jaxbMarshaller" result-type="StringResult" />

  <int:channel id="xml-outboundChannel" />

  <int:object-to-string-transformer id="xml-to-string-transformer"
    input-channel="xml-outboundChannel" output-channel="string-
outboundChannel" />

  <int:channel id="string-outboundChannel" />

  <int-file:outbound-channel-adapter
    channel="string-outboundChannel" id="consumer-file-adapter"
    directory="file:c://outboundXML" auto-create-directory="true" />

</beans>
```


RevenueServiceActivator.java

```
package com.intertech.lab7;

import org.springframework.messaging.Message;
import com.intertech.domain.Shiporder;
import com.intertech.domain.Shiporder.Item;

public class RevenueServiceActivator {

  private double revenue = 0.0;

  public Message<Shiporder> adjustRevenue(Message<Shiporder> order) {
    System.out.println("Processing order");
    for (Item item : order.getPayload().getItem()) {
      revenue = revenue
          + (item.getPrice().doubleValue() * item.getQuantity()
              .intValue());
      System.out.println("Revenue now up to:  " + revenue);
    }
    System.out.println("Done processing order");
    return order;
  }
}
```