Spring Integration Adapter Example

Using messaging as communication medium to interact with different external systems is always a challenging task. There is always complexity around the connection mechanisms and transformation of the format across different systems. In this article, we are going to discuss about a useful component for Spring Integration-Adapters.

1. Introduction

Adapter is the most important component for enterprise application integration. Adapter acts as a bridge between the integration framework and the external components. Adapters are of two types. Inbound Adapter and Outbound Adapter. Inbound adapters fetch files, messages or database result set from different external systems. Outbound adapters take messages from channels and convert them to desired format or persist them to database.

Spring Integration provides a comprehensive adapter framework that provides several out-of-box adapters that support different protocols and technologies such as File,JDBC,JMS,FTP and JMX. Let's have a brief look at the definition and purpose of these adapters as below.

1.1 File System Adapter

File System Adapter provides us a capability of sharing files across multiple applications in a distributed environment. File System Adapters can fetch or copy files from different distributed file systems. Behind the scenes, File System Adapter picks a file from the file system and converts into a framework's message and publish it to the channel and vice versa. It's always advisable to use namespace while reading and writing files using File System Adapter

1.2 JDBC Adapter

Most of the enterprise applications require interaction with database and JDBC adapters support sending and receiving messages through database queries. The inbound adapters extracts data from the database and passes the result set as a message on local channels. The outbound adapters persist the data record on the database by reading from the local channels.

1.3 FTP Adapter

Spring Integration supports receiving and sending files to and from the remote server using FTP protocol. Remote files are fetched using FTP adapters and also transferred to the remote server using FTP adapters.

The inbound channel adapters connect to an FTP server to fetch the remote files and pass them as messages in the payload. The outbound channel adapters connect to channels and consume messages and write to remote file directories.

1.4 JMS Adapter

Spring integration framework has a good support for building messaging applications using JMS. Spring Integration framework provides inbound and outbound channel adapters for sending and receiving message across different applications in a distributed system.

The inbound channel adapters pick up a message from JMS destination topic and publish them to local channels. The outbound channel will read the payload from the channel and convert into JMS message and publish it to a JMS destination topic.

1.5 JMX adapter

Spring integration supports JMX adapters for sending and receiving JMX notifications. There is also an inbound channel adapter for polling JMX MBean values and outbound JMX adapter for invoking JMX operations. We will take a detailed look at the types of JMX adapter and also a sample implementation for the same as below.

1.5.1 Notification Publishing Channel Adapter

When we send messages to the channel corresponding to the Notification Publishing adapter, notification content is created from the message. For example if the payload is a String it will be passed as message text for Notification. JMX notifications also have a type and it's a dot-delimited string. We can provide the notification type in multiple ways. We can pass it as a value to the message header `JmxHeaders` i.e `NOTIFICATION_TYPE` or we can pass it as attribute type to `default-notification-type` attribute

```
<int-jmx:notification-publishing-channel-adapter id="adapter"
    channel="channel"
    object-name="some.example.domain:name=publisher"
    default-notification-type="some.example.type"/>
```

1.5.2 Notification Listening Channel Adapter

As the name indicates, Notification Listening Adapter listens for notifications from MBeans. Any notification received from MBeans is put as a message on the channel. Following is a sample configuration of Notification Channel adapter. The object-name indicates the name of the MBean we are listening for events and channel indicates the channel where we will be receiving the notification as messages.

```
<int-jmx:notification-listening-channel-adapter id="notifListener"
     channel="listenForNotification"
     object-name="some.example.domain:name=testMBean,type=TestMBean"/>
```

1.5.3 Attribute Polling Channel Adapter

Attribute Polling adapter polls for an attribute that is managed by MBean. The attribute name of the MBean that has to be polled and the object-name of the MBean has to be defined as part of the declaration. The following is a sample configuration for Attribute Polling Channel Adapter. If there is a change in the `PerfData` attribute of `MonitorMBean` then the change is captured by `attribute-polling-channel-adapter` and these changes are converted to notification messages and are dropped to the `attrDataChannel`. We can configure a ServiceActivator to listen for these messages and take corresponding actions for the same.

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

```
<int-jmx:attribute-polling-channel-adapter id="attribPoller"
    channel="attrDataChannel"
                "some.example.domain:name=monitorMBean,
    object-name=type=MonitorMBean"
    attribute-name="PerfData">
<int:poller max-messages-per-poll="1" fixed-rate="5000"/>
</int-jmx:attribute-polling-channel-adapter>

<int:service-activator ref="exampleServiceActivator" method="attributePolled" input-channel=
"attrDataChannel"/>
```

1.5.4 Operation Invoking Channel Adapter

Putting a message on a predefined channel will trigger the Operation Invoking Channel adapter to invoke an operation exposed by MBean. As you can see in the example below if there is any message dropped on the `messageChannel`then the `setAttrData`method of `TestMBean`will get automatically triggered.

```
<int:channel id="messsageChannel"/>
<int-jmx:operation-invoking-channel-adapter id="triggerOperation"
  channel="messsageChannel"
  object-name="some.example.domain:name=testMBean,type=TestMBean"
  operation-name="setAttrData"/>
```

Sample java code for adding message to the message channel as below.

```
MessageChannel messsageChannel =                                   ,
context.getBean(                           "messsageChannel"MessageChannel.  class);
                                          "Test message for
messsageChannel.send(MessageBuilder.withPayload(trigger"                  ).build());
```

Let’s look at an example of configuring a sample JMX adapter. The below example will explain in detail about different steps for configuring JMX Attribute Polling Channel Adapter.

2. Maven Configuration

Following is the set of dependencies for configuring the sample application for JMX attribute polling adapter.

Pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.springintegration.adapter</groupId>
    <artifactId>spring-integration-adapter</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
         >spring-integration-adapter Maven
    <nameWebapp</                                  name>
    <url>http://maven.apache.org</url>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <springframework.version>4.2.0.RELEASE</springframework.version>
        <spring.integration.version>4.2.0.RELEASE</spring.integration.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-core</artifactId>
            <version>${spring.integration.version}</version>
        </dependency>
        <dependency>
             <groupId>org.springframework.integration</groupId>
             <artifactId>spring-integration-jmx</artifactId>
             <version>${spring.integration.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-stream</artifactId>
            <scope>compile</scope>
            <version>${spring.integration.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <scope>test</scope>
            <version>${spring.integration.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-test</artifactId>
            <scope>test</scope>
            <version>${spring.integration.version}</version>
        </dependency>
        <dependency>
```

```
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>3.8.1</version>
                <scope>test</scope>
            </dependency>

        </dependencies>
        <build>
            <finalName>spring-integration-adapter</finalName>
        </build>
</project>
```

3. Spring Integration Configuration

The core components that are defined as part of configuring JMX attribute polling adapter are mbean, mbean server, attribute polling channel adapter. Spring integration provides a convenient way to define and start mbean servers and also export mbeans using simple tags as below.

The tag to create and start an MBean server is

```
<context:mbean-server/>
```

The tag to export mbeans is

```
<context:mbean-export/>
```

The detailed `spring-integ-context.xml` with different components for JMX attribute polling adapter is as below

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

```xml
    xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration.xsd
http://www.springframework.org/schema/integration/jmx
http://www.springframework.org/schema/integration/jmx/spring-integration-jmx.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/integration/stream
http://www.springframework.org/schema/integration/stream/spring-integration-stream.xsd">

    <context:mbean-export />
    <context:mbean-server />
    <bean id="commonServiceActivator" class=
"com.springinteg.activator.CommonServiceActivator" />
    <context:component-scan base-package="com.springinteg.adapter" />

    <jmx:attribute-polling-channel-adapter
        channel="orders"
        object-name="com.springinteg.adapter.mbean:type=OrderMBean,name=orderMBean"
        attribute-name="Orders">
        <int:poller max-messages-per-poll="1" fixed-delay="1000" />
    </jmx:attribute-polling-channel-adapter>

    <int:publish-subscribe-channel id="orders" />

    <int:service-activator ref="commonServiceActivator"
        method="attributePolled" input-channel="orders" output-channel="processedOrders" />
    <int:channel id="processedOrders">
        <int:queue />
    </int:channel>

    <int:filter ref="maxItemsFilter" method="checkThreshold"
        input-channel="orders" output-channel="reset" />

    <jmx:operation-invoking-channel-adapter
        id="reset" object-name=
"com.springinteg.adapter.mbean:type=OrderMBean,name=orderMBean"
        operation-name="resetOrders" />
</beans>
```

4. Application Configuration

As you have noticed in the above spring context configuration we have defined an `OrderMBean` as part of the attribute polling adapter. Any change to the Orders attribute is captured and sent as a notification message to the message channel. We have configured a common service activator bean that listens to this message channel and then outputs the message payload to console.
A filter component `maxItemsFilter` is defined that basically checks for the number of processed orders on the channel and once it reaches the limit i.e 10 orders then a jmx `operation-invoking-channel-adapter` is defined that basically resets the `orderIds` back to 0.
The following are the list of classes including the MBean configuration as below

4.1 MBean Configuration

OrderMBean.java

```
package com.springinteg.adapter.mbean;
```

```

import java.util.concurrent.atomic.AtomicInteger;

import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Component;

@Component
@ManagedResource
public class OrderMBean{
    private final AtomicInteger orders = new AtomicInteger();

    @ManagedAttribute
    public int getOrders() {
        return this.orders.get();
    }

    @ManagedOperation
    public void incrementOrder() {
        orders.incrementAndGet();
    }
```

```java

    @ManagedOperation
    public void resetOrders() {
        this.orders.set(0);
    }

}
```

4.2 Filter Component Configuration

MaxItemsFilter.java

```java
package com.springinteg.adapter.filter;


import org.springframework.messaging.Message;
import org.springframework.stereotype.Component;

@Component("maxItemsFilter")
public class MaxItemsFilter {
    private static int MAX_THRESHOLD = 10;
    public boolean checkThreshold(Message<?> orderId) {
        if (orderId.getPayload() != null) {
            int orderVal = (Integer) orderId.getPayload();
            if (orderVal > MAX_THRESHOLD){
                return true;
            }
        }
        return false;
    }
}
```

4.3 Service Activator Configuration

CommonServiceActivator.java

```java
package com.springinteg.activator;

import org.springframework.messaging.Message;

public class CommonServiceActivator {

    public String attributePolled(Message msg) {
        String processedMsg = "Order Id ::" + msg.getPayload().toString() + " is being processed";
        return processedMsg;
    }

}
```

5. Verification Test Configuration

The below code shows a basic test for verifying the JMX attribute change notification. In the below test we basically get an instance of `OrderMBean`, then call the attribute method increment order and each time the “Orders” attribute value is incremented , the notification message is sent to the `processedOrders` channel. You can notice that after the order reaches the threshold of greater than 11 items then the `OrderId` gets reset .

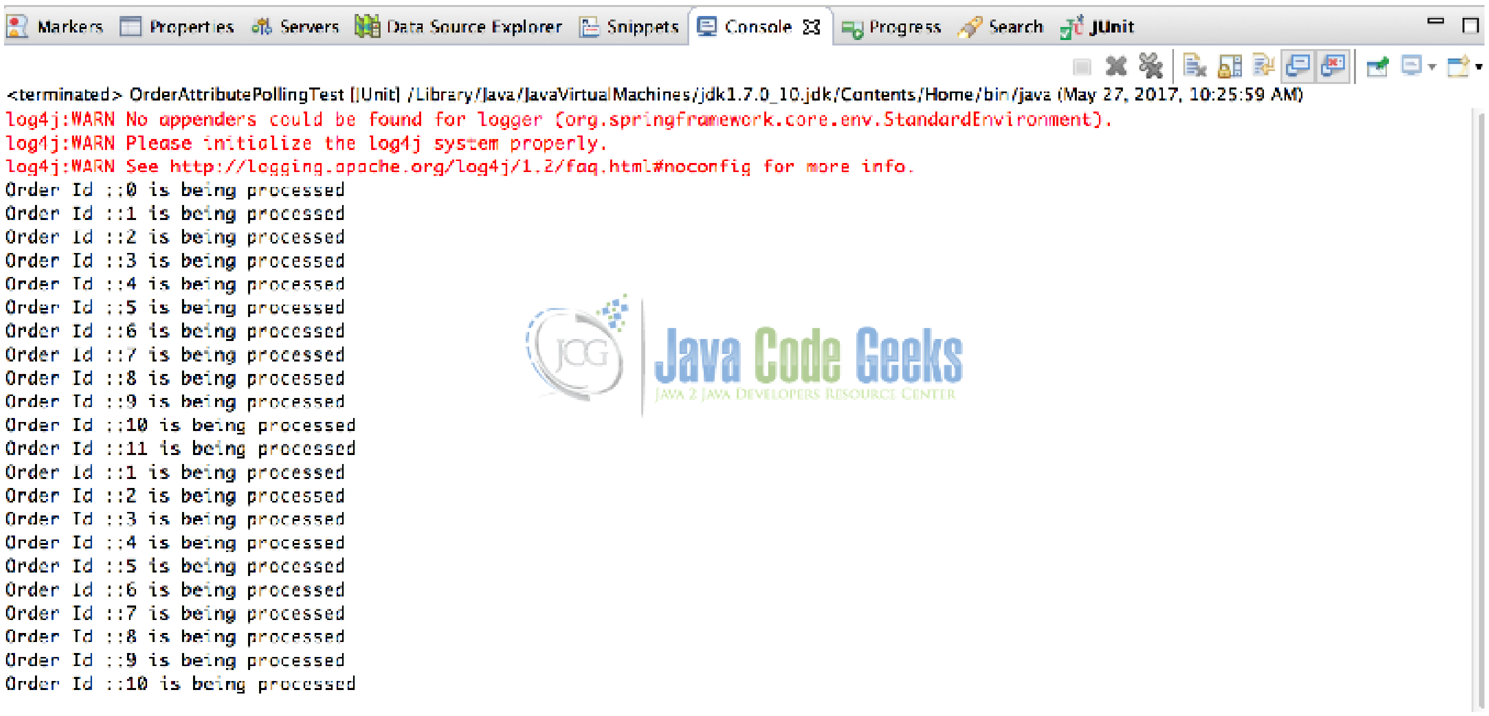
OrderAttributePollingTest.java

```
package com.springinteg.adapter.listener;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.channel.QueueChannel;
import org.springframework.messaging.Message;
import com.springinteg.adapter.mbean.OrderMBean;


public class OrderAttributePollingTest{
    ClassPathXmlApplicationContext context = null;


    @Before
    public void setUp() {
        context = new ClassPathXmlApplicationContext("spring-integ-context.xml");
    }


    @After
    public void destroy() {
        context.stop();
    }

    @Test
    public void testJmxNotification() throws InterruptedException {
        OrderMBean orderMBean = context.getBean("orderMBean",OrderMBean.class);
        orderMBean.incrementOrder();
        Thread.sleep(2000);
        for (int i=1;i<=22;i++){
            QueueChannel processedOrder = context.getBean("processedOrders",QueueChannel.class);
            Message processedMsg = (Message) processedOrder.receive();
            assertNotNull(processedMsg);
            System.out.println(processedMsg.getPayload());
            orderMBean.incrementOrder();
```

```
                Thread.sleep(1000);

            }

        }

}
```

5.1 Screenshot of Test Execution

The below screenshot shows the successful execution of the above test case and the processing of messages from JMX notification channel.

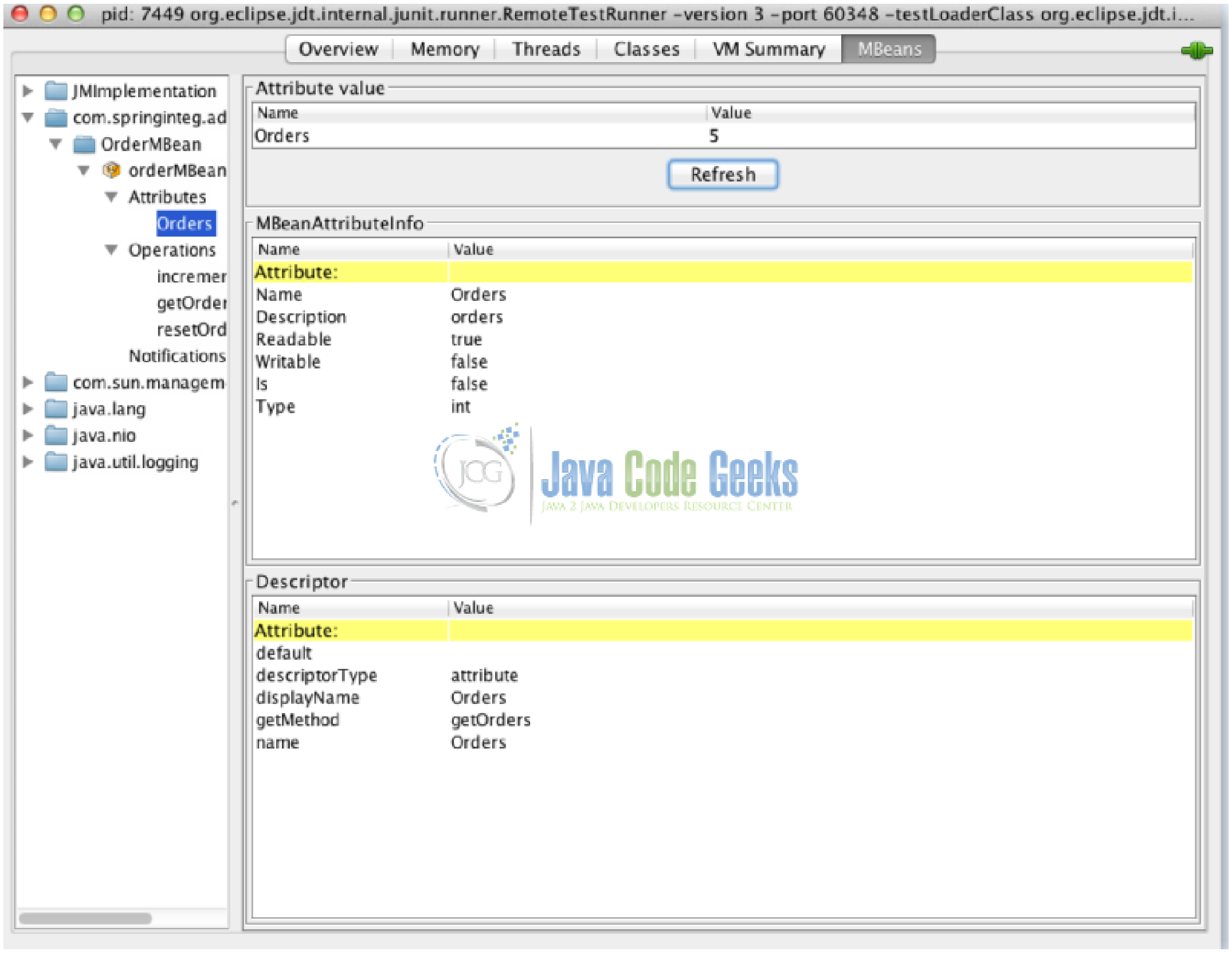


```
<terminated> OrderAttributePollingTest [JUnit] /Library/Java/JavaVirtualMachines/jdk1.7.0_10.jdk/Contents/Home/bin/java (May 27, 2017, 10:25:59 AM)
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Order Id ::0 is being processed
Order Id ::1 is being processed
Order Id ::2 is being processed
Order Id ::3 is being processed
Order Id ::4 is being processed
Order Id ::5 is being processed
Order Id ::6 is being processed
Order Id ::7 is being processed
Order Id ::8 is being processed
Order Id ::9 is being processed
Order Id ::10 is being processed
Order Id ::11 is being processed
Order Id ::1 is being processed
Order Id ::2 is being processed
Order Id ::3 is being processed
Order Id ::4 is being processed
Order Id ::5 is being processed
Order Id ::6 is being processed
Order Id ::7 is being processed
Order Id ::8 is being processed
Order Id ::9 is being processed
Order Id ::10 is being processed
```

JMX attribute polling adapter output

5.2 Screenshot of Jconsole verification

The below screenshot shows the verification of MBeans and the “Orders” attribute value changes



Attribute value

Name	Value
Orders	5

MBeanAttributeInfo

Name	Value
Attribute:	
Name	Orders
Description	orders
Readable	true
Writable	false
Is	false
Type	int

Descriptor

Name	Value
Attribute:	
default	
descriptorType	attribute
displayName	Orders
getMethod	getOrders
name	Orders

JMX Jconsole output

6. Conclusion

Monitoring and management support is one of the critical requirements for a successful enterprise integration. In the above example we have seen how we can leverage the capability of Spring Integration and JMX to create a simple attribute change monitoring adapter. In addition Spring Integration provides capability of error handling, monitoring using JMX and performance measurement.

7. Download The Source Code

The source code for Spring Integration Adapter example as below.

Download
You can download the full source code of this example here: **spring-integration-adapter**