Testing and Monitoring Web Services
by Hamidreza Sattari, IT Professional & Shameer Kunjumohamed, Solutions Architect

Abstract: This article, adapted for the Service Technology Magazine from the Spring Web Services 2 Cookbook [REF-1], covers integration testing using Spring-JUnit support, server-side integration testing using MockWebServiceClient, client-side integration testing using MockWebServiceServer, monitoring TCP messages of a Web Service using TCPMon, and monitoring and load/functional testing a Web Service using soapUI.

Introduction

New software development strategies require comprehensive testing in order to achieve the quality in the software development process. Test-driven design (TDD) is an evolutionary approach to the development process, which combines the test-first development process and re-factoring. In the test-first development process, you write a test before writing the complete production code to simplify the test. This testing includes unit testing as well as integration testing.

Spring provides support for integration testing features using the spring-test package. These features include dependency injection and loading the application context within the test environment. Writing a unit test that uses mock frameworks (such as EasyMock and JMock to test a Web Service) is quite easy. However, it is not testing the content of the XML messages, so it is not simulating the real production environment of testing. Spring Web Services 2.0 provides features to create server-side integration tests as well as the client-side one. Using these integration test features, it is very simple to test a SOAP service without deploying it on the server when you are testing the server side, and without the need to set up a server when you are testing the client side.

In the first recipe, we will discuss how to use the Spring framework for Integration testing. In the next two recipes, new features for integration testing of Spring-WS 2.0 are detailed. In the last two recipes, using tools, such as soapUI and TCPMon for monitoring and testing Web Services, are presented.

Integration Testing Using Spring-JUnit Support

Spring supports integration testing features using the classes in the org.springframework.test package. These features provide dependency injection in your test case using either the production’s application context or any customized one for testing purposes. This recipe presents how to use JUnit test cases using features, spring-test.jar, JUnit 4.7, and XMLUnit 1.1. Please note that to run Integration test, we need to start the server.

Getting Ready

In this recipe, the project’s name is LiveRestaurant_R-3.1 (for server-side Web Service) and has the following Maven dependencies:

- spring-ws-core-2.0.1.RELEASE.jar
- log4j-1.2.9.jar

The following are the Maven dependencies for LiveRestaurant_R-3.1-Client (for the client-side Web Service):

- spring-ws-core-2.0.1.RELEASE.jar
How to Do It

This recipe uses the project used in the recipe Setting up an endpoint by annotating the payload-root, discussed in [REF-1]'s Chapter 1, as the server-side project.

Here is the setup for the client side:

1. Create a test class that calls the Web Service server using WebServiceTemplate in src/test.
2. Configure WebServiceTemplate in applicationContext.xml.
3. From the folder Liverestaurant_R-3.1, run the following command:
   - mvn clean package tomcat:run
4. Open a new command window to Liverestaurant_R-3.1-Client and run the following command:
   - mvn clean package

The following is the client-side output:

```
..............

.................................................................
   T E S T S
.................................................................
Running com.packtpub.liverestaurant.client.OrderServiceClientTest
..............
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.633 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

How It Works

The server-side projects set up a Web Service server and the client-side project runs an integration test and sends predefined request messages to the server and gets the response message from the server. Then it compares the server response with the expected response. Setting up a Web Service and a client of the Web Service are detailed in [REF-1]'s Chapters 1 and 2. Here, only the testing framework is detailed.

In OrderServiceClientTest.java, the method setUpBefore() will be called first to initialize data (since it is annotated by @before) and test methods that are annotated by @Test (testCancelOrderRequest or
testPlaceOrderRequest) to follow, and finally, the method setUpAfter() will be called to free up the resources (since it is annotated by @after).

When you run mvn clean package, Maven builds and runs any test class inside the src/test/java folder. So in OrderServiceClientTest.java, first the test application context will be loaded. In the application context, only the configuration of WebServiceTemplate is required:

```xml
<bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory" />

<bean id="webServiceTemplate" class="org.springframework.ws.client.core.WebServiceTemplate">
    <constructor-arg ref="messageFactory" />
    <property name="defaultUri" value="http://localhost:8080/LiveRestaurant/spring-ws/OrderService"/>
</bean>
```

In OrderServiceClientTest.java, to include the Spring dependency injection and to set up and run the test, code is annotated with some information. The JUnit @RunWith annotation tells JUnit to use the Spring TestRunner. The @ContextConfiguration annotation from Spring tells to load which application context and use this context to inject applicationContext and webServiceTemplate, which are annotated with @Autowired:

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext.xml")
public class OrderServiceClientTest {
    @Autowired
    private WebServiceTemplate webServiceTemplate;
    .......
}
```

@Before from JUnit tells to run the marked method (setUpBefore) before running the test case. JUnit @After causes the marked method to be called after the test case is executed. @Test from JUnit converts the marked methods (testCancelOrderRequest and testPlaceOrderRequest) into JUnit test methods:
@After
public void setUpAfter() {
    applicationContext.close();
}

@Test:
public final void testPlaceOrderRequest() throws Exception {
    Result result = invokeWS(placeOrderRequest);
    XMLAssert.assertXMLEqual("Invalid content received", getStringFromInputStream(placeOrderResponse), result.toString());
}

@Test:
public final void testCaseOrderRequest() throws Exception {
    Result result = invokeWS(caseOrderRequest);
    XMLAssert.assertXMLEqual("Invalid content received", getStringFromInputStream(caseOrderResponse), result.toString());
}

private Result invokeWS(InputStream is) {
    String source = new String(is);
    String result = new String(result);
    webServiceTemplate.sendSourceAndReceiveToResult(source, result);
    return result;
}

public String getStringFromInputStream (InputStream is) throws IOException {
    ByteArrayInputStream bis = new ByteArrayInputStream(is);
    ByteArrayOutputStream buf = new ByteArrayOutputStream();
    int result = bis.read();
    while(result != -1) {
        byte b = (byte)result;
        buf.write(b);
        result = bis.read();
    }
    return buf.toString();
}

Note that for each test method, the @After and @Before methods will be executed once. XMLAsser.
assertXMLEqual compares the real result and the expected XML messages. In a real situation, the data will
change dynamically every day. We should be able to build data dynamically based on dates and from the
database. This helps continuous integration and smoke testing over a period of time. For more information on
this see Chapters 1 and 2 in [REF-1].

Server-Side Integration Testing Using MockWebServiceClient

Writing a unit test that uses mock frameworks, such as EasyMock and JMock, to test a Web Service is quite
easy. However, it does not test the content of the XML messages, so it is not simulating the real production
environment of testing (since these mock objects mimic a part of the software, which is not running, this is
neither unit testing nor integration testing).

Spring Web Services 2.0 provides features to create server-side integration tests. Using this feature, it is very
simple to test a SOAP service without deploying on the server and without the need to configure a test client in
the Spring configuration file.
The main class of server-side integration tests is MockWebServiceClient from the org.springframework.ws.test.server package. This class creates a request message, sends the request to the service, and gets the response message. The client compares the response with the expected message.

Getting Ready
In this recipe, the project's name is LiveRestaurant_R-3.2 (as the server-side Web Service that includes a test case that uses MockWebServiceClient) and has the following Maven dependencies:

- spring-ws-core-2.0.1.RELEASE.jar
- spring-ws-test-2.0.1.RELEASE.jar
- spring-test-3.0.5.RELEASE.jar
- log4j-1.2.9.jar
- junit-4.7.jar

How to Do It
This recipe uses the project from Setting up an endpoint by annotating the payload-root, discussed in [REF-1]'s Chapter 1, Building SOAP Web Services, as the server-side project. Here is the setup for the test case:

1. Include the following data in pom.xml:

   ```xml
   <testResources>
     <testResource>
       <directory>src/main/webapp</directory>
     </testResource>
   </testResources>
   </build>

2. Run the following command for Liverestaurant_R-3.2:

   - mvn clean package

   The following is the server-side output:
How It Works

In the class OrderServiceServerSideIntegrationTest.java, annotation and unit testing materials are the same as those used in the recipe Integration testing using Spring-JUnit support. The only difference here is that we are not setting up the server.

Instead, we load the server application context in the test case class:

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/WEB-INF/spring-vs-servlet.xml")
public class OrderServiceServerSideIntegrationTest {

    @Before
    public void createClient() {
        wsMockClient = MockWebServiceClient.createClient(applicationConf
text);
    
        placeOrderRequest = new OrderServiceServerSideIntegrationTest().
getClass().getResourceAsStream("placeOrderRequest.xml");

    }

    @Before
    public void createClient() {
        wsMockClient = MockWebServiceClient.createClient(applicationConf
text);
        placeOrderRequest = new OrderServiceServerSideIntegrationTest().
getClass().getResourceAsStream("placeOrderRequest.xml");
        }
```

The test case class, in the @Before method, initializes an instance of the client mock object and XML messages:
Then, it sends a message and receives the response. It then compares the expected response and the real response:

```java
@After
public void setSpAfterClass() {
    applicationContext.close();
}

@Test
public final void testPlaceOrderRequest() throws Exception {
    Source requestPayload = new StreamSource(placeOrderRequest);
    Source responsePayload = new StreamSource(placeOrderResponse);
    wsMockClient.sendRequest(withPayload(requestPayload)).andExpect(payload(responsePayload));
}

@Test
public final void testCancelOrderRequest() throws Exception {
    Source requestPayload = new StreamSource(cancelOrderRequest);
    Source responsePayload = new StreamSource(cancelOrderResponse);
    wsMockClient.sendRequest(withPayload(requestPayload)).andExpect(payload(responsePayload));
}
```

In the method createClient(), MockWebServiceClient.createClient (applicationContext) creates an instance of the client mock object (wsMockClient). In the test case methods (testCancelOrderRequest, testPlaceOrderRequest), using the code wsMockClient.sendRequest(withPayload(requestPayload)).andExpect (payload(responsePayload)), the mock client sends an XML message and compares the response (from server endpoint) with the expected response (The client mock is aware of server endpoint from application context file and when it sends request to server, invokes the endpoint method and gets the response back). For more information on this see Chapter 1 of [REF-1].

**Client-Side Integration Testing Using MockWebServiceServer**

Writing a client-side unit test that uses mock frameworks to test a client of a Web Service is quite easy. However, it does not test the content of the XML messages that are sent over the wire, especially when mocking out the entire client class.

Spring Web Services 2.0 provides features to create client-side integration tests. Using this feature, it is very simple to test the client of a SOAP service without setting up a server. The main class of client-side integration tests is MockWebServiceServer from the org.springframework.ws.test.server package. This class accepts a request message from a client, verifies it against the expected request messages, and then returns the response message back to the client.

Since this project is a client-side test integration using MockWebServiceServer, it doesn't need any external server-side Web Service.
Getting Ready

In this recipe, the project’s name is LiveRestaurant_R-3.3-Client (as the client-side project that includes a test case that uses MockWebServiceServer as the server) and has the following Maven dependencies:

- spring-ws-core-2.0.1.RELEASE.jar
- spring-ws-test-2.0.1.RELEASE.jar
- spring-test-3.0.5.RELEASE.jar
- log4j-1.2.9.jar
- junit-4.7.jar

How to Do It

This recipe uses the client-side project from Creating a Web Service client on HTTP transport, discussed in Chapter 2 of [REF-1]. Here is the setup for the test case:

1. Create a test case class under src/test:
2. Create a class that extends WebServiceGatewaySupport to send/receive messages.
3. Run the following command for LiveRestaurant_R-3.3-Client:
   - mvn clean package
   The following is the server-side output:

   *********************************************
   __________________________________________
   Tests
   __________________________________________
   Running com.packtpub.liverestaurant.client.test.
   ClientSideIntegrationTest
   .......
   Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.945 sec
   Results:
   Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

How It Works

The flow in the test case class ClientSideIntegrationTest.java is as follows:

1. Create a MockWebServiceServer using WebServiceGatewaySupport (OrderServiceClient that extends WebServiceGatewaySupport). You can also create MockWebServiceServer using WebServiceTemplate or using ApplicationContext.
2. Set up request expectations using RequestMatcher and return the response using ResponseCreator.
3. Make a client call by using the WebServiceTemplate.

4. Call the verify method to make sure all the expectations are met. The application context file is just a configuration of WebServiceTemplate and OrderServiceClient:

```xml
<bean id="client" class="com.packtpub.liverestaurant.client.test.OrderServiceClient">
    <property name="webServiceTemplate" ref="webServiceTemplate"/>
</bean>

<bean id="webServiceTemplate" class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="defaultUri" value="http://www.packtpub.com/liverestaurant/OrderService/schema"/>
</bean>
</beans>
```

Inside ClientSideIntegrationTest.java, the annotation and unit testing materials are the same as those used in the recipe Integration testing using Spring-JUnit support. The method createServer() creates MockWebServiceServer using WebServiceGatewaySupport (OrderServiceClient extends WebServiceGatewaySupport):

```java
public class OrderServiceClient extends WebServiceGatewaySupport {

    public Result getStringResult(Source source) {
        StringResult result = new StringResult();
        getWebServiceTemplate().sendSourceAndReceiveToResult(source, result);
        return result;
    }
}
```

In the test, the method testExpectedRequestResponse, mockServer.expect sets the expected request and response (webServiceTemplate is configured in ‘testing mode’ in client-integration-test.xml. When the sendSourceAndReceiveToResult method is being called, the template calls server virtually without any real HTTP connection). Then client.getStringResult calls webserviceTemplate to call the server(MockWebServiceServer). Then, mockServer.verify checks if the returned response matches the expected one:
In the test method `testSchemaWithWrongRequest`, the schema of the expected request and response is used. However, the client is trying to send invalid request that is to be failed:

```java
@Test
class TestService {
    public void testSchemaWithWrongRequest() throws Exception {
        Resource schema = new FileSystemResource("orderService.xsd");
        mockServer.expect(new StringSource(getStringFromInputStream(cancelOrderRequestWrong))).andRespond(new Payload("Invalid content received"), new StreamSource(placeOrderRequest));
        client.getStringResult(new StringSource(placeOrderRequest));
    }
}
```

In the test method `testSchema`, instead of using a hardcoded request/response, the schema of the expected request and response is used. This test can test if the format of the request/response is as expected. This is shown as follows:

```java
@Test
public void testSchema() throws Exception {
    Resource schema = new FileSystemResource("orderService.xsd");
    mockServer.expect(new StringSource(getStringFromInputStream(placeOrderRequest))).andRespond(new Payload("Invalid content received"), new StreamSource(placeOrderRequest));
    client.getStringResult(new StringSource(placeOrderRequest));
}
```

In the test method `testExpectedRequestResponse`, the expected request and response is used. However, the client is trying to send invalid request that is to be failed:

```java
@Test
public void testExpectedRequestResponse() throws Exception {
    StreamSource requestPayload = getStreamFromInputStream(placeOrderRequest);
    StreamSource responsePayload = getStreamFromInputStream(placeOrderResponse);
    mockServer.expect(new Payload(requestPayload)).andRespond(new Payload(responsePayload));
    Result result =
        client.getStringResult(requestPayload);
    XMLAssert.assertXMLEqual("Invalid content received", xmlToString(responsePayload), result.toXML());
    mockServer.verify();
}
```
Monitoring TCP Messages of a Web Service Using TCPMon

TCPMon is an Apache project with a Swing UI, which provides features to monitor TCP-based messages transmitted between the client and server. A SOAP message can also be sent to the server using TCPMon.

This recipe presents how to monitor messages passed between a Web Service client and the server. In addition, it shows how to send a SOAP message using TCPMon. The recipe Integration testing using Spring-JUnit support is used for server-side and client-side projects.

Getting Ready

Download and install TCPMon 1.0 from the website http://ws.apache.org/commons/tcpmon/download.cgi.

How to Do It

1. Run it on Windows using tcpmon.bat (tcpmon.sh for Linux).
2. Enter the values 8081 and 8080 into the Listen port # and Target port # fields and click on the Add option.
3. Change applicationContext.xml in LiveRestaurant_R-3.1-Client to use the 8081 port for webserviceTemplate:

```xml
    <bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory" />
    
    <bean id="webServiceTemplate" class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory" />
        <property name="defaultUri" value="http://localhost:8081/LiveRestaurant/spring-ws/OrderService" />
    </bean>
```

4. Run the server from the project LiveRestaurant_R-3.1 using the following command:
   - mvn clean package tomcat:run

5. Run the client from the project LiveRestaurant_R-3.1-Client using the following command:
   - mvn clean package
Go to the Port 8081 tab and see request and response messages, as shown in the following screenshot:
Send a SOAP request to the server as follows: Go to the Sender tab. Enter the SOAP service address and a SOAP request message and click on the Send button to view the response:

**How to Do It**

Monitoring transmitted messages between a client and a Web Service server is the most important usage of the TCPMon. In addition, TCPMon can be used as a client to send a message to a Web Service server. This is an intermediary role that shows the transmitted messages between the client and server. The client has to point to the intermediary instead of the server service.

The second activity (sending a SOAP request to the server) shows the sending of a message using TCPMon to the server, the reception of the response, and shows all of this on TCPMon.
Monitoring and Load/Functional Testing of a Web Service Using soapUI

soapUI is an open source testing solution for testing web services. Using a user-friendly GUI, this tool provides a feature to create and execute automated functional and load testing as well as monitor SOAP messages.

This recipe presents how to monitor SOAP messages of the Web Service and functional and load testing using soapUI. To set up a Web Service, recipe Integration testing using Spring-JUnit support, is used.

Getting Ready

Get started by carrying out the following steps:

1. Install and run soapUI 4.0 (http://www.soapui.org/).
2. Run the following command from the folder LiveRestaurant_R-3.1:
   - mvn clean package tomcat:run

How to Do It

To run the functional tests and monitor the SOAP messages, carry out the following steps:

1. Right-click on the Projects node. Select New soapUI Project and enter the WSDL URL and the Project Name.
2. Right-click on the project's name, OrderService, in the navigator pane. Select Launch HTTP Monitor and enable the option Set as Global Proxy. Click on the OK button:
3. Expand the OrderService methods (cancelOrder and placeOrder). Double-click cancelOrder. Click on Submit Request to Specific Endpoint URL (The green icon on the top-left corner of the Request1 screen). The following is the output of this action:
4. Right-click OrderServiceSoap11 | Generate Test Suite | OK. Enter OrderServiceSoap11 TestSuite.

5. Double-click on OrderServiceSoap11 TestSuite on the navigator pane. Click Run the selected TestCases.
6. The following is the output when the test suite is run:
Run a load test as follows:

1. Right-click the cancelOrder test case. Select New Local Test and enter the Load Test Name.
2. Double-click Load test name. Enter Parameter and click on Run Load Test.
3. The following is the output of the test:

![Test output](image)

**How It Works**

Functional testing and monitoring SOAP messages: soapUI provides three levels of functional testing: test suites, test cases, and test steps.

Test cases are the unit tests that are generated from the WSDL file and test suites are a collection of these unit tests. Test steps control the flow of execution and validate the functionality of the service that is to be tested. For example, a test case in the test suite for the cancelOrder mentioned previously may test the database first. If there is such an order available, it cancels the order.

Load testing: soapUI provides a feature to run multiple threads (as many as your machine's hardware limits you to) on your test cases. When you run a load test, the underlying test case will be cloned internally for each thread. Delay settings let each thread wait before starting and let the Web Service rest for each thread.

**References**

Hamidreza Sattari

Hamidreza Sattari is an IT professional and has been involved in several areas of Software Engineering, from programming to architecture as well as management since 2002. His area of interest has been integration among the software applications. He holds a Master’s degree in Software Engineering from Herriot Watt University, UK, and a Bachelor’s degree in Electrical Engineering (Electronics) from Tehran Azad University, Iran. In recent years, his research area of interest has been scientific data mining using algorithms and statistical techniques in pattern recognition, estimation, and machine learning. He maintains the blog Just Developed and is a co-author for the book Spring Web Services 2 Cookbook.

Contributions

■ Testing and Monitoring Web Services

Shameer Kunjumohamed

Shameer Kunjumohamed is a software architect, specialized in enterprise application integrations and SOA. He is well versed in J2EE and Microsoft.Net platforms. He is interested in various mobile platforms including Android, Blackberry and other cross-platform mobile frameworks these days. After graduating from Calicut University, India in 2000, Shameer handled different roles in software engineering. He graduated with a Master’s in Software Engineering from Herriot Watt University, UK from Dubai International Academic City in 2009. At present he works as a Solutions Architect in Dubai, UAE, and is a guest lecturer in the same university for the Masters students in Information Technology.

Contributions

■ Testing and Monitoring Web Services