Chapter No. 3
"Testing and Monitoring Web-Services"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.3 "Testing and Monitoring Web-Services"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

Hamidreza Sattari started software development in 2002 and has been involved in several areas of Software Engineering, from programming to architecture as well as management. His area of interest has been integration among the software applications.

Hamidreza Sattari earned his Master's degree in Software Engineering in 2008 from Herriot Watt University, UK, and his Bachelor's degree in 1994 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 justdeveloped.blogspot.com.

First, I should thank the open source community that is too large to name. Definitely without using the products, ideas, articles, and web log of this community, I would have never been able to write this book. Besides, I would like to thank my friend, Shameer P.K., for his cooperation in writing this book.

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
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 that are in the market these days.

After graduating from Calicut University, India, in 2000, Shameer handled different roles in software engineering. He earned his Master's degree in Software Engineering from Heriot Watt University of UK (Dubai campus) in 2009. He has worked for Wellogic ME, at Dubai Internet City. At present, he works as a Solutions Architect in Dubai, UAE, and is a guest lecturer at Heriot Watt University for Post Graduate students in Information Technology.

He maintains the blog http://justcompiled.blogspot.com.

I would like to thank a number of people who encouraged me to write this book and reviewed my blog (http://justcompiled.blogspot.com), which was a route to this book. I thank my wife Shehida, and my daughters Shireen and Shahreen, who were supporting me and bearing with me when I was busy writing the chapters. It was their precious time I was utilizing for this book. Also, I would like to thank my friend, Hamidreza Sattari, who is a great friend and colleague; without his support and hard work, I wouldn't have taken up this challenge. I extend my thanks to all those who contributed to my knowledge and passion towards technology to make me capable of writing this book.
Spring Web Services 2
Cookbook

Spring Web-Services (Spring-WS), introduced by the SpringSource community (http://www.springsource.org/), aims to create contract-first SOAP Web-Services in which either a WSDL or an XSD is required primarily for the creation of a Web-Service. Since Spring-WS is a Spring-based product, it takes advantage of Spring’s concepts such as Inversion of Control (IOC) and dependency injection. Some of the key features of Spring-WS are:

- Powerful endpoint mappings: The incoming XML requests can be forwarded to any handler object, based on the payload, SOAP action, and an XPath expression
- Rich XML API support: The incoming XML messages can be read using a variety of Java's XML APIs such as DOM, JDOM, dom4j, and so on
- Built by Maven: Spring-WS can be easily integrated with your Maven project
- Support for Marshalling technologies: Several OXM technologies, such as JAXB, XMLBean, XStream, and JiBX, can be used alternatively for the conversion of XML messages to/from an object
- Security support: Security operations, such as encryption/decryption, signature, and authentication

Covering all of these key features of Spring-WS 2.x has been the main goal of this book. However, in the last two chapters, a different approach toward Web-Service development using REST-style and contract-last development using Spring remoting feature are detailed.

What This Book Covers

Chapter 1, Building SOAP Web-Services: This chapter covers setting up SOAP Web-Services over HTTP, JMS, XMPP, and E-mail protocols. It also covers the different implementations of Web-Service's endpoint using technologies such as DOM, JDOM, XPath, and Marshaller.

Chapter 2, Building Clients for SOAP Web-Services: This chapters covers building SOAP Web-Services clients over HTTP, JMS, XMPP, and E-mail protocols, using Spring-WS template classes.

Chapter 3, Testing and Monitoring Web-Services: This chapter explains the testing of Web-Services using the latest features of Spring-WS and monitoring a Web-Service using tools such as soapUI and TCPMon.

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
Chapter 4, Exception/ SOAP Fault Handling: This chapter explains exception handling in the case of application/system failure.

Chapter 5, Logging and Tracing of SOAP Messages: In this chapter, we will see how to log important events and trace Web-Services.

Chapter 6, Marshalling and Object-XML Mapping (OXM): We will discuss marshalling/unmarshalling technologies as well as creating a custom marshaller in this chapter.

Chapter 7, Securing SOAP Web-Services using XWSS Library: This chapter covers security topics, such as encryption, decryption, digital signature authentication, and authorization using the Spring-WS feature, based on XWSS, and has a recipe about creating key stores.

Chapter 8, Securing SOAP Web-Services using WSS4J Library: In this chapter, we will see security topics, such as encryption, decryption, digital signature authentication, and authorization using the Spring-WS feature, based on the WSS4J package.

Chapter 9, RESTful Web-Services: This chapter explains REST Web-Service development using RESTful support in Spring.

Chapter 10, Spring Remoting: We will discuss contract-last Web-Service development using Spring remoting features to expose local business services as a Web-Service using Hessian/ Burlap, JAX-WS, JMS, and a recipe to set up a Web-Service by Apache CXF using JAX-WS API.

Testing and Monitoring Web-Services

In this chapter, we will cover:

- 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
- 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.

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
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. However, in the next two recipes, we will use new features for integration testing of Spring-WS 2.0 that do not require starting up 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
- spring-test-3.0.5.RELEASE.jar
- log4j-1.2.9.RELEASE.jar
- junit-4.7.jar
- xmlunit-1.1.jar

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
How to do it...

This recipe uses the project used in the recipe Setting up an endpoint by annotating the payload-root discussed in Chapter 1, Building SOAP Web-Services, 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 compare the server response with the expected response. Setting up a Web-Service and a client of the Web-Service have already been detailed in the first two chapters. 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
    public void setUpBefore() {
        applicationContext.close();
    }

    @Test
    public final void testPlaceOrderRequest() throws Exception {
```

For More Information:

www.packtpub.com/spring-web-services-cookbook/book
Result result = invokeWS(placeOrderRequest);
XMLAssert.assertXMLEqual("Invalid content received", getStringFromInputStream(placeOrderResponse), result.toString());

@Test
public final void testCancelOrderRequest() throws Exception {
    Result result = invokeWS(cancelOrderRequest);
    XMLAssert.assertXMLEqual("Invalid content received", getStringFromInputStream(cancelOrderResponse), result.toString());
}

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

public String getStringFromInputStream (InputStream is) throws IOException {
    BufferedInputStream bis = new BufferedInputStream(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. XMLAssert.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.

See also

The recipe Setting up an endpoint by annotating the payload-root, discussed in Chapter 1, Building SOAP Web-Service.

The recipe Creating a Web-Service client on HTTP transport, discussed in Chapter 2, Building Clients for SOAP Web-Services.

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
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 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>
   ```

For More Information:

www.packtpub.com/spring-web-services-cookbook/book
Add the test case class in the folder `src/test/java`.

2. Run the following command for `Liverestaurant_R-3.2`:

```shell
mvn clean package
```

The following is the server-side output:

```
..............
-------------------------------------------------------
TESTS
-------------------------------------------------------
Running com.packtpub.liverestaurant.service.test.OrderServiceServerSideIntegrationTest
1...........
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.047 sec
Results :

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

**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-ws-servlet.xml")
public class OrderServiceServerSideIntegrationTest {
    ...
}
```

The test case class, in the `@Before` method, initializes an instance of the client mock object and XML messages:

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

For More Information:

cancelOrderRequest = new OrderServiceServerSideIntegrationTest().getClass().getResourceAsStream("cancelOrderRequest.xml");
placeOrderResponse = new OrderServiceServerSideIntegrationTest().getClass().getResourceAsStream("placeOrderResponse.xml");
cancelOrderResponse = new OrderServiceServerSideIntegrationTest().getClass().getResourceAsStream("cancelOrderResponse.xml");
}

Then, it sends a message and receives the response. It then compares the expected response and the real response:

```java
@After
public void setUpAfterClass() {
    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:
www.packtpub.com/spring-web-services-cookbook/book
See also

The recipes *Integration testing using Spring-JUnit support and Client-side integration testing using MockWebServiceServer*, discussed in this chapter.

The recipe *Setting up an endpoint by annotating the payload-root*, discussed in Chapter 1, *Building SOAP Web-Services*.

**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 `MockServiceServer` 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`

For More Information:  
www.packtpub.com/spring-web-services-cookbook/book
Testing and Monitoring Web-Services

How to do it...

This recipe uses the client-side project from Creating a Web-Service client on HTTP transport, discussed in Chapter 2, Building Clients for SOAP Web-Services. 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 client-side output:

   **************************
   -------------------------------------------------------
   T E S T S
   -------------------------------------------------------
   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.

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
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:

```java
@Test
    public void testExpectedRequestResponse() throws Exception {
        Source requestPayload = new StringSource(getStringFromInputStream(placeOrderRequest));
        Source responsePayload = new StringSource(getStringFromInputStream(placeOrderResponse));
        mockServer.expect(payload(requestPayload)).andRespond(withPayload(responsePayload));
    }
```

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
Testing and Monitoring Web-Services

```java
Result result = client.getStringResult(requestPayload);
XMLAssert.assertXMLEqual("Invalid content received",
xmlToString(responsePayload), result.toString());
mockServer.verify();
}
```

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(validPayload(schema));
    client.getStringResult(new StreamSource(placeOrderRequest));
    mockServer.verify();
}
```

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(expected = AssertionError.class)
public void testSchemaWithWrongRequest() throws Exception {
    Resource schema=new FileSystemResource("orderService.xsd");
    mockServer.expect(validPayload(schema));
    client.getStringResult(new StringSource(getStringFromInputStream(cancelOrderRequestWrong)));
    mockServer.verify();
}
```

See also

The recipe *Integration testing using Spring-JUnit support*, discussed in this chapter.

**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.

For More Information:

Getting ready

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

How to do it...

Monitor the messages between the client and server as follows:

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">
   </bean>
   ``

For More Information:

www.packtpub.com/spring-web-services-cookbook/book
Testing and Monitoring Web-Services

<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

6. Go to the Port 8081 tab and see request and response messages, as shown in the following screenshot:

   ![Port 8081 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:

For More Information:

www.packtpub.com/spring-web-services-cookbook/book
How it works...

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.

See also

The recipe Integration testing using Spring-JUnit support discussed in this chapter.
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 3.1, *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/](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:

![HTTP Monitor Settings](image)

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:

![HTTP Monitor Request1](image)

For More Information:

4. Right-click **OrderServiceSoap11 | Generate Test Suite | OK.** Enter **OrderServiceSoap11 TestSuite.**

![Generate TestSuite dialog](image)

5. Double-click on **OrderServiceSoap11 TestSuite** on the navigator pane. Click Run the selected **TestCases.**

![HTTP Monitor](image)

For More Information:

www.packtpub.com/spring-web-services-cookbook/book
6. The following is the output when the test suite is run:

For More Information:  
www.packtpub.com/spring-web-services-cookbook/book
Testing and Monitoring Web-Services

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**.

For More Information:
www.packtpub.com/spring-web-services-cookbook/book
3. The following is the output of the test:
Testing and Monitoring Web-Services

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.

See also

The recipe Integration testing using Spring-JUnit support, discussed in this chapter.

For More Information:  
www.packtpub.com/spring-web-services-cookbook/book
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.