1.0.4

Preface

In the current age of Service Oriented Architectures, more and more people are using Web Services to connect previously unconnected systems. Initially, Web services were considered to be just another way to do a Remote Procedure Call (RPC). Over time however, people found out that there is a big difference between RPCs and Web services. Especially when interoperability with other platforms is important, it is often better to send encapsulated XML documents, containing all the data necessary to process the request. Conceptually, XML-based Web services are better off being compared to message queues rather than remoting solutions. Overall, XML should be considered the platform-neutral representation of data, the *interlingua* of SOA. When developing or using Web services, the focus should be on this XML, and not on Java.

Spring Web Services focusses on creating these document-driven Web services. Spring Web Services facilitates contract-first SOAP service development, allowing for the creation of flexible web services using one of the many ways to manipulate XML payloads. Spring-WS provides a powerful message dispatching framework, various XML marshalling techniques that can be used outside a Web service environment, a WS-Security solution that integrates with your existing application security solution, and a Client-side API that follows the familiar Spring template pattern.

This document provides a reference guide to Spring-WS's features. Since this document is still a work-in-progress, if you have any requests or comments, please post them on the support forums at http://forum.springframework.org/forumdisplay.php?f=39.

Part I. Introduction

This first part of the reference documentation is an overview of Spring Web Services and the underlying concepts. Spring-WS is then introduced, and the concepts behind contract-first Web service development are explained.

Chapter 1. What is Spring Web Services?

1.1. Introduction

Spring Web Services (Spring-WS) is a product of the Spring community focused on creating document-driven Web services. Spring Web Services aims to facilitate contract-first SOAP service development, allowing for the creation of flexible web services using one of the many ways to manipulate XML payloads. The product is based on Spring itself, which means you can use the Spring concepts such as dependency injection as an integral part of your Web service.

People use Spring-WS for many reasons, but most are drawn to it after finding alternative SOAP stacks lacking when it comes to following Web service best practices. Spring-WS makes the best practice an easy practice. This includes practices such as the WS-I basic profile, Contract-First development, and having a loose coupling between contract and implementation. The other key features of Spring Web services are:

Powerful mappings. You can distribute incoming XML requests to any object, depending on message payload, SOAP Action header, or an XPath expression.

XML API support. Incoming XML messages can be handled not only with standard JAXP APIs such as DOM, SAX, and StAX, but also JDOM, dom4j, XOM, or even marshalling technologies.

Flexible XML Marshalling. The Object/XML Mapping module in the Spring Web Services distribution supports JAXB 1 and 2, Castor, XMLBeans, JiBX, and XStream. And because it is a separate module, you can use it in non-Web services code as well.

Reuses your Spring expertise. Spring-WS uses Spring application contexts for all configuration, which should help Spring developers get up-to-speed nice and quickly. Also, the architecture of Spring-WS resembles that of Spring-MVC.

Supports WS-Security. WS-Security allows you to sign SOAP messages, encrypt and decrypt them, or authenticate against them.

Integrates with Acegi Security. The WS-Security implementation of Spring Web Services provides integration with Acegi Security. This means you can use your existing Acegi configuration for your SOAP service as well.

Built by Maven. This assists you in effectively reusing the Spring Web Services artifacts in your own Maven-based projects.

Apache license. You can confidently use Spring-WS in your project.

1.2. Runtime environment

Spring Web Services runs within a standard Java 1.3 Runtime Environment. It also supports Java 5.0, although the Java types which are specific to this release are packaged in a separate modules with the suffix "tiger" in their JAR filename. Note that the security module also requires Java 5.

Spring-WS consists of a number of modules, which are described in the remainder of this section.

- The XML module (`spring-xml.jar`) contains various XML support classes for Spring Web Services. This

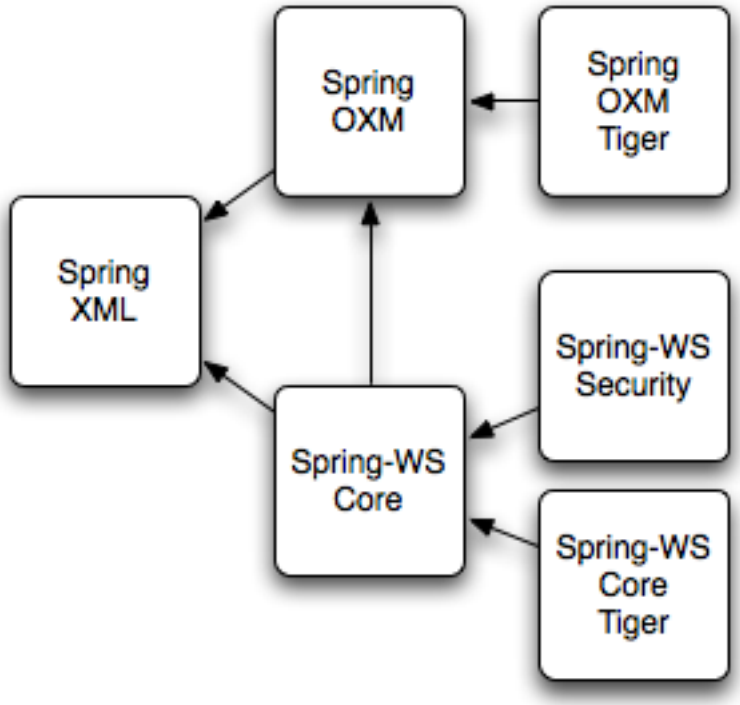
module is mainly intended for the Spring-WS framework itself, and not a Web service developers.

- The Core package (`spring-ws-core.jar` and `spring-ws-core-tiger.jar`) is the central part of the Spring's Web services functionality. It provides the central `WebServiceMessage` and `SoapMessage` interfaces, the server-side framework, with powerful message dispatching, and the various support classes for implementing Web service endpoints; and the client-side `WebServiceTemplate`.

- The Security package (`spring-ws-security.jar`) provides a WS-Security implementation that integrates with the core Web service package. It allows you to add principal tokens, sign, and decrypt and encrypt SOAP messages. Addtionally, it allows you to leverage your existing Acegi security implementation for authentication and authorization.

- The OXM package (`spring-oxm.jar` and `spring-oxm-tiger.jar`) provides integration for popular XML marshalling APIs, including JAXB 1 and 2. Using the OXM package means that you benefit from a unified exception hierarchy, and can wire up your favorite XML marshalling technology easily.

The following figure illustrates the Spring-WS modules and the dependencies between them. Arrows indicate dependencies, i.e. Spring-WS Core depends on Spring-XML and Spring-OXM.



Dependencies between Spring-WS modules

Chapter 2. Why Contract First?

2.1. Introduction

When creating Web services, there are two development styles: *Contract Last* and *Contract First*. When using a contract-last approach, you start with the Java code, and let the Web service contract (WSDL, see sidebar) be generated from that. When using contract-first, you start with the WSDL contract, and use Java to implement said contract.

> **What is WSDL?**
>
> WSDL stands for Web Services Description Language. A WSDL file is an XML document that describes a Web service. It specifies the location of the service and the operations (or methods) the service exposes. For more information about WSDL, refer to the WSDL specification, or read the WSDL tutorial

Spring-WS only supports the contract-first development style, and this section explains why.

2.2. Object/XML Impedance Mismatch

Similar to the field of ORM, where we have an Object/Relational impedance mismatch, there is a similar problem when converting Java objects to XML. At first glance, the O/X mapping problem appears simple: create an XML element for each Java object, converting all Java properties and fields to sub-elements or attributes. However, things are not as simple as they appear: there is a fundamental difference between hierarchical languages such as XML (and especially XSD) and the graph model of Java[1].

2.2.1. XSD extensions

In Java, the only way to change the behavior of a class is to subclass it, adding the new behavior to that subclass. In XSD, you can extend a data type by restricting it: that is, constraining the valid values for the elements and attributes. For instance, consider the following example:

```
<simpleType name="AirportCode">
  <restriction base="string">
      <pattern value="[A-Z][A-Z][A-Z]"/>
  </restriction>
</simpleType>
```

This type restricts a XSD string by ways of a regular expression, allowing only three upper case letters. If this type is converted to Java, we will end up with an ordinary `java.lang.String`; the regular expression is lost in the conversion process, because Java does not allow for these sorts of extensions.

2.2.2. Unportable types

One of the most important goals of a Web service is to be interoperable: to support multiple platforms such as Java, .NET, Python, etc. Because all of these languages have different class libraries, you must use some common, interlingual format to communicate between them. That format is XML, which is supported by all of

[1] Most of the contents in this section was inspired by [alpine] and [effective-enterprise-java].

these languages.

Because of this conversion, you must make sure that you use portable types in your service implementation. Consider, for example, a service that returns a `java.util.TreeMap`, like so:

```
public Map getFlights() {
  // use a tree map, to make sure it's sorted
  TreeMap map = new TreeMap();
  map.put("KL1117", "Stockholm");
  ...
  return map;
}
```

Undoubtedly, the contents of this map can be converted into some sort of XML, but since there is no *standard* way to describe a map in XML, it will be proprietary. Also, even if it can be converted to XML, many platforms do not have a data structure similar to the `TreeMap`. So when a .NET client accesses your Web service, it will probably end up with a `System.Collections.Hashtable`, which has different semantics.

This problem is also present when working on the client side. Consider the following XSD snippet, which describes a service contract:

```
<element name="GetFlightsRequest">
  <complexType>
    <all>
      <element name="departureDate" type="date"/>
      <element name="from" type="string"/>
      <element name="to" type="string"/>
    </all>
  </complexType>
</element>
```

This contract defines a request that takes an date, which is a XSD datatype representing a year, month, and day. If we call this service from Java, we will probably use either a `java.util.Date` or `java.util.Calendar`. However, both of these classes actually describe times, rather than dates. So, we will actually end up sending data that represents the fourth of April 2007 at midnight (`2007-04-04T00:00:00`), which is not the same as `2007-04-04`.

2.2.3. Cyclic graphs

Imagine we have the following simple class structure:

```
public class Flight {
  private String number;
  private List<Passenger> passengers;

  // getters and setters omitted
}

public class Passenger {
  private String name;
  private Flight flight;

  // getters and setters omitted
}
```

This is a cyclic graph: the `Flight` refers to the `Passenger`, which refers to the `Flight` again. Cyclic graphs like these are quite common in Java. If we took a naive approach to converting this to XML, we will end up with something like:

```
<flight number="KL1117">
  <passengers>
```

```
    <passenger>
      <name>Arjen Poutsma</name>
      <flight number="KL1117">
        <passengers>
          <passenger>
            <name>Arjen Poutsma</name>
            <flight number="KL1117">
              <passengers>
                <passenger>
                   <name>Arjen Poutsma</name>
                   ...
```

which will take a pretty long time to finish, because there is no stop condition for this loop.

One way to solve this problem is to use references to objects that were already marshalled, like so:

```
<flight number="KL1117">
  <passengers>
    <passenger>
      <name>Arjen Poutsma</name>
      <flight href="KL1117" />
    </passenger>
    ...
  </passengers>
</flight>
```

This solves the recursiveness problem, but introduces new ones. For one, you cannot use an XML validator to validate this structure. Another issue is that the standard way to use these references in SOAP (RPC/encoded) has been deprecated in favor of document/literal (see WS-I Basic Profile).

These are just a few of the problems when dealing with O/X mapping. It is important to respect these issues when writing Web services. The best way to respect them is to focus on the XML completely, while using Java as an implementation language. This is what contract-first is all about.

2.3. Contract-first versus Contract-last

Besides the Object/XML Mapping issues mentioned in the previous section, there are other reasons for preferring a contract-first development style.

2.3.1. Fragility

As mentioned earlier, the contract-last development style results in your web service contract (WSDL and your XSD) being generated from your Java contract (usually an interface). If you are using this approach, you will have no guarantee that the contract stays constant over time. Each time you change your Java contract and redeploy it, there might be subsequent changes to the web service contract.

Aditionally, not all SOAP stacks generate the same web service contract from a Java contract. This means changing your current SOAP stack for a different one (for whatever reason), might also change your web service contract.

When a web service contract changes, users of the contract will have to be instructed to obtain the new contract and potentially change their code to accommodate for any changes in the contract.

In order for a contract to be useful, it must remain constant for as long as possible. If a contract changes, you will have to contact all of the users of your service, and instruct them to get the new version of the contract.

2.3.2. Performance

When Java is automatically transformed into XML, there is no way to be sure as to what is sent across the wire. An object might reference another object, which refers to another, etc. In the end, half of the objects on the heap in your virtual machine might be converted into XML, which will result in slow response times.

When using contract-first, you explicitly describe what XML is sent where, thus making sure that it is exactly what you want.

2.3.3. Reusability

Defining your schema in a separate file allows you to reuse that file in different scenarios. If you define an AirportCode in a file called `airline.xsd`, like so:

```
<simpleType name="AirportCode">
    <restriction base="string">
        <pattern value="[A-Z][A-Z][A-Z]"/>
    </restriction>
</simpleType>
```

You can reuse this definition in other schemas, or even WSDL files, using an `import` statement.

2.3.4. Versioning

Even though a contract must remain constant for as long as possible, they *do* need to be changed sometimes. In Java, this typically results in a new Java interface, such as `AirlineService2`, and a (new) implementation of that interface. Of course, the old service must be kept around, because there might be clients who have not migrated yet.

If using contract-first, we can have a looser coupling between contract and implementation. Such a looser coupling allows us to implement both versions of the contract in one class. We could, for instance, use an XSLT stylesheet to convert any "old-style" messages to the "new-style" messages.

Chapter 3. Writing Contract-First Web Services

3.1. Introduction

This tutorial shows you how to write contract-first Web services, that is, developing web services that start with the XML Schema/WSDL contract first followed by the Java code second. Spring-WS focuses on this development style, and this tutorial will help you get started. Note that the first part of this tutorial contains almost no Spring-WS specific information: it is mostly about XML, XSD, and WSDL. The second part focusses on implementing this contract using Spring-WS .

The most important thing when doing contract-first Web service development is to try and think in terms of XML. This means that Java-language concepts are of lesser importance. It is the XML that is sent across the wire, and you should focus on that. The fact that Java is used to implement the Web service is an implementation detail. An important detail, but a detail nonetheless.

In this tutorial, we will define a Web service that is created by a Human Resources department. Clients can send holiday request forms to this service to book a holiday.

3.2. Messages

In this section, we will focus on the actual XML messages that are sent to and from the Web service. We will start out by determining what these messages look like.

3.2.1. Holiday

In the scenario, we have to deal with holiday requests, so it makes sense to determine what a holiday looks like in XML:

```
<Holiday xmlns="http://mycompany.com/hr/schemas">
    <StartDate>2006-07-03</StartDate>
    <EndDate>2006-07-07</EndDate>
</Holiday>
```

A holiday consists of a start date and an end date. We have also decided to use the standard ISO 8601 date format for the dates, because that will save a lot of parsing hassle. We have also added a namespace to the element, to make sure our elements can used within other XML documents.

3.2.2. Employee

There is also the notion of an employee in the scenario. Here is what it looks like in XML:

```
<Employee xmlns="http://mycompany.com/hr/schemas">
    <Number>42</Number>
    <FirstName>Arjen</FirstName>
    <LastName>Poutsma</LastName>
</Employee>
```

We have used the same namespace as before. If this `<Employee/>` element could be used in other scenarios, it might make sense to use a different namespace, such as `http://mycompany.com/employees/schemas`.

3.2.3. HolidayRequest

Both the holiday and employee element can be put in a `<HolidayRequest/>`:

```
<HolidayRequest xmlns="http://mycompany.com/hr/schemas">
    <Holiday>
        <StartDate>2006-07-03</StartDate>
        <EndDate>2006-07-07</EndDate>
    </Holiday>
    <Employee>
        <Number>42</Number>
        <FirstName>Arjen</FirstName>
        <LastName>Poutsma</LastName>
    </Employee>
</HolidayRequest>
```

The order of the two elements does not matter: `<Employee/>` could have been the first element just as well. What is important is that all of the data is there. In fact, the data is the only thing that is important: we are taking a *data-driven* approach.

3.3. Data Contract

Now that we have seen some examples of the XML data that we will use, it makes sense to formalize this into a schema. This data contract defines the message format we accept. There are four different ways of defining such a contract for XML:

- DTDs
- XML Schema (XSD)
- RELAX NG
- Schematron

DTDs have limited namespace support, so they are not suitable for Web services. Relax NG and Schematron certainly are easier than XML Schema. Unfortunately, they are not so widely supported across platforms. We will use XML Schema.

By far the easiest way to create an XSD is to infer it from sample documents. Any good XML editor or Java IDE offers this functionality. Basically, these tools use some sample XML documents, and generate a schema from it that validates them all. The end result certainly needs to be polished up, but it's a great starting point.

Using the sample described above, we end up with the following generated schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        targetNamespace="http://mycompany.com/hr/schemas"
        xmlns:hr="http://mycompany.com/hr/schemas">
    <xs:element name="HolidayRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="hr:Holiday"/>
                <xs:element ref="hr:Employee"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Holiday">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="hr:StartDate"/>
```

```
                <xs:element ref="hr:EndDate"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="StartDate" type="xs:NMTOKEN"/>
    <xs:element name="EndDate" type="xs:NMTOKEN"/>
    <xs:element name="Employee">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="hr:Number"/>
                <xs:element ref="hr:FirstName"/>
                <xs:element ref="hr:LastName"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Number" type="xs:integer"/>
    <xs:element name="FirstName" type="xs:NCName"/>
    <xs:element name="LastName" type="xs:NCName"/>
</xs:schema>
```

This generated schema obviously can be improved. The first thing to notice is that every type has a root-level element declaration. This means that the Web service should be able to accept all of these elements as data. This is not desirable: we only want to accept a `<HolidayRequest/>`. By removing the wrapping element tags (thus keeping the types), and inlining the results, we can accomplish this.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:hr="http://mycompany.com/hr/schemas"
        elementFormDefault="qualified"
        targetNamespace="http://mycompany.com/hr/schemas">
    <xs:element name="HolidayRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Holiday" type="hr:HolidayType"/>
                <xs:element name="Employee" type="hr:EmployeeType"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="HolidayType">
        <xs:sequence>
            <xs:element name="StartDate" type="xs:NMTOKEN"/>
            <xs:element name="EndDate" type="xs:NMTOKEN"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="EmployeeType">
        <xs:sequence>
            <xs:element name="Number" type="xs:integer"/>
            <xs:element name="FirstName" type="xs:NCName"/>
            <xs:element name="LastName" type="xs:NCName"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

The schema still has one problem: with a schema like this, you can expect the following messages to validate:

```
<HolidayRequest xmlns="http://mycompany.com/hr/schemas">
    <Holiday>
        <StartDate>this is not a date</StartDate>
        <EndDate>neither is this</EndDate>
    </Holiday>
    <!-- ... -->
</HolidayRequest>
```

Clearly, we must make sure that the start and end date are really dates. XML Schema has an excellent built-in `date` type which we can use. We also change the `NCNames` to `strings`. Finally, we change the `sequence` in `<HolidayRequest/>` to `all`. This tells the XML parser that the order of `<Holiday/>` and `<Employee/>` is not significant. Our final XSD now looks like this:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
        xmlns:hr="http://mycompany.com/hr/schemas"
        elementFormDefault="qualified"
        targetNamespace="http://mycompany.com/hr/schemas">
    <xs:element name="HolidayRequest">
        <xs:complexType>
            <xs:all>
                <xs:element name="Holiday" type="hr:HolidayType"/>                    ❶
                <xs:element name="Employee" type="hr:EmployeeType"/>
            </xs:all>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="HolidayType">
        <xs:sequence>
            <xs:element name="StartDate" type="xs:date"/>
            <xs:element name="EndDate" type="xs:date"/>                               ❷
        </xs:sequence>                                                                ❷
    </xs:complexType>
    <xs:complexType name="EmployeeType">
        <xs:sequence>
            <xs:element name="Number" type="xs:integer"/>
            <xs:element name="FirstName" type="xs:string"/>
            <xs:element name="LastName" type="xs:string"/>                            ❸
        </xs:sequence>                                                                ❸
    </xs:complexType>
</xs:schema>
```

❶ `all` tells the XML parser that the order of `<Holiday/>` and `<Employee/>` is not significant.

❷ We use the `xsd:date` data type, which consist of a year, month, and day, for `<StartDate/>` and `<EndDate/>`.

❸ `xsd:string` is used for the first and last name.

We store this file as `hr.xsd`.

3.4. Service contract

A service contract is generally expressed as a WSDL file. Note that in Spring-WS, *writing the WSDL by hand is not required*. Based on the XSD and some conventions, Spring-WS can create the WSDL for you, as explained in the section entitled Section 3.6, “Implementing the Endpoint”. You can skip to the next section if you want to; the remainder of this section will show you how to write your own WSDL by hand.

We start our WSDL with the standard preamble, and by importing our existing XSD. To separate the schema from the definition, we will use a separate namespace for the WSDL definitions: `http://mycompany.com/hr/definitions`.

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:schema="http://mycompany.com/hr/schemas"
                  xmlns:tns="http://mycompany.com/hr/definitions"
                  targetNamespace="http://mycompany.com/hr/definitions">
    <wsdl:types>
        <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xsd:import namespace="http://mycompany.com/hr/schemas" schemaLocation="hr.xsd"/>
        </xsd:schema>
    </wsdl:types>
```

Next, we add our messages based on the written schema types. We only have one message: one with the `<HolidayRequest/>` we put in the schema:

```
    <wsdl:message name="HolidayRequest">
        <wsdl:part element="schema:HolidayRequest" name="HolidayRequest"/>
    </wsdl:message>
```

We add the message to a port type as an operation:

```
    <wsdl:portType name="HumanResource">
        <wsdl:operation name="Holiday">
            <wsdl:input message="tns:HolidayRequest" name="HolidayRequest"/>
        </wsdl:operation>
    </wsdl:portType>
```

That finished the abstract part of the WSDL (the interface, as it were), and leaves the concrete part. The concrete part consists of a `binding`, which tells the client *how* to invoke the operations you've just defined; and a `service`, which tells it *where* to invoke it.

Adding a concrete part is pretty standard: just refer to the abstract part you defined previously, make sure you use *document/literal* for the `soap:binding` elements (`rpc/encoded` is deprecated), pick a `soapAction` for the operation (in this case `http://mycompany.com/RequestHoliday`, but any URI will do), and determine the `location` URL where you want request to come in (in this case `http://mycompany.com/humanresources`):

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:schema="http://mycompany.com/hr/schemas"
                  xmlns:tns="http://mycompany.com/hr/definitions"
                  targetNamespace="http://mycompany.com/hr/definitions">
    <wsdl:types>
        <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xsd:import namespace="http://mycompany.com/hr/schemas"                    ❶
                schemaLocation="hr.xsd"/>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="HolidayRequest">                                               ❷
        <wsdl:part element="schema:HolidayRequest" name="HolidayRequest"/>             ❸
    </wsdl:message>
    <wsdl:portType name="HumanResource">                                               ❹
        <wsdl:operation name="Holiday">
            <wsdl:input message="tns:HolidayRequest" name="HolidayRequest"/>           ❷
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HumanResourceBinding" type="tns:HumanResource">                ❹❺
        <soap:binding style="document"                                                 ❻
            transport="http://schemas.xmlsoap.org/soap/http"/>                         ❼
        <wsdl:operation name="Holiday">
            <soap:operation soapAction="http://mycompany.com/RequestHoliday"/>         ❽
            <wsdl:input name="HolidayRequest">
                <soap:body use="literal"/>                                             ❻
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HumanResourceService">
        <wsdl:port binding="tns:HumanResourceBinding" name="HumanResourcePort">        ❺
            <soap:address location="http://localhost:8080/holidayService/"/>           ❾
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

❶ We import the schema defined in Section 3.3, “Data Contract”.
❷ We define the `HolidayRequest` message, which gets used in the `portType`.
❸ The `HolidayRequest` type is defined in the schema.
❹ We define the `HumanResource` port type, which gets used in the `binding`.
❺ We define the `HumanResourceBinding` binding, which gets used in the `port`.
❻ We use a document/literal style.
❼ The literal `http://schemas.xmlsoap.org/soap/http` signifies a HTTP transport.
❽ The `soapAction` attribute signifies the `SOAPAction` HTTP header that will be sent with every request.
❾ The `http://localhost:8080/holidayService/` address is the URL where the Web service can be invoked.

This is the final WSDL. We will describe how to implement the resulting schema and WSDL in the next section.

3.5. Creating the project

In this section, we will be using Maven2 to create the initial project structure for us. Doing so is not required, but greatly reduces the amount of code we have to write to setup our HolidayService.

The following command creates a Maven2 web application project for us, using the Spring-WS archetype (that is, project template)

```
mvn archetype:create -DarchetypeGroupId=org.springframework.ws \
  -DarchetypeArtifactId=spring-ws-archetype \
  -DarchetypeVersion=1.0.4 \
  -DgroupId=com.mycompany.hr \
  -DartifactId=holidayService
```

This command will create a new directory called `holidayService`. In this directory, there is a `'src/main/webapp'` directory, which will contain the root of the WAR file. You will find the standard web application deployment descriptor `'WEB-INF/web.xml'` here, which defines a Spring-WS `MessageDispatcherServlet` and maps all incoming requests to this servlet:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
             http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">

    <display-name>MyCompany HR Holiday Service</display-name>

    <!-- take especial notice of the name of this servlet -->
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

In addition to the above `'WEB-INF/web.xml'` file, you will also need another, Spring-WS-specific configuration file, named `'WEB-INF/spring-ws-servlet.xml'`. This file contains all of the Spring-WS-specific beans such as `EndPoints`, `WebServiceMessageReceivers`, and suchlike, and is used to create a new Spring container. The name of this file is derived from the name of the attendant servlet (in this case `'spring-ws'`) with `'-servlet.xml'` appended to it. So if you defined a `MessageDispatcherServlet` with the name `'dynamite'`, the name of the Spring-WS-specific configuration file would be `'WEB-INF/dynamite-servlet.xml'`.

(You can see the contents of the `'WEB-INF/spring-ws-servlet.xml'` file for this example in ???.)

3.6. Implementing the Endpoint

In Spring-WS, you will implement *Endpoints* to handle incoming XML messages. There are two flavors of endpoints: message endpoints and payload endpoints. *Message endpoints* give access to the entire XML message, including SOAP headers. Typically, the endpoint will only be interested in the *payload* of the

message, that is the contents of the SOAP body. In that case, creating a *payload endpoint* makes more sense.

3.6.1. Handling the XML Message

In this sample application, we are going to use JDom to handle the XML message. We are also using XPath, because it allows us to select particular parts of the XML JDOM tree, without requiring strict schema conformance. We extend our endpoint from `AbstractJDomPayloadEndpoint`, because that will give us a JDOM element to execute the XPath queries on.

```
package com.mycompany.hr.ws;

import java.text.SimpleDateFormat;
import java.util.Date;

import com.mycompany.hr.service.HumanResourceService;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.xpath.XPath;
import org.springframework.ws.server.endpoint.AbstractJDomPayloadEndpoint;

public class HolidayEndpoint extends AbstractJDomPayloadEndpoint {

    private XPath startDateExpression;

    private XPath endDateExpression;

    private XPath nameExpression;

    private final HumanResourceService humanResourceService;

    public HolidayEndpoint(HumanResourceService humanResourceService) {                ❶
        this.humanResourceService = humanResourceService;
        Namespace namespace = Namespace.getNamespace("hr", "http://mycompany.com/hr/schemas");
        startDateExpression = XPath.newInstance("//hr:StartDate");
        startDateExpression.addNamespace(namespace);
        endDateExpression = XPath.newInstance("//hr:EndDate");
        endDateExpression.addNamespace(namespace);
        nameExpression = XPath.newInstance("concat(//hr:FirstName,' ',//hr:LastName)");
        nameExpression.addNamespace(namespace);
    }

    protected Element invokeInternal(Element holidayRequest) throws Exception {        ❷
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        Date startDate = dateFormat.parse(startDateExpression.valueOf(holidayRequest));
        Date endDate = dateFormat.parse(endDateExpression.valueOf(holidayRequest));
        String name = nameExpression.valueOf(holidayRequest);

        humanResourceService.bookHoliday(startDate, endDate, name);
        return null;
    }
}
```

❶ The `HolidayEndpoint` requires the `HumanResourceService` business service to operate, so we inject the dependency via the constructor. Next, we set up XPath expressions using the JDOM API. There are three expressions: `//hr:StartDate` for extracting the `<StartDate>` text value, `//hr:EndDate` for extracting the end date and `concat(//hr:FirstName,' ',//hr:LastName)` for extracting and concatenating the names of the employee.

❷ The `invokeInternal(..)` method is a template method, which gets passed with the `<HolidayRequest/>` element from the incoming XML message. We use the XPath expressions to extract the string values from the XML messages, and convert these values to `Date` objects using a `SimpleDateFormat`. With these values, we invoke a method on the business service. Typically, this will result in a database transaction being started, and some records being altered in the database. Finally, we return `null`, which indicates to Spring-WS that we do not want to send a response message. If we wanted a response message, we could have returned a JDOM Element that represents the payload of the response message.

Using JDOM is just one of the options to handle the XML: other options include DOM, dom4j, XOM, SAX, and StAX, but also marshalling techniques like JAXB, Castor, XMLBeans, JiBX, and XStream. We chose JDOM because it gives us access to the raw XML, and because it is based on classes (not interfaces and factory methods as with W3C DOM and dom4j), which makes the code less verbose. We use XPath because it is less fragile than marshalling technologies: we don't care for strict schema conformance, as long as we can find the dates and the name.

Because we use JDOM, we must add some dependencies to the Maven `pom.xml`, which is in the root of our project directory. Here is the relevant section of the POM:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-core</artifactId>
        <version>1.0.4</version>
    </dependency>
    <dependency>
        <groupId>jdom</groupId>
        <artifactId>jdom</artifactId>
        <version>1.0</version>
    </dependency>
    <dependency>
        <groupId>jaxen</groupId>
        <artifactId>jaxen</artifactId>
        <version>1.1</version>
    </dependency>
    <dependency>
        <groupId>javax.xml.soap</groupId>
        <artifactId>saaj-api</artifactId>
        <version>1.3</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.sun.xml.messaging.saaj</groupId>
        <artifactId>saaj-impl</artifactId>
        <version>1.3</version>
        <scope>runtime</scope>
    </dependency>
<dependencies>
```

Here is how we would configure these classes in our `spring-ws-servlet.xml` Spring XML configuration file:

```

<beans xmlns="http://www.springframework.org/schema/beans">

    <bean id="holidayEndpoint" class="com.mycompany.hr.ws.HolidayEndpoint">
        <constructor-arg ref="hrService"/>
    </bean>

    <bean id="hrService" class="com.mycompany.hr.service.StubHumanResourceService"/>

</beans>
```

3.6.2. Routing the Message to the Endpoint

Now that we have written an endpoint that handles the message, we must define how incoming messages are routed to that endpoint. In Spring-WS, this is the responsibility of an `EndpointMapping`. In this tutorial, we will route messages based on their content, by using a `PayloadRootQNameEndpointMapping`. Here is how we configure a `PayloadRootQNameEndpointMapping` in `spring-ws-servlet.xml`:

```

<bean class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
    <property name="mappings">
        <props>
            <prop key="{http://mycompany.com/hr/schemas}HolidayRequest">holidayEndpoint</prop>
        </props>
```

```
    </property>
    <property name="interceptors">
        <bean class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor"/>
    </property>
</bean>
```

This means that whenever an XML message is received with the namespace `http://mycompany.com/hr/schemas` and the `HolidayRequest` local name, it will be routed to the `holidayEndpoint`. (It also adds a `PayloadLoggingInterceptor`, that dumps incoming and outgoing messages to the log.)

3.7. Publishing the WSDL

Finally, we need to publish the WSDL. As stated in Section 3.4, “Service contract”, we don't need to write a WSDL ourselves; Spring-WS can generate one for us based on some conventions. Here is how we define the generation:

```
<bean id="holiday" class="org.springframework.ws.wsdl.wsdl11.DynamicWsdl11Definition">   ❶
  <property name="builder">
    <bean class="org.springframework.ws.wsdl.wsdl11.builder.XsdBasedSoap11Wsdl4jDefinitionBuilder">
      <property name="schema" value="/WEB-INF/hr.xsd"/>                                  ❷
      <property name="portTypeName" value="HumanResource"/>                              ❸
      <property name="locationUri" value="http://localhost:8080/holidayService/"/>       ❹
      <property name="targetNamespace" value="http://mycompany.com/hr/definitions"/>     ❺
    </bean>
  </property>
</bean>
```

❶ The bean id determines the URL where the WSDL can be retrieved. In this case, the bean id is `holiday`, which means that the WSDL can be retrieved as `holiday.wsdl` in the servlet context. The full URL will typically be `http://localhost:8080/holidayService/holiday.wsdl`.

❷ The `schema` property is set to the human resource schema we defined in Section 3.3, “Data Contract”: we simply placed the schema in the `WEB-INF` directory of the application.

❸ Next, we define the WSDL port type to be `HumanResource`.

❹ We set the location where the service can be reached: `http://localhost:8080/holidayService`. For development, this will suffice, but obviously we need to change this to `http://mycompany.com/humanresources` when going live. A common way to to accomplish this is to use a Spring `PropertyPlaceholderConfigurer`.

❺ Finally, we define the target namespace for the WSDL definition itself. Setting these is not required. If not set, we give the WSDL the same namespace as the schema.

You can create a WAR file using **mvn install**. If you deploy the application (to Tomcat, Jetty, etc.), and point your browser at this location, you will see the generated WSDL. This WSDL is ready to be used by clients, such as soapUI, or other SOAP frameworks.

That concludes this tutorial. The tutorial code can be found in the full distribution of Spring-WS. The next step would be to look at the echo sample application that is part of the distribution. After that, look at the airline sample, which is a bit more complicated, because it uses JAXB, WS-Security, Hibernate, and a transactional service layer. Finally, you can read the rest of the reference documentation.

Part II. Reference

This part of the reference documentation details the various components that comprise Spring Web Services. This includes a chapter that discusses the parts common to both client- and server-side WS, a chapter devoted to the specifics of writing server-side Web services, a chapter about using Web services on the client-side, and chapters on using WS-Security and Object/XML mapping.

Chapter 4. Shared components

In this chapter, we will explore the components which are shared between client- and server-side Spring-WS development. These interfaces and classes represent the building blocks of Spring-WS, so it is important to understand what they do, even if you do not use them directly.

4.1. Web service messages

4.1.1. `WebServiceMessage`

One of the core interfaces of Spring Web Services is the `WebServiceMessage`. This interface represents a protocol-agnostic XML message. The interface contains methods that provide access to the payload of the message, in the form of a `javax.xml.transform.Source` or a `javax.xml.transform.Result`. `Source` and `Result` are tagging interfaces that represent an abstraction over XML input and output. Concrete implementations wrap various XML representations, as indicated in the following table.

Source/Result implementation	Wraps XML representation
`javax.xml.transform.dom.DOMSource`	`org.w3c.dom.Node`
`javax.xml.transform.dom.DOMResult`	`org.w3c.dom.Node`
`javax.xml.transform.sax.SAXSource`	`org.xml.sax.InputSource` and `org.xml.sax.XMLReader`
`javax.xml.transform.sax.SAXResult`	`org.xml.sax.ContentHandler`
`javax.xml.transform.stream.StreamSource`	`java.io.File`, `java.io.InputStream`, or `java.io.Reader`
`javax.xml.transform.stream.StreamResult`	`java.io.File`, `java.io.OutputStream`, or `java.io.Writer`

In addition to reading from and writing to the payload, a Web service message can write itself to an output stream.

4.1.2. `SoapMessage`

The `SoapMessage` is a subclass of `WebServiceMessage`. It contains SOAP-specific methods, such as getting SOAP Headers, SOAP Faults, etc. Generally, your code should not be dependent on `SoapMessage`, because the content of the SOAP Body can be obtained via `getPayloadSource()` and `getPayloadResult()` in the `WebServiceMessage`. Only when it is necessary to perform SOAP-specific actions, such as adding a header, get an attachment, etc., should you need to cast `WebServiceMessage` to `SoapMessage`.

4.1.3. Message Factories

Concrete message implementations are created by a `WebServiceMessageFactory`. This factory can create an empty message, or read a message based on an input stream. There are two concrete implementations of `WebServiceMessageFactory`; one is based on SAAJ, the SOAP with Attachments API for Java, the other based on Axis 2's AXIOM, the AXis Object Model.

4.1.3.1. `SaajSoapMessageFactory`

The `SaajSoapMessageFactory` uses the SOAP with Attachments API for Java to create `SoapMessage` implementations. SAAJ is part of J2EE 1.4, so it should be supported under most modern application servers. Here is an overview of the SAAJ versions supplied by common application servers:

Application Server	SAAJ Version
BEA WebLogic 8	1.1
BEA WebLogic 9	1.1/1.2[a]
IBM WebSphere 6	1.2
SUN Glassfish 1	1.3

[a] Weblogic 9 has a known bug in the SAAJ 1.2 implementation: it implement all the 1.2 interfaces, but throws a `UnsupportedOperationException` when called. Spring Web Services has a workaround: it uses SAAJ 1.1 when operating on WebLogic 9.

Additionally, Java SE 6 includes SAAJ 1.3. You wire up a `SaajSoapMessageFactory` like so:

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

> **Note**
>
> SAAJ is based on DOM, the Document Object Model. This means that all SOAP messages are stored *in memory*. For larger SOAP messages, this may not be very performant. In that case, the `AxiomSoapMessageFactory` might be more applicable.

4.1.3.2. `AxiomSoapMessageFactory`

The `AxiomSoapMessageFactory` uses the AXis 2 Object Model to create `SoapMessage` implementations. AXIOM is based on StAX, the Streaming API for XML. StAX provides a pull-based mechanism for reading XML messages, which can be more efficient for larger messages.

To increase reading performance on the `AxiomSoapMessageFactory`, you can set the payloadCaching property to false (default is true). This will read the contents of the SOAP body directly from the stream. When this setting is enabled, the payload can only be read once. This means that you have to make sure that any preprocessing of the message does not consume it.

You use the `AxiomSoapMessageFactory` as follows:

```
<bean id="messageFactory" class="org.springframework.ws.soap.axiom.AxiomSoapMessageFactory">
    <property name="payloadCaching" value="true"/>
</bean>
```

4.1.3.3. SOAP 1.1 or 1.2

Both the `SaajSoapMessageFactory` and the `AxiomSoapMessageFactory` have a soapVersion property, where you can inject a `SoapVersion` constant. By default, the version is 1.1, but you an set it to 1.2 like so:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-2.0.xsd">

    <bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
        <property name="soapVersion">
            <util:constant static-field="org.springframework.ws.soap.SoapVersion.SOAP_12"/>
        </property>
    </bean>

</beans>
```

In the example above, we define a `SaajSoapMessageFactory` that only accepts SOAP 1.2 messages.

> **Caution**
>
> Even though both versions of SOAP are quite similar in format, the 1.2 version is not backwards compatible with 1.1 because it uses a different XML namespace. Other major differences between SOAP 1.1 and 1.2 include the different structure of a Fault, and the fact that `SOAPAction` HTTP headers are deprecated, which means that you cannot use the `SoapActionEndpointMapping` or the `SoapActionAnnotationMethodEndpointMapping`.
>
> One important thing to note with SOAP version numbers, or WS-* specification version numbers in general, is that the latest version of a specification is generally not the most popular version. For SOAP, this means that currently, the best version to use is 1.1. Version 1.2 might become more popular in the future, but currently 1.1 is the safest bet.

4.1.4. `MessageContext`

Typically, messages come in pairs: a request and a response. A request is created on the client-side, which is sent over some transport to the server-side, where a response is generated. This response gets sent back to the client, where it is read.

In Spring Web Services, such a conversation is contained in a `MessageContext`, which has properties to get request and response messages. On the client-side, the message context is created by the `WebServiceTemplate`. On the server-side, the message context is read from the transport-specific input stream. For example, in HTTP, it is read from the `HttpServletRequest` and the response is written back to the `HttpServletResponse`.

4.2. `TransportContext`

One of the key properties of the SOAP protocol is that it tries to be transport-agnostic. This is why, for instance, Spring-WS does not support mapping messages to endpoints by HTTP request URL, but rather by mesage content.

However, sometimes it is necessary to get access to the underlying transport, either on the client or server side. For this, Spring Web Services has the `TransportContext`. The transport context allows access to the underlying `WebServiceConnection`, which typically is a `HttpServletConnection` on the server side; or a `HttpUrlConnection` or `CommonsHttpConnection` on the client side. For example, you can obtain the IP address of the current request in a server-side endpoint or interceptor like so:

```
TransportContext context = TransportContextHolder.getTransportContext();
HttpServletConnection connection = (HttpServletConnection )context.getConnection();
HttpServletRequest request = connection.getHttpServletRequest();
String ipAddress = request.getRemoteAddr();
```

4.3. Handling XML With XPath

One of the best ways to handle XML is to use XPath. Quoting [effective-xml], item 35:

> XPath is a fourth generation declarative language that allows you to specify which nodes you want to process without specifying exactly how the processor is supposed to navigate to those nodes. XPath's data model is very well designed to support exactly what almost all developers want from XML. For instance, it merges all adjacent text including that in CDATA sections, allows values to be calculated that skip over comments and processing instructions` and include text from child and descendant elements, and requires all external entity references to be resolved. In practice, XPath expressions tend to be much more robust against unexpected but perhaps insignificant changes in the input document.
>
> —Elliotte Rusty Harold

Spring Web Services has two ways to use XPath within your application: the faster `XPathExpression` or the more flexible `XPathTemplate`.

4.3.1. `XPathExpression`

The `XPathExpression` is an abstraction over a compiled XPath expression, such as the Java 5 `javax.xml.xpath.XPathExpression`, or the Jaxen `XPath` class. To construct an expression in an application context, there is the `XPathExpressionFactoryBean`. Here is an example which uses this factory bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="nameExpression" class="org.springframework.xml.xpath.XPathExpressionFactoryBean">
        <property name="expression" value="/Contacts/Contact/Name"/>
    </bean>

    <bean id="myEndpoint" class="sample.MyXPathClass">
        <constructor-arg ref="nameExpression"/>
    </bean>

</beans>
```

The expression above does not use namespaces, but we could set those using the namespaces property of the factory bean. The expression can be used in the code as follows:

```
package sample;

public class MyXPathClass {

    private final XPathExpression nameExpression;

    public MyXPathClass(XPathExpression nameExpression) {
        this.nameExpression = nameExpression;
    }

    public void doXPath(Document document) {
        String name = nameExpression.evaluateAsString(document.getDocumentElement());
        System.out.println("Name: " + name);
    }

}
```

For a more flexible approach, you can use a `NodeMapper`, which is similar to the `RowMapper` in Spring's JDBC

support. The following example shows how we can use it:

```
package sample;

public class MyXPathClass  {

   private final XPathExpression contactExpression;

   public MyXPathClass(XPathExpression contactExpression) {
      this.contactExpression = contactExpression;
   }

   public void doXPath(Document document) {
      List contacts = nameExpression.evaluate(requestElement,
        new NodeMapper() {
           public Object mapNode(Node node, int nodeNum) throws DOMException {
              Element contactElement = (Element) node;
              Element nameElement = (Element) contactElement.getElementsByTagName("Name").item(0);
              Element phoneElement = (Element) contactElement.getElementsByTagName("Phone").item(0);
              return new Contact(nameElement.getTextContent(), phoneElement.getTextContent());
           }
        });
      // do something with list of Contact objects
   }
}
```

Similar to mapping rows in Spring JDBC's `RowMapper`, each result node is mapped using an anonymous inner class. In this case, we create a `Contact` object, which we use later on.

4.3.2. `XPathTemplate`

The `XPathExpression` only allows you to evaluate a single, pre-compiled expression. A more flexible, though slower, alternative is the `XpathTemplate`. This class follows the common template pattern used throughout Spring (JdbcTemplate, JmsTemplate, etc.). Here is an example:

```
package sample;

public class MyXPathClass {

    private XPathOperations template = new Jaxp13XPathTemplate();

    public void doXPath(Source source) {
        String name = template.evaluateAsString("/Contacts/Contact/Name", request);
        // do something with name
    }

}
```

Chapter 5. Creating a Web service with Spring-WS

5.1. Introduction

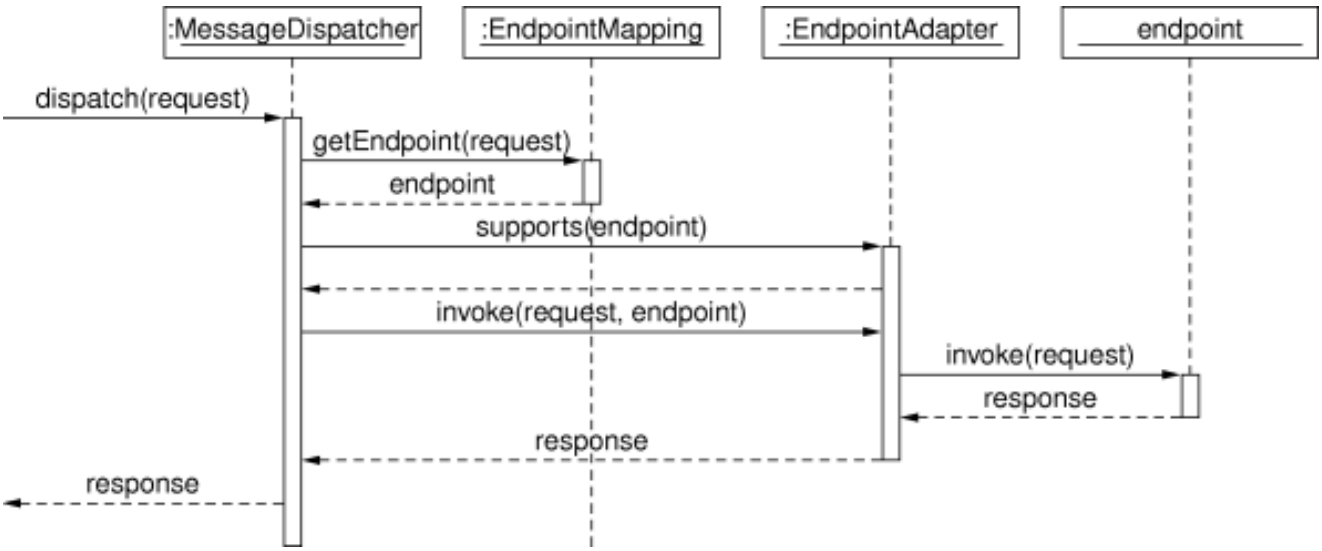
Spring-WS's server-side support is designed around a `MessageDispatcher` that dispatches incoming messages to endpoints, with configurable endpoint mappings, response generation, and endpoint interception. The simplest endpoint is a `PayloadEndpoint`, which just offers the `Source invoke(Source request)` method. You are of course free to implement this interface directly, but you will probably prefer to extend one of the included abstract implementations such as `AbstractDomPayloadEndpoint`, `AbstractSaxPayloadEndpoint`, and `AbstractMarshallingPayloadEndpoint`. Alternatively, there is a endpoint development that uses Java 5 annotations, such as `@Endpoint` for marking a POJO as endpoint, and marking a method with `@PayloadRoot` or `@SoapAction`.

Spring-WS's XML handling is extremely flexible. An endpoint can choose from a large amount of XML handling libraries supported by Spring-WS, including the DOM family (W3C DOM, JDOM, dom4j, and XOM), SAX or StAX for faster performance, XPath to extract information from the message, or even marshalling techniques (JAXB, Castor, XMLBeans, JiBX, or XStream) to convert the XML to objects and vice-versa.

5.2. The `MessageDispatcher`

The server-side of Spring-WS is designed around a central class that dispatches incoming XML messages to endpoints. Spring-WS's `MessageDispatcher` is extremely flexible, allowing you to use any sort of class as an endpoint, as long as it can be configured in the Spring IoC container. In a way, the message dispatcher resembles Spring's `DispatcherServlet`, the “Front Controller” used in Spring Web MVC.

The processing and dispatching flow of the `MessageDispatcher` is illustrated in the following sequence diagram.



The request processing workflow in Spring Web Services

When a `MessageDispatcher` is set up for use and a request comes in for that specific dispatcher, said `MessageDispatcher` starts processing the request. The list below describes the complete process a request goes through when handled by a `MessageDispatcher`:

1. An appropriate endpoint is searched for using the configured `EndpointMapping(s)`. If an endpoint is found, the invocation chain associated with the endpoint (preprocessors, postprocessors, and endpoints) will be executed in order to create a response.

2. An appropriate adapter is searched for the endpoint. The `MessageDispatcher` delegates to this adapter to invoke the endpoint.

3. If a response is returned, it is sent on its way. If no response is returned (which could be due to a pre- or post-processor intercepting the request, for example, for security reasons), no response is sent.

Exceptions that are thrown during handling of the request get picked up by any of the endpoint exception resolvers that are declared in the application context. Using these exception resolvers allows you to define custom behaviors (such as returning a SOAP Fault) in case such exceptions get thrown.

The `MessageDispatcher` has several properties, for setting endpoint adapters, mappings, exception resolvers. However, setting these properties is not required, since the dispatcher will automatically detect all of these types that are registered in the application context. Only when detection needs to be overriden, should these properties be set.

The message dispatcher operates on a message context, and not transport-specific input stream and output stream. As a result, transport specific requests need to read into a `MessageContext`. For HTTP, this is done with a `WebServiceMessageReceiverHandlerAdapter`, which is a Spring Web `HandlerInterceptor`, so that the `MessageDispatcher` can be wired in a standard `DispatcherServlet`. There is a more convenient way to do this, however, which is shown in the next section.

5.2.1. `MessageDispatcherServlet`

The `MessageDispatcherServlet` is a standard `Servlet` which conveniently extends from the standard Spring Web `DispatcherServlet`, and wraps a `MessageDispatcher`. As such, it combines the attributes of these into one: as a `MessageDispatcher`, it follows the same request handling flow as described in the previous section. As a servlet, the `MessageDispatcherServlet` is configured in the `web.xml` of your web application. Requests that you want the `MessageDispatcherServlet` to handle will have to be mapped using a URL mapping in the same `web.xml` file. This is standard Java EE servlet configuration; an example of such a `MessageDispatcherServlet` declaration and mapping can be found below.

```
<web-app>

    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

In the example above, all requests will be handled by the `'spring-ws' MessageDispatcherServlet`. This is only the first step in setting up Spring Web Services, because the various component beans used by the Spring-WS framework also need to be configured; this configuration consists of standard Spring XML `<bean/>` definitions. Because the `MessageDispatcherServlet` is a standard Spring `DispatcherServlet`, it will *look for a file named `[servlet-name]-servlet.xml`* in the `WEB-INF` directory of your web application and create the beans defined there in a Spring container. In the example above, that means that it looks for

`/WEB-INF/spring-ws-servlet.xml`'. This file will contain all of the SWS-specific beans such as endpoints, marshallers and suchlike.

5.2.1.1. Automatic WSDL exposure

The `MessageDispatcherServlet` will automatically detect any `WsdlDefinition` beans defined in it's Spring container. All such `WsdlDefinition` beans that are detected will also be exposed via a `WsdlDefinitionHandlerAdapter`; this is a very convenient way to expose your WSDL to clients simply by just defining some beans.

By way of an example, consider the following bean definition, defined in the Spring-WS framework's configuration file ('`/WEB-INF/[servlet-name]-servlet.xml`'). Take notice of the value of the bean's '`id`' attribute, because this will be used when exposing the WSDL.

```
<bean id="orders" class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">
    <constructor-arg value="/WEB-INF/wsdl/Orders.wsdl"/>
</bean>
```

The WSDL defined in the '`Orders.wsdl`' file can then be accessed via `GET` requests to a URL of the following form (substitute the host, port and servlet context path as appropriate).

```
http://localhost:8080/spring-ws/orders.wsdl
```

Another cool feature of the `MessageDispatcherServlet` (or more correctly the `WsdlDefinitionHandlerAdapter`) is that it is able to transform the value of the '`location`' of all the WSDL that it exposes to reflect the URL of the incoming request.

Please note that this '`location`' transformation feature is *off* by default.To switch this feature on, you just need to specify an initialization parameter to the `MessageDispatcherServlet`, like so:

```
<web-app>

    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
        <init-param>
            <param-name>transformWsdlLocations</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Consult the class-level Javadoc on the `WsdlDefinitionHandlerAdapter` class which explains the whole transformation process in more detail.

5.2.1.1.1. Exposing a static WSDL

As indicated above, a static WSDL file can be exposed by using the `SimpleWsdl11Definition`. Simply wire it up, and give it a `Resource` for the wsdl property, or use the contructor, as shown in the example above.

5.2.1.1.2. Dynamically creating a WSDL from an XSD

As shown in Section 3.7, “Publishing the WSDL”, Spring Web Services can generate a WSDL file from a XSD

schema, using conventions. The next application context snippet shows how to create such a dynamic WSDL file:

```
<bean id="holiday" class="org.springframework.ws.wsdl.wsdl11.DynamicWsdl11Definition">
  <property name="builder">
    <bean class="org.springframework.ws.wsdl.wsdl11.builder.XsdBasedSoap11Wsdl4jDefinitionBuilder">
      <property name="schema" value="/WEB-INF/xsd/Orders.xsd"/>
      <property name="portTypeName" value="Orders"/>
      <property name="locationUri" value="http://localhost:8080/ordersService/"/>
    </bean>
  </property>
</bean>
```

The `DynamicWsdl11Definition` uses a `Wsdl11DefinitionBuilder` implementation to generate a WSDL the first time it is requested. Typically, we use a `XsdBasedSoap11Wsdl4jDefinitionBuilder`, which builds a WSDL from a XSD schema. This builder iterates over all `element` elements found in the schema, and creates a `message` for elements that end with the defined request or response suffix. The default request suffix is `Request`; the default response suffix is `Response`, though these can be changed by setting the requestSuffix and responseSuffix properties, respectively. Next, the builder combines the request and response messages into a WSDL `operations`, and builds a `portType` based on the operations.

For instance, if our `Orders.xsd` schema defines the `GetOrdersRequest` and `GetOrdersResponse` elements, the `XsdBasedSoap11Wsdl4jDefinitionBuilder` will create a `GetOrdersRequest` and `GetOrdersResponse` message, and a `GetOrders` operation, which is put in a `Orders` port type.

5.2.2. Wiring up Spring-WS in a `DispatcherServlet`

As an alternative to the `MessageDispatcherServlet`, you can wire up a `MessageDispatcher` in a standard, Spring-Web MVC `DispatcherServlet`. By default, the `DispatcherServlet` can only delegate to `Controllers`, but we can instruct it to delegate to a `MessageDispatcher` by adding a `WebServiceMessageReceiverHandlerAdapter` to the servlet's web application context:

```
<beans>

    <bean class="org.springframework.ws.transport.http.WebServiceMessageReceiverHandlerAdapter"/>

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="defaultHandler" ref="messageDispatcher"/>
    </bean

    <bean id="messageDispatcher" class="org.springframework.ws.server.MessageDispatcher"/>

    ...

    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
```

Note that by explicitely adding the `WebServiceMessageReceiverHandlerAdapter`, the dispatcher servlet does not load the default adapters, and is unable to handle standard Spring-MVC `Controllers`. Therefore, we add the `SimpleControllerHandlerAdapter` at the end.

5.3. Endpoints

Endpoints are the central concept in Spring-WS's server-side support. Endpoints provide access to the application behavior which is typically defined by a business service interface. An endpoint interprets the XML request message and uses that input to invoke a method on the business service (typically). The result of that service invocation is represented as a response message. Spring-WS has a wide variety of endpoints, using various ways to handle the XML message, and to create a response.

The basis for most endpoints in Spring Web Services is the `org.springframework.ws.server.endpoint.PayloadEndpoint` interface, the source code of which is listed below.

```
public interface PayloadEndpoint {

    /**
     * Invokes an operation.
     */
    Source invoke(Source request) throws Exception;
}
```

As you can see, the `PayloadEndpoint` interface defines a single method that is invoked with the XML payload of a request (typically the contents of the SOAP Body, see Section 4.1.2, "`SoapMessage`"). The returned `Source`, if any, is stored in the response XML message. While the `PayloadEndpoint` interface is quite abstract, Spring-WS offers a lot of endpoint implementations out of the box that already contain a lot of the functionality you might need. The `PayloadEndpoint` interface just defines the most basic responsibility required of every endpoint; namely handling a request and returning a response.

Alternatively, there is the `MessageEndpoint`, which operates on a whole `MessageContext` rather than just the payload. Typically, your code should not be dependent on messages, because the payload should contain the information of interest. Only when it is necessary to perform actions on the message as a whole, such as adding a SOAP header, get an attachment, and so forth, should you need to implement `MessageEndpoint`, though these actions are usually performed in an endpoint interceptor.

5.3.1. `AbstractDomPayloadEndpoint` and other DOM endpoints

One of the most basic ways to handle the incoming XML payload is by using a DOM (Document Object Model) API. By extending from `AbstractDomPayloadEndpoint`, you can use the org.w3c.dom.Element and related classes to handle the request and create the response. When using the `AbstractDomPayloadEndpoint` as the baseclass for your endpoints you only have to override the `invokeInternal(Element, Document)` method, implement your logic, and return an `Element` if a response is necessary. Here is a short example consisting of a class and a declaration in the application context.

```
package samples;

public class SampleEndpoint extends AbstractDomPayloadEndpoint {

    private String responseText;

    public SampleEndpoint(String responseText) {
        this.responseText = responseText;
    }

    protected Element invokeInternal(
            Element requestElement,
            Document document) throws Exception {
        String requestText = requestElement.getTextContent();
        System.out.println("Request text: " + requestText);

        Element responseElement = document.createElementNS("http://samples", "response");
        responseElement.setTextContent(responseText);
        return responseElement;
    }
}
```

```
<bean id="sampleEndpoint" class="samples.SampleEndpoint">
    <constructor-arg value="Hello World!"/>
</bean>
```

The above class and the declaration in the application context are all you need besides setting up an endpoint mapping (see the section entitled Section 5.4, “Endpoint mappings”) to get this very simple endpoint working. The SOAP message handled by this endpoint will look something like:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <request xmlns="http://samples">
            Hello
        </request>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Though it could also handle the following Plain Old XML (POX) message, since we are only working on the *payload* of the message, and do not care whether it is SOAP or POX.

```
<request xmlns="http://samples">
    Hello
</request>
```

The SOAP response looks like:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <response xmlns="http://samples">
            Hello World!
        </response>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Besides the `AbstractDomPayloadEndpoint`, which uses W3C DOM, there are other base classes which use alternative DOM APIs. Spring Web Services supports most DOM APIs, so that you can use the one you are familiar with. For instance, the `AbstractJDomPayloadEndpoint` allows you to use JDOM, and the `AbstractXomPayloadEndpoint` uses XOM to handle the XML. All of these endpoints have an `invokeInternal` method similar to above. Also, consider using Spring-WS's XPath support to extract the information you need out of the payload. (See the section entitled Section 4.3, “Handling XML With XPath” for details.)

5.3.2. `AbstractMarshallingPayloadEndpoint`

Rather than handling XML directly using DOM, you can use marshalling to convert the payload of the XML message into a Java Object. Spring Web Services offers the `AbstractMarshallingPayloadEndpoint` for this purpose, which is built on the marshalling abstraction described in Chapter 8, *Marshalling XML using O/X Mappers*. The `AbstractMarshallingPayloadEndpoint` has two properties: marshaller and unmarshaller, in which you can inject in the constructor or by setters.

When extending from `AbstractMarshallingPayloadEndpoint`, you have to override the `invokeInternal(Object)` method, where the passed `Object` represents the unmarshalled request payload, and return an `Object` that will be marshalled into the response payload. Here is an example:

```
package samples;

import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;

public class MarshallingOrderEndpoint extends AbstractMarshallingPayloadEndpoint{

    private final OrderService orderService;

    public SampleMarshallingEndpoint(OrderService orderService, Marshaller marshaller) {
        super(marshaller);
        this.orderService = orderService;
```

```
    }

    protected Object invokeInternal(Object request) throws Exception {
        OrderRequest orderRequest = (OrderRequest) request;
        Order order = orderService.getOrder(orderRequest.getId());
        return order;
    }
}
```

```
<beans>
    <bean id="orderEndpoint" class="samples.MarshallingOrderEndpoint">
        <constructor-arg ref="orderService"/>
        <constructor-arg ref="marshaller"/>
    </bean>

    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>samples.OrderRequest</value>
                <value>samples.Order</value>
            </list>
        </property>
    </bean>

    <bean id="orderService" class="samples.DefaultOrderService"/>

    <!-- Other beans, such as the endpoint mapping -->
</beans>
```

In this sample, we configure a Jaxb2Marshaller for the `OrderRequest` and `Order` classes, and inject that marshaller together with the `DefaultOrderService` into our endpoint. This business service is not shown, but it is a normal transactional service, probably using DAOs to obtain data from a database. In the `invokeInternal` method, we cast the request object to an `OrderRequest` object, which is the JAXB object representing the payload of the request. Using the identifier of that request, we obtain an order from our business service and return it. The returned object is marshalled into XML, and used as the payload of the response message. The SOAP request handled by this endpoint will look like:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <orderRequest xmlns="http://samples" id="42"/>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The resulting response will be something like:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <order xmlns="http://samples" id="42">
            <item id="100">
                <quantity>1</quantity>
                <price>20.0</price>
            </item>
            <item id="101">
                <quantity>1</quantity>
                <price>10.0</price>
            </item>
        </order>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Instead of JAXB 2, we could have used any of the other marshallers described in Chapter 8, *Marshalling XML using O/X Mappers*. The only thing that would change in the above example is the configuration of the `marshaller` bean.

5.3.3. `@Endpoint`

The previous two programming models were based on inheritance, and handled individual XML messages. Spring Web Services offer another endpoint with which you can aggregate multiple handling into one controller, thus grouping functionality together. This model is based on annotations, so you can use it only with Java 5 and higher. Here is an example that uses the same marshalled objects as above:

```
package samples;

import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;

@Endpoint
public class AnnotationOrderEndpoint {
    private final OrderService orderService;

    public AnnotationOrderEndpoint(OrderService orderService) {
        this.orderService = orderService;
    }

    @PayloadRoot(localPart = "orderRequest", namespace = "http://samples")
    public Order getOrder(OrderRequest orderRequest) {
        return orderService.getOrder(orderRequest.getId());
    }

    @PayloadRoot(localPart = "order", namespace = "http://samples")
    public void order(Order order) {
        orderService.createOrder(order);
    }

}
```

By annotating the class with `@Endpoint`, you mark it as a Spring-WS endpoint. Because the endpoint class can have multiple request handling methods, we need to instruct Spring-WS which method to invoke for which request. This is done using the `@PayloadRoot` annotation: the `getOrder` method will be invoked for requests with a `orderRequest` local name and a `http://samples` namespace URI; the `order` method for requests with a `order` local name. For more information about these annotations, refer to Section 5.4.3, "`MethodEndpointMapping`". We also need to configure Spring-WS to support the JAXB objects `OrderRequest` and `Order` by defining a `Jaxb2Marshaller`:

```
<beans>

    <bean id="orderEndpoint" class="samples.AnnotationOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>

    <bean id="orderService" class="samples.DefaultOrderService"/>

    <bean class="org.springframework.ws.server.endpoint.adapter.GenericMarshallingMethodEndpointAdapter">
        <constructor-arg ref="marshaller"/>
    </bean>

    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>samples.OrderRequest</value>
                <value>samples.Order</value>
            </list>
        </property>
    </bean>

    <bean class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodEndpointMapping"/>

</beans>
```

The `GenericMarshallingMethodEndpointAdapter` converts the incoming XML messages to marshalled objects used as parameters and return value; the `PayloadRootAnnotationMethodEndpointMapping` is the

mapping that detects and handles the `@PayloadRoot` annotations.

5.3.3.1. `@XPathParam`

As an alternative to using marshalling, we could have used XPath to extract the information out of the incoming XML request. Spring-WS offers another annotation for this purpose: `@XPathParam`. You simply annotate one or more method parameter with this annotation (each), and each such annotated parameter will be bound to the evaluation of that annotation. Here is an example:

```
package samples;

import javax.xml.transform.Source;

import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.XPathParam;

@Endpoint
public class AnnotationOrderEndpoint {

    private final OrderService orderService;

    public AnnotationOrderEndpoint(OrderService orderService) {
        this.orderService = orderService;
    }

    @PayloadRoot(localPart = "orderRequest", namespace = "http://samples")
    public Source getOrder(@XPathParam("/s:orderRequest/@id") double orderId) {
        Order order = orderService.getOrder((int) orderId);
        // create Source from order and return it
    }

}
```

Since we use the prefix 's' in our XPath expression, we must bind it to the `http://samples` namespace:

```
<beans>
    <bean id="orderEndpoint" class="samples.AnnotationOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>

    <bean id="orderService" class="samples.DefaultOrderService"/>

    <bean class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodEndpointMapping"/>

    <bean class="org.springframework.ws.server.endpoint.adapter.XPathParamAnnotationMethodEndpointAdapter">
        <property name="namespaces">
            <props>
                <prop key="s">http://samples</prop>
            </props>
        </property>
    </bean>

</beans>
```

Using the `@XPathParam`, you can bind to all the data types supported by XPath:

- boolean or `Boolean`
- double or `Double`
- `String`
- `Node`
- `NodeList`

5.4. Endpoint mappings

The endpoint mapping is responsible for mapping incoming messages to appropriate endpoints. There are some endpoint mappings you can use out of the box, for example, the `PayloadRootQNameEndpointMapping` or the `SoapActionEndpointMapping`, but let's first examine the general concept of an `EndpointMapping`.

An `EndpointMapping` delivers a `EndpointInvocationChain`, which contains the endpoint that matches the incoming request, and may also contain a list of endpoint interceptors that will be applied to the request and response. When a request comes in, the `MessageDispatcher` will hand it over to the endpoint mapping to let it inspect the request and come up with an appropriate `EndpointInvocationChain`. Then the `MessageDispatcher` will invoke the endpoint and any interceptors in the chain.

The concept of configurable endpoint mappings that can optionally contain interceptors (which can manipulate the request or the response, or both) is extremely powerful. A lot of supporting functionality can be built into custom `EndpointMapping`s. For example, there could be a custom endpoint mapping that chooses an endpoint not only based on the contents of a message, but also on a specific SOAP header (or indeed multiple SOAP headers).

Most endpoint mappings inherit from the `AbstractEndpointMapping`, which offers an 'interceptors' property, which is the list of interceptors to use. `EndpointInterceptors` are discussed in Section 5.4.4, “Intercepting requests - the `EndpointInterceptor` interface”. Additionally, there is the 'defaultEndpoint', which is the default endpoint to use, when this endpoint mapping does not result in a matching endpoint.

5.4.1. `PayloadRootQNameEndpointMapping`

The `PayloadRootQNameEndpointMapping` will use the qualified name of the root element of the request payload to determine the endpoint that handles it. A qualified name consists of a *namespace URI* and a *local part*, the combination of which should be unique within the mapping. Here is an example:

```
<beans>

    <!-- no 'id' required, EndpointMapping beans are automatically detected by the MessageDispatcher -->
    <bean id="endpointMapping" class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMap
        <property name="mappings">
            <props>
                <prop key="{http://samples}orderRequest">getOrderEndpoint</prop>
                <prop key="{http://samples}order">createOrderEndpoint</prop>
            </props>
        </property>
    </bean>

    <bean id="getOrderEndpoint" class="samples.GetOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>

    <bean id="createOrderEndpoint" class="samples.CreateOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>
<beans>
```

The qualified name is expressed as `{` + namespace URI + `}` + local part. Thus, the endpoint mapping above routes requests for which have a payload root element with namespace `http://samples` and local part `orderRequest` to the `'getOrderEndpoint'`. Requests with a local part `order` will be routed to the `'createOrderEndpoint'`.

5.4.2. `SoapActionEndpointMapping`

Rather than base the routing on the contents of the message with the `PayloadRootQNameEndpointMapping`, you can use the `SOAPAction` HTTP header to route messages. Every client sends this header when making a SOAP request, and the header value used for a request is defined in the WSDL. By making the `SOAPAction` unique per operation, you can use it as a discriminator. Here is an example:

```
<beans>
    <bean id="endpointMapping" class="org.springframework.ws.soap.server.endpoint.mapping.SoapActionEndpointMapp
        <property name="mappings">
            <props>
                <prop key="http://samples/RequestOrder">getOrderEndpoint</prop>
                <prop key="http://samples/CreateOrder">createOrderEndpoint</prop>
            </props>
        </property>
    </bean>

    <bean id="getOrderEndpoint" class="samples.GetOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>

    <bean id="createOrderEndpoint" class="samples.CreateOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>
</beans>
```

The mapping above routes requests which have a `SOAPAction` of `http://samples/RequestOrder` to the `'getOrderEndpoint'`. Requests with `http://samples/CreateOrder` will be routed to the `'createController'`.

> **Caution**
>
> Note that using SOAP Action headers is SOAP 1.1-specific, so it cannot be used when using Plain Old XML, nor with SOAP 1.2.

5.4.3. `MethodEndpointMapping`

As explained in Section 5.3.3, "`@Endpoint`", the `@Endpoint` style allows you to handle multiple requests in one endpoint class. This is the responsibility of the `MethodEndpointMapping`. Similar to the endpoint mapping described above, this mapping determines which method is to be invoked for an incoming request message.

There are two endpoint mappings that can direct requests to methods: the `PayloadRootAnnotationMethodEndpointMapping` and the `SoapActionAnnotationMethodEndpointMapping`, both of which are very similar to their non-method counterparts described above.

The `PayloadRootAnnotationMethodEndpointMapping` uses the `@PayloadRoot` annotation, with the `localPart` and `namespace` elements, to mark methods with a particular qualified name. Whenever a message comes in which has this qualified name for the payload root element, the method will be invoked. For an example, see above.

Alternatively, the `SoapActionAnnotationMethodEndpointMapping` uses the `@SoapAction` annotation to mark methods with a particular SOAP Action. Whenever a message comes in which has this `SOAPAction` header, the method will be invoked.

5.4.4. Intercepting requests - the `EndpointInterceptor` interface

The endpoint mapping mechanism has the notion of endpoint interceptors. These can be extremely useful when you want to apply specific functionality to certain requests, for example, dealing with security-related SOAP

headers, or the logging of request and response message.

Interceptors located in the endpoint mapping must implement the `EndpointInterceptor` interface from the org.springframework.ws.server package. This interface defines three methods, one that can be used for handling the request message *before* the actual endpoint will be executed, one that can be used for handling a normal response message, and one that can be used for handling fault messages, both of which will be called *after* the endpoint is executed. These three methods should provide enough flexibility to do all kinds of pre- and post-processing.

The `handleRequest(..)` method on the interceptor returns a boolean value. You can use this method to interrupt or continue the processing of the invocation chain. When this method returns `true`, the endpoint execution chain will continue, when it returns `false`, the `MessageDispatcher` interprets this to mean that the interceptor itself has taken care of things and does not continue executing the other interceptors and the actual endpoint in the invocation chain. The `handleResponse(..)` and `handleFault(..)` methods also have a boolean return value. When these methods return `false`, the response will not be sent back to the client.

There are a number of standard `EndpointInterceptor` implementations you can use in your Web service. Additionally, there is the `XwsSecurityInterceptor`, which is described in Section 7.2, "`XwsSecurityInterceptor`".

5.4.4.1. `PayloadLoggingInterceptor` and `SoapEnvelopeLoggingInterceptor`

When developing a Web service, it can be useful to log the incoming and outgoing XML messages. SWS facilitates this with the `PayloadLoggingInterceptor` and `SoapEnvelopeLoggingInterceptor` classes. The former logs just the payload of the message to the Commons Logging Log; the latter logs the entire SOAP envelope, including SOAP headers. The following example shows you how to define them in an endpoint mapping:

```
<beans>
    <bean id="endpointMapping"
        class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
        <property name="interceptors">
            <list>
                <ref bean="loggingInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <props>
                <prop key="{http://samples}orderRequest">getOrderEndpoint</prop>
                <prop key="{http://samples}order">createOrderEndpoint</prop>
            </props>
        </property>
    </bean>

    <bean id="loggingInterceptor"
    class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor"/>
</beans>
```

Both of these interceptors have two properties: 'logRequest' and 'logResponse', which can be set to `false` to disable logging for either request or response messages.

5.4.4.2. `PayloadValidatingInterceptor`

One of the benefits of using a contract-first development style is that we can use the schema to validate incoming and outgoing XML messages. Spring-WS facilitates this with the `PayloadValidatingInterceptor`. This interceptor requires a reference to one or more W3C XML or RELAX NG schemas, and can be set to validate requests or responses, or both.

Note

Note that request validation may sound like a good idea, but makes the resulting Web service very strict. Usually, it is not really important whether the request validates, only if the endpoint can get sufficient information to fullfill a request. Validating the response *is* a good idea, because the endpoint should adhere to its schema. Remember Postel's Law: “Be conservative in what you do; be liberal in what you accept from others.”

Here is an example that uses the `PayloadValidatingInterceptor`; in this example, we use the schema in `/WEB-INF/orders.xsd` to validate the response, but not the request. Note that the `PayloadValidatingInterceptor` can also accept multiple schemas using the schemas property.

```
<bean id="validatingInterceptor"
        class="org.springframework.ws.soap.server.endpoint.interceptor.PayloadValidatingInterceptor">
    <property name="schema" value="/WEB-INF/orders.xsd"/>
    <property name="validateRequest" value="false"/>
    <property name="validateResponse" value="true"/>
</bean>
```

5.4.4.3. `PayloadTransformingInterceptor`

To transform the payload to another XML format, Spring Web Services offers the `PayloadTransformingInterceptor`. This endpoint interceptor is based on XSLT stylesheets, and is especially useful when supporting multiple versions of a Web service: you can transform the older message format to the newer format. Here is an example to use the `PayloadTransformingInterceptor`:

```
<bean id="transformingInterceptor"
        class="org.springframework.ws.server.endpoint.interceptor.PayloadTransformingInterceptor">
    <property name="requestXslt" value="/WEB-INF/oldRequests.xslt"/>
    <property name="requestXslt" value="/WEB-INF/oldResponses.xslt"/>
</bean>
```

We are simply transforming requests using `/WEB-INF/oldRequests.xslt`, and response messages using `/WEB-INF/oldResponses.xslt`. Note that, since endpoint interceptors are registered at the endpoint mapping level, you can simply create a endpoint mapping that applies to the "old style" messages, and add the interceptor to that mapping. Hence, the transformation will apply only to these "old style" message.

5.5. Handling Exceptions

Spring-WS provides `EndpointExceptionResolvers` to ease the pain of unexpected exceptions occurring while your message is being processed by an endpoint which matched the request. Endpoint exception resolvers somewhat resemble the exception mappings that can be defined in the web application descriptor `web.xml`. However, they provide a more flexible way to handle exceptions. They provide information about what endpoint was invoked when the exception was thrown. Furthermore, a programmatic way of handling exceptions gives you many more options for how to respond appropriately. Rather than expose the innards of your application by giving an exception and stack trace, you can handle the exception any way you want, for example by returning a SOAP fault with a specific fault code and string.

Endpoint exception resolvers are automatically picked up by the `MessageDispatcher`, so no explicit configuration is necessary.

Besides implementing the `EndpointExceptionResolver` interface, which is only a matter of implementing the `resolveException(MessageContext, endpoint, Exception)` method, you may also use one of the provided

implementations. The simplest implementation is the `SimpleSoapExceptionResolver`, which just creates a SOAP 1.1 Server or SOAP 1.2 Receiver Fault, and uses the exception message as the fault string. The `SimpleSoapExceptionResolver` is the default, but it can be overriden by explicitly adding another resolver.

5.5.1. `SoapFaultMappingExceptionResolver`

The `SoapFaultMappingExceptionResolver` is a more sophisticated implementation. This resolver enables you to take the class name of any exception that might be thrown and map it to a SOAP Fault, like so:

```
<beans>
    <bean id="exceptionResolver"
        class="org.springframework.ws.soap.server.endpoint.SoapFaultMappingExceptionResolver">
        <property name="defaultFault" value="SERVER">
        </property>
        <property name="exceptionMappings">
            org.springframework.oxm.ValidationFailureException=CLIENT,Invalid request
        </property>
    </bean>
</beans>
```

The key values and default endpoint use the format `faultCode,faultString,locale`, where only the fault code is required. If the fault string is not set, it will default to the exception message. If the language is not set, it will default to English. The above configuration will map exceptions of type `ValidationFailureException` to a client-side SOAP Fault with a fault string `"Invalid request"`, as can be seen in the following response:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:Client</faultcode>
           <faultstring>Invalid request</faultstring>
       </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If any other exception occurs, it will return the default fault: a server-side fault with the exception message as fault string.

5.5.2. `SoapFaultAnnotationExceptionResolver`

Finally, it is also possible to annotate exception classes with the `@SoapFault` annotation, to indicate the SOAP Fault that should be returned whenever that exception is thrown. In order for these annotations to be picked up, you need to add the `SoapFaultAnnotationExceptionResolver` to your application context. The elements of the annotation include a fault code enumeration, fault string or reason, and language. Here is an example exception:

```
package samples;

import org.springframework.ws.soap.server.endpoint.annotation.FaultCode;
import org.springframework.ws.soap.server.endpoint.annotation.SoapFault;

@SoapFault(faultCode = FaultCode.SERVER)
public class MyBusinessException extends Exception {

    public MyClientException(String message) {
        super(message);
    }
}
```

Whenever the `MyBusinessException` is thrown with the constructor string `"Oops!"` during endpoint invocation, it will result in the following response:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

```
    <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:Server</faultcode>
           <faultstring>Oops!</faultstring>
       </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Chapter 6. Using Spring Web Services on the Client

6.1. Introduction

Spring-WS provides a client-side Web service API that allows for consistent, XML-driven access to Web services. It also caters for the use of marshallers and unmarshallers so that your service tier code can deal exclusively with Java objects.

The org.springframework.ws.client.core package provides the core functionality for using the client-side access API. It contains template classes that simplify the use of Web services, much like the core Spring `JdbcTemplate` does for JDBC. The design principle common to Spring template classes is to provide helper methods to perform common operations, and for more sophisticated usage, delegate to user implemented callback interfaces. The Web service template follows the same design. The classes offer various convenience methods for the sending and receiving of XML messages, marshalling objects to XML before sending, and allows for multiple transport options.

6.2. Using the client-side API

6.2.1. `WebServiceTemplate`

The `WebServiceTemplate` is the core class for client-side Web service access in Spring-WS. It contains methods for sending `Source` objects, and receiving response messages as either `Source` or `Result`. Additionally, it can marshal objects to XML before sending them across a transport, and unmarshal any response XML into an object again.

6.2.1.1. URIs and Transports

The `WebServiceTemplate` class uses an URI as the message destination. You can either set a defaultUri property on the template itself, or supply an URI explicitly when calling a method on the template. The URI will be resolved into a `WebServiceMessageSender`, which is responsible for sending the XML message across a transport layer. You can set one or more message senders using the messageSender or messageSenders properties of the `WebServiceTemplate` class.

There are two implementations of the `WebServiceMessageSender` interface for sending messages via HTTP. The default implementation is the `HttpUrlConnectionMessageSender`, which uses the facilities provided by Java itself. The alternative is the `CommonsHttpMessageSender`, which uses the Jakarta Commons HttpClient. Use the latter if you need more advanced and easy-to-use functionality (such as authentication, HTTP connection pooling, and so forth).

6.2.1.2. Message factories

In addition to a message sender, the `WebServiceTemplate` requires a Web service message factory. There are two message factories for SOAP: `SaajSoapMessageFactory` and `AxiomSoapMessageFactory`. If no message factory is specified (via the '`messageFactory`' property), Spring-WS will use the `SaajSoapMessageFactory` by default.

6.2.2. Sending and receiving a `WebServiceMessage`

The `WebServiceTemplate` contains many convenience methods to send and receive web service messages. There are methods that accept and return a `Source` and those that return a `Result`. Additionally, there are methods which marshal and unmarshal objects to XML. Here is an example that sends a simple XML message to a Web service.

```
import java.io.StringReader;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import org.springframework.ws.WebServiceMessageFactory;
import org.springframework.ws.client.core.WebServiceTemplate;
import org.springframework.ws.transport.WebServiceMessageSender;

public class WebServiceClient {

    private static final String MESSAGE = "<message xmlns=\"http://tempuri.org\">Hello Web Service World</messag

    private final WebServiceTemplate webServiceTemplate = new WebServiceTemplate();

    public void setDefaultUri(String defaultUri) {
        webServiceTemplate.setDefaultUri(defaultUri);
    }

    // send to the configured default URI
    public void simpleSendAndReceive() {
        StreamSource source = new StreamSource(new StringReader(MESSAGE));
        StreamResult result = new StreamResult(System.out);
        webServiceTemplate.sendSourceAndReceiveToResult(source, result);
    }

    // send to an explicit URI
    public void customSendAndReceive() {
        StreamSource source = new StreamSource(new StringReader(MESSAGE));
        StreamResult result = new StreamResult(System.out);
        webServiceTemplate.sendSourceAndReceiveToResult("http://localhost:8080/AnotherWebService", source, resul
    }

}
```

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <bean id="webServiceClient" class="WebServiceClient">
        <property name="defaultUri" value="http://localhost:8080/WebService"/>
    </bean>

</beans>
```

The above example uses the `WebServiceTemplate` to send a hello world message to the web service located at `http://localhost:8080/WebService` (in the case of the `simpleSendAndReceive()` method), and writes the result to the console. The `WebServiceTemplate` is injected with the default URI, which is used because no URI was supplied explicitly in the Java code.

Please note that the `WebServiceTemplate` class is threadsafe once configured (assuming that all of it's dependencies are threadsafe too, which is the case for all of the dependencies that ship with Spring-WS), and so multiple objects can use the same shared `WebServiceTemplate` instance if so desired. The `WebServiceTemplate` exposes a zero argument constructor and messageFactory/messageSender bean properties which can be used for constructing the instance (using a Spring container or plain Java code). Alternatively, consider deriving from Spring-WS's `WebServiceGatewaySupport` convenience base class, which exposes convenient bean properties to enable easy configuration. (You do *not* have to extend this base class... it is provided as a convenience class only.)

6.2.3. Sending and receiving POJOs - marshalling and unmarshalling

In order to facilitate the sending of plain Java objects, the `WebServiceTemplate` has a number of `send(..)` methods that take an `Object` as an argument for a message's data content. The method `marshalSendAndReceive(..)` in the `WebServiceTemplate` class delegates the conversion of the request object to XML to a `Marshaller`, and the conversion of the response XML to an object to an `Unmarshaller`. (For more information about marshalling and unmarshaller, refer to Chapter 8, *Marshalling XML using O/X Mappers*.) By using the marshallers, your application code can focus on the business object that is being sent or received and not be concerned with the details of how it is represented as XML. In order to use the marshalling functionality, you have to set a marshaller and unmarshaller with the marshaller/unmarshaller properties of the `WebServiceTemplate` class.

6.2.4. `WebServiceMessageCallback`

To accommodate the setting of SOAP headers and other settings on the message, the `WebServiceMessageCallback` interface gives you access to the message *after* it has been created, but *before* it is sent. The example below demonstrates how to set the SOAP Action header on a message that is created by marshalling an object.

```
public void marshalWithSoapActionHeader(MyObject o) {

    webServiceTemplate.marshalSendAndReceive(o, new WebServiceMessageCallback() {

        public void doInMessage(WebServiceMessage message) {
            ((SoapMessage)message).setSoapAction("http://tempuri.org/Action");
        }
    });
}
```

6.2.5. `WebServiceMessageExtractor`

The `WebServiceMessageExtractor` interface is a low-level callback interface that allows you to have full control over the process to extract an `Object` from a received `WebServiceMessage`. The `WebServiceTemplate` will invoke the `extractData(..)` method on a supplied `WebServiceMessageExtractor` *while the underlying connection to the serving resource is still open*. The following example illustrates the `WebServiceMessageExtractor` in action:

```
public void marshalWithSoapActionHeader(final Source s) {
    final Transformer transformer = transformerFactory.newTransformer();
    webServiceTemplate.sendAndReceive(new WebServiceMessageCallback() {

        public void doInMessage(WebServiceMessage message) {
            transformer.transform(s, message.getPayloadResult());
        }, new WebServiceMessageExtractor() {

        public Object extractData(WebServiceMessage message) throws IOException
            // do your own transforms with message.getPayloadResult()
            //     or message.getPayloadSource()
        }
    });
}
```

Chapter 7. Securing your Web services with Spring-WS

7.1. Introduction

This chapter explains how to add WS-Security aspects to your Web services. We will focus on the three different areas of WS-Security, namely:

Authentication. This is the process of determining whether a *principal* is who they claim to be. In this context, a "principal" generally means a user, device or some other system which can perform an action in your application.

Digital signatures. The digital signature of a message is a piece of information based on both the document and the signer's private key. It is created through the use of a hash function and a private signing function (encrypting with the signer's private key).

Encryption and Decryption. *Encryption* is the process of transforming data into a form that is impossible to read without the appropriate key. It is mainly used to keep information hidden from anyone for whom it is not intended. *Decryption* is the reverse of encryption; it is the process of transforming of encrypted data back into an readable form.

All of these three areas are implemented using the `XwsSecurityInterceptor`, which we will describe in Section 7.2, “`XwsSecurityInterceptor`”.

> **Note**
>
> Note that WS-Security (especially encryption and signing) requires substantial amounts of memory, and will also decrease performance. If performance is important to you, you might want to consider not using WS-Security.

7.2. `XwsSecurityInterceptor`

The `XwsSecurityInterceptor` is an `EndpointInterceptor` (see Section 5.4.4, “Intercepting requests - the `EndpointInterceptor` interface”) that is based on SUN's XML and Web Services Security package (XWSS). This WS-Security implementation is part of the Java Web Services Developer Pack (*Java WSDP*).

Like any other endpoint interceptor, it is defined in the endpoint mapping (see Section 5.4, “Endpoint mappings”). This means that you can be selective about adding WS-Security support: some endpoint mappings require it, while others do not.

The `XwsSecurityInterceptor` requires a *security policy file* to operate. This XML file tells the interceptor what security aspects to require from incoming SOAP messages, and what aspects to add to outgoing messages. The basic format of the policy file will be explained in the following sections, but you can find a more in-depth tutorial *here*. You can set the policy with the `policyConfiguration` property, which requires a Spring resource. The policy file can contain multiple elements, e.g. require a username token on incoming messages, and sign all outgoing messages. It contains a `SecurityConfiguration` element as root (not a `JAXRPCSecurity` element).

Additionally, the security interceptor requires one or more `CallbackHandlers` to operate. These handlers are used to retrieve certificates, private keys, validate user credentials, etc. Spring-WS offers handlers for most

common security concerns, e.g. authenticating against a Acegi authentication manager, signing outgoing messages based on a X509 certificate. The following sections will indicate what callback handler to use for which security concern. You can set the callback handlers using the `callbackHandler` or `callbackHandlers` property.

Here is an example that shows how to wire the `XwsSecurityInterceptor` up:

```
<beans>
    <bean id="wsSecurityInterceptor"
        class="org.springframework.ws.soap.security.xwss.XwsSecurityInterceptor">
        <property name="policyConfiguration" value="classpath:securityPolicy.xml"/>
        <property name="callbackHandlers">
            <list>
                <ref bean="certificateHandler"/>
                <ref bean="authenticationHandler"/>
            </list>
        </property>
    </bean>
    ...
</beans>
```

This interceptor is configured using the `securityPolicy.xml` file on the classpath. It uses two callback handlers which are defined further on in the file.

7.3. Keystores

For most cryptographic operations, you will use standard `java.security.KeyStore` objects. This includes certificate verification, message signing, signature verification, and encryption, but excludes username and time-stamp verification. This section aims to give you some background knowledge on keystores, and the Java tools that you can use to store keys and certificates in a keystore file. This information is mostly not related to Spring-WS, but to the general cryptographic features of Java.

The `java.security.KeyStore` class represents a storage facility for cryptographic keys and certificates. It can contain three different sort of elements:

Private Keys. These keys are used for self-authentication. The private key is accompanied by certificate chain for the corresponding public key. Within the field of WS-Security, this accounts to message signing and message decryption.

Symmetric Keys. Symmetric (or secret) keys are used for message encryption and decryption as well. The difference being that both sides (sender and recipient) share the same, secret key.

Trusted certificates. These X509 certificates are called a *trusted certificate* because the keystore owner trusts that the public key in the certificates indeed belong to the owner of the certificate. Within WS-Security, these certificates are used for certificate validation, signature verification, and encryption.

7.3.1. KeyTool

Supplied with your Java Virtual Machine is the **keytool** program, a key and certificate management utility. You can use this tool to create new keystores, add new private keys and certificates to them, etc. It is beyond the scope of this document to provide a full reference of the **keytool** command, but you can find a reference *here*, or by giving the command `keytool -help` on the command line.

7.3.2. KeyStoreFactoryBean

To easily load a keystore using Spring configuration, you can use the `KeyStoreFactoryBean`. It has a resource location property, which you can set to point to the path of the keystore to load. A password may be given to check the integrity of the keystore data. If a password is not given, integrity checking is not performed.

```
<bean id="keyStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
    <property name="password" value="password"/>
    <property name="location" value="classpath:org/springframework/ws/soap/security/xwss/test-keystore.jks"/>
</bean>
```

Caution

If you don't specify the location property, a new, empty keystore will be created, which is most likely not what you want.

7.3.3. KeyStoreCallbackHandler

To use the keystores within a `XwsSecurityInterceptor`, you will need to define a `KeyStoreCallbackHandler`. This callback has three properties with type keystore: (`keyStore`, `trustStore`, and `symmetricStore`). The exact stores used by the handler depend on the cryptographic operations that are to be performed by this handler. For private key operation, the `keyStore` is used, for symmetric key operations the `symmetricStore`, and for determining trust relationships, the `trustStore`. The following table indicates this:

Cryptographic operation	**Keystore used**
Certificate validation	first the `keyStore`, then the `trustStore`
Decryption based on private key	`keyStore`
Decryption based on symmetric key	`symmetricStore`
Encryption based on public key certificate	`trustStore`
Encryption based on symmetric key	`symmetricStore`
Signing	`keyStore`
Signature verification	`trustStore`

Additionally, the `KeyStoreCallbackHandler` has a `privateKeyPassword` property, which should be set to unlock the private key(s) contained in the `keyStore`.

If the `symmetricStore` is not set, it will default to the `keyStore`. If the key or trust store is not set, the callback handler will use the standard Java mechanism to load or create it. Refer to the JavaDoc of the `KeyStoreCallbackHandler` to know how this mechanism works.

For instance, if you want to use the `KeyStoreCallbackHandler` to validate incoming certificates or signatures, you would use a trust store, like so:

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="trustStore" ref="trustStore"/>
    </bean>

    <bean id="trustStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:truststore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
```

```
</beans>
```

If you want to use it to decrypt incoming certificates or sign outgoing messages, you would use a key store, like so:

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="keyStore" ref="keyStore"/>
        <property name="privateKeyPassword" value="changeit"/>
    </bean>

    <bean id="keyStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:keystore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

The following sections will indicate where the `KeyStoreCallbackHandler` can be used, and which properties to set for particular cryptographic operations.

7.4. Authentication

As stated in the introduction, *authentication* is the task of determining whether a principal is who they claim to be. Within WS-Security, authentication can take two forms: using a username and password token (using either a plain text password or a password digest), or using a X509 certificate.

7.4.1. Plain Text Username Authentication

The simplest form of username authentication uses *plain text passwords*. In this scenario, the SOAP message will contain a `UsernameToken` element, which itself contains a `Username` element and a `Password` element which contains the plain text password. Plain text authentication can be compared to the Basic Authentication provided by HTTP servers.

> **Warning**
>
> Note that plain text passwords are not very secure. Therefore, you should always add additional security measures to your transport layer if you are using them (using HTTPS instead of plain HTTP, for instance).

To require that every incoming message contains a `UsernameToken` with a plain text password, the security policy file should contain a `RequireUsernameToken` element, with the `passwordDigestRequired` attribute set to `false`. You can find a reference of possible child elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    ...
    <xwss:RequireUsernameToken passwordDigestRequired="false" nonceRequired="false"/>
    ...
</xwss:SecurityConfiguration>
```

If the username token is not present, the `XwsSecurityInterceptor` will return a SOAP Fault to the sender. If it is present, it will fire a `PasswordValidationCallback` with a `PlainTextPasswordRequest` to the registered handlers. Within Spring-WS, there are three classes which handle this particular callback.

7.4.1.1. SimplePasswordValidationCallbackHandler

The simplest password validation handler is the `SimplePasswordValidationCallbackHandler`. This handler validates passwords against an in-memory `Properties` object, which you can specify using the `users` property, like so:

```
<bean id="passwordValidationHandler"
    class="org.springframework.ws.soap.security.xwss.callback.SimplePasswordValidationCallbackHandler">
    <property name="users">
        <props>
            <prop key="Bert">Ernie</prop>
        </props>
    </property>
</bean>
```

In this case, we are only allowing the user "Bert" to log in using the password "Ernie".

7.4.1.2. AcegiPlainTextPasswordValidationCallbackHandler

The `AcegiPlainTextPasswordValidationCallbackHandler` uses the excellent *Acegi Security Framework* to authenticate users. It is beyond the scope of this document to describe Acegi, but suffice it to say that Acegi is a full-fledged security framework. You can read more about Acegi in the *Acegi reference documentation*.

The `AcegiPlainTextPasswordValidationCallbackHandler` requires an Acegi `AuthenticationManager` to operate. It uses this manager to authenticate against a `UsernamePasswordAuthenticationToken` that it creates. If authentication is successful, the token is stored in the `SecurityContextHolder`. You can set the authentication manager using the `authenticationManager` property:

```
<beans>
  <bean id="acegiHandler"
      class="org.springframework.ws.soap.security.xwss.callback.acegi.AcegiPlainTextPasswordValidationCallbackHa
    <property name="authenticationManager" ref="authenticationManager"/>
  </bean>

  <bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">
      <property name="providers">
          <bean class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
              <property name="userDetailsService" ref="userDetailsService"/>
          </bean>
      </property>
  </bean>

  <bean id="userDetailsService" class="com.mycompany.app.dao.UserDetailService" />
  ...
</beans>
```

7.4.1.3. JaasPlainTextPasswordValidationCallbackHandler

The `JaasPlainTextPasswordValidationCallbackHandler` is based on the standard *Java Authentication and Authorization Service*. It is beyond the scope of this document to provide a full introduction into JAAS, but there is a *good tutorial* available.

The `JaasPlainTextPasswordValidationCallbackHandler` requires only a `loginContextName` to operate. It creates a new JAAS `LoginContext` using this name, and handles the standard JAAS `NameCallback` and `PasswordCallback` using the username and password provided in the SOAP message. This means that this callback handler integrates with any JAAS `LoginModule` that fires these callbacks during the `login()` phase, which is standard behavior.

You can wire up a `JaasPlainTextPasswordValidationCallbackHandler` as follows:

```
<bean id="jaasValidationHandler"
    class="org.springframework.ws.soap.security.xwss.callback.jaas.JaasPlainTextPasswordValidationCallbackHandle
    <property name="loginContextName" value="MyLoginModule" />
</bean>
```

In this case, the callback handler uses the `LoginContext` named "MyLoginModule". This module should be defined in your `jaas.config` file, as explained in the abovementioned tutorial.

7.4.2. Digest Username Authentication

When using password digests, the SOAP message also contains a `UsernameToken` element, which itself contains a `Username` element and a `Password` element. The difference is that the password is not sent as plain text, but as a *digest*. The recipient compares this digest to the digest he calculated from the known password of the user, and if they are the same, the user is authenticated. It can be compared to the Digest Authentication provided by HTTP servers.

To require that every incoming message contains a `UsernameToken` element with a password digest, the security policy file should contain a `RequireUsernameToken` element, with the `passwordDigestRequired` attribute set to `true`. Additionally, the `nonceRequired` should be set to `true`: You can find a reference of possible child elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    ...
    <xwss:RequireUsernameToken passwordDigestRequired="true" nonceRequired="true"/>
    ...
</xwss:SecurityConfiguration>
```

If the username token is not present, the `XwsSecurityInterceptor` will return a SOAP Fault to the sender. If it is present, it will fire a `PasswordValidationCallback` with a `DigestPasswordRequest` to the registered handlers. Within Spring-WS, there are two classes which handle this particular callback.

7.4.2.1. SimplePasswordValidationCallbackHandler

The `SimplePasswordValidationCallbackHandler` can handle both plain text passwords as well as password digests. It is described in Section 7.4.1.1, “SimplePasswordValidationCallbackHandler”.

7.4.2.2. AcegiDigestPasswordValidationCallbackHandler

The `AcegiPlainTextPasswordValidationCallbackHandler` requires an Acegi `UserDetailService` to operate. It uses this service to retrieve the password of the user specified in the token. The digest of the password contained in this details object is then compared with the digest in the message. If they are equal, the user has successfully authenticated, and a `UsernamePasswordAuthenticationToken` is stored in the `SecurityContextHolder`. You can set the service using the `userDetailsService`. Additionally, you can set a `userCache` property, to cache loaded user details.

```
<beans>
    <bean class="org.springframework.ws.soap.security.xwss.callback.acegi.AcegiDigestPasswordValidationCallbackH
        <property name="userDetailsService" ref="userDetailsService"/>
    </bean>

    <bean id="userDetailsService" class="com.mycompany.app.dao.UserDetailService" />
    ...
</beans>
```

7.4.3. Certificate Authentication

A more secure way of authentication uses X509 certificates. In this scenerario, the SOAP message contains a `BinarySecurityToken`, which contains a Base 64-encoded version of a X509 certificate. The recipient is used by the recipient to authenticate. The certificate stored in the message is also used to sign the message (see Section 7.5.1, “Verifying Signatures”).

To make sure that all incoming SOAP messages carry a `BinarySecurityToken`, the security policy file should contain a `RequireSignature` element. This element can further carry other elements, which will be covered in Section 7.5.1, “Verifying Signatures”. You can find a reference of possible child elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    ...
    <xwss:RequireSignature requireTimestamp="false">
    ...
</xwss:SecurityConfiguration>
```

When a message arrives that carries no certificate, the `XwsSecurityInterceptor` will return a SOAP Fault to the sender. If it is present, it will fire a `CertificateValidationCallback`. There are three handlers within Spring-WS which handle this callback for authentication purposes.

> **Note**
>
> In most cases, certificate *authentication* should be preceded by certificate *validation*, since you only want to authenticate against valid certificates. Invalid certificates such as certificates for which the expiration date has passed, or which are not in your store of trusted certificates, should be ignored.
>
> In Spring-WS terms, this means that the `AcegiCertificateValidationCallbackHandler` or `JaasCertificateValidationCallbackHandler` should be preceded by `KeyStoreCallbackHandler`. This can be accomplished by setting the order of the `callbackHandlers` property in the configuration of the `XwsSecurityInterceptor`:

```
<bean id="wsSecurityInterceptor"
    class="org.springframework.ws.soap.security.xwss.XwsSecurityInterceptor">
    <property name="policyConfiguration" value="classpath:securityPolicy.xml"/>
    <property name="callbackHandlers">
        <list>
            <ref bean="keyStoreHandler"/>
            <ref bean="acegiHandler"/>
        </list>
    </property>
</bean>
```

> Using this setup, the interceptor will first determine if the certificate in the message is valid using the keystore, and then authenticate against it.

7.4.3.1. KeyStoreCallbackHandler

The `KeyStoreCallbackHandler` uses a standard Java keystore to validate certificates. This certificate validation process consists of the following steps:

1. First, the handler will check whether the certificate is in the private `keyStore`. If it is, it is valid.
2. If the certificate is not in the private keystore, the handler will check whether the the current date and time

are within the validity period given in the certificate. If they are not, the certificate is invalid; if it is, it will continue with the final step.

3. Finally, a *certification path* for the certificate is created. This basically means that the handler will determine whether the certificate has been issued by any of the certificate authorities in the `trustStore`. If a certification path can be built successfully, the certificate is valid. Otherwise, the certificate is not.

To use the `KeyStoreCallbackHandler` for certificate validation purposes, you will most likely set only the `trustStore` property:

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="trustStore" ref="trustStore"/>
    </bean>

    <bean id="trustStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:truststore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

Using this setup, the certificate that is to be validated must either be in the trust store itself, or the trust store must contain a certificate authority that issued the certificate.

7.4.3.2. AcegiCertificateValidationCallbackHandler

The `AcegiCertificateValidationCallbackHandler` requires an Acegi `AuthenticationManager` to operate. It uses this manager to authenticate against a `X509AuthenticationToken` that it creates. The configured authentication manager is expected to supply a provider which can handle this token (usually an instance of `X509AuthenticationProvider`). If authentication is succesful, the token is stored in the `SecurityContextHolder`. You can set the authentication manager using the `authenticationManager` property:

```
<beans>
    <bean id="acegiCertificateHandler"
        class="org.springframework.ws.soap.security.xwss.callback.acegi.AcegiCertificateValidationCallbackHandle
        <property name="authenticationManager" ref="authenticationManager"/>
    </bean>

    <bean id="authenticationManager"
        class="org.acegisecurity.providers.ProviderManager">
        <property name="providers">
            <bean class="org.acegisecurity.providers.x509.X509AuthenticationProvider">
                <property name="x509AuthoritiesPopulator">
                    <bean class="org.acegisecurity.providers.x509.populator.DaoX509AuthoritiesPopulator">
                        <property name="userDetailsService" ref="userDetailsService"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

  <bean id="userDetailsService" class="com.mycompany.app.dao.UserDetailService" />
  ...
</beans>
```

In this case, we are using a custom user details service to obtain authentication details based on the certificate. Refer to the *Acegi reference documentation* for more information about authentication against X509 certificates.

7.4.3.3. JaasCertificateValidationCallbackHandler

The `JaasCertificateValidationCallbackHandler` requires a `loginContextName` to operate. It creates a new JAAS `LoginContext` using this name and with the `X500Principal` of the certificate. This means that this callback handler integrates with any JAAS `LoginModule` that handles X500 principals.

You can wire up a `JaasCertificateValidationCallbackHandler` as follows:

```
<bean id="jaasValidationHandler"
    class="org.springframework.ws.soap.security.xwss.callback.jaas.JaasCertificateValidationCallbackHandler">
    <property name="loginContextName">MyLoginModule</property>
</bean>
```

In this case, the callback handler uses the `LoginContext` named "MyLoginModule". This module should be defined in your `jaas.config` file, and should be able to authenticate against X500 principals.

7.5. Digital Signatures

The *digital signature* of a message is a piece of information based on both the document and the signer's private key. There are two main tasks related to signatures in WS-Security: verifying signatures and signing messages.

7.5.1. Verifying Signatures

Just like certificate-based authentication, a signed message contains a `BinarySecurityToken`, which contains the certificate used to sign the message. Additionally, it contains a `SignedInfo` block, which indicates what part of the message was signed.

To make sure that all incoming SOAP messages carry a `BinarySecurityToken`, the security policy file should contain a `RequireSignature` element. It can also contain a `SignatureTarget` element, which specifies the target message part which was expected to be signed, and various other subelements. You can also define the private key alias to use, whether to use a symmetric instead of a private key, and many other properties. You can find a reference of possible child elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:RequireSignature requireTimestamp="false"/>
</xwss:SecurityConfiguration>
```

If the signature is not present, the `XwsSecurityInterceptor` will return a SOAP Fault to the sender. If it is present, it will fire a `SignatureVerificationKeyCallback` to the registered handlers. Within Spring-WS, there are is one class which handles this particular callback: the `KeyStoreCallbackHandler`.

7.5.1.1. KeyStoreCallbackHandler

As described in Section 7.3.3, “KeyStoreCallbackHandler”, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` for handling various cryptographic callbacks, including signature verification. For signature verification, the handler uses the `trustStore` property:

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="trustStore" ref="trustStore"/>
    </bean>

    <bean id="trustStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:org/springframework/ws/soap/security/xwss/test-truststore.jks
        <property name="password" value="changeit"/>
```

```
    </bean>
</beans>
```

7.5.2. Signing Messages

When signing a message, the `XwsSecurityInterceptor` adds the `BinarySecurityToken` to the message, and a `SignedInfo` block, which indicates what part of the message was signed.

To sign all outgoing SOAP messages, the security policy file should contain a `Sign` element. It can also contain a `SignatureTarget` element, which specifies the target message part which was expected to be signed, and various other subelements. You can also define the private key alias to use, whether to use a symmetric instead of a private key, and many other properties. You can find a reference of possible child elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
        <xwss:Sign includeTimestamp="false" />
</xwss:SecurityConfiguration>
```

The `XwsSecurityInterceptor` will fire a `SignatureKeyCallback` to the registered handlers. Within Spring-WS, there are is one class which handles this particular callback: the `KeyStoreCallbackHandler`.

7.5.2.1. KeyStoreCallbackHandler

As described in Section 7.3.3, “KeyStoreCallbackHandler”, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` for handling various cryptographic callbacks, including signing messages. For adding signatures, the handler uses the `keyStore` property. Additionally, you must set the `privateKeyPassword` property to unlock the private key used for signing.

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="keyStore" ref="keyStore"/>
        <property name="privateKeyPassword" value="changeit"/>
    </bean>

    <bean id="keyStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:keystore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

7.6. Encryption and Decryption

When *encrypting*, the message is transformed into a form that can only be read with the appropriate key. The message can be *decrypted* to reveal the original, readable message.

7.6.1. Decryption

To decrypt incoming SOAP messages, the security policy file should contain a `RequireEncryption` element. This element can further carry a `EncryptionTarget` element which indicates which part of the message should be encrypted, and a `SymmetricKey` to indicate that a shared secret instead of the regular private key should be used to decrypt the message. You can read a description of the other elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
```

```
    <xwss:RequireEncryption />
</xwss:SecurityConfiguration>
```

If an incoming message is not encrypted, the `XwsSecurityInterceptor` will return a SOAP Fault to the sender. If it is present, it will fire a `DecryptionKeyCallback` to the registered handlers. Within Spring-WS, there is one class which handled this particular callback: the `KeyStoreCallbackHandler`.

7.6.1.1. KeyStoreCallbackHandler

As described in Section 7.3.3, “KeyStoreCallbackHandler”, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` for handling various cryptographic callbacks, including decryption. For decryption, the handler uses the `keyStore` property. Additionally, you must set the `privateKeyPassword` property to unlock the private key used for decryption. For decryption based on symmetric keys, it will use the `symmetricStore`.

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="keyStore" ref="keyStore"/>
        <property name="privateKeyPassword" value="changeit"/>
    </bean>

    <bean id="keyStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:keystore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

7.6.2. Encryption

To encrypt outgoing SOAP messages, the security policy file should contain a `Encrypt` element. This element can further carry a `EncryptionTarget` element which indicates which part of the message should be encrypted, and a `SymmetricKey` to indicate that a shared secret instead of the regular private key should be used to decrypt the message. You can read a description of the other elements *here*.

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:Encrypt />
</xwss:SecurityConfiguration>
```

The `XwsSecurityInterceptor` will fire a `EncryptionKeyCallback` to the registered handlers in order to retrieve the encryption information. Within Spring-WS, there is one class which handled this particular callback: the `KeyStoreCallbackHandler`.

7.6.2.1. KeyStoreCallbackHandler

As described in Section 7.3.3, “KeyStoreCallbackHandler”, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` for handling various cryptographic callbacks, including encryption. For encryption based on public keys, the handler uses the `trustStore` property. For encryption based on symmetric keys, it will use the `symmetricStore`.

```
<beans>
    <bean id="keyStoreHandler" class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbackHandler
        <property name="trustStore" ref="trustStore"/>
    </bean>

    <bean id="trustStore" class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:truststore.jks"/>
        <property name="password" value="changeit"/>
```

```
    </bean>
</beans>
```

Chapter 8. Marshalling XML using O/X Mappers

8.1. Introduction

In this chapter, we will describe Spring's Object/XML Mapping support. Object/XML Mapping, or O/X mapping for short, is the act of converting an XML document to and from an object. This conversion process is also known as XML Marshalling, or XML Serialization. This chapter uses these terms interchangeably.

Within the field of O/X mapping, a *marshaller* is responsible for serializing an object (graph) to XML. In similar fashion, an *unmarshaller* deserializes the XML to an object graph. This XML can take the form of a DOM document, an input or output stream, or a SAX handler.

Some of the benefits of using Spring for your O/X mapping needs are:

Ease of configuration. Spring's bean factory makes it easy to configure marshallers, without needing to construct JAXB context, JiBX binding factories, etc. The marshallers can be configured as any other bean in your application context.

Consistent Interfaces. Spring's O/X mapping operates through two global interfaces: the `Marshaller` and `Unmarshaller` interface. These abstractions allow you to switch O/X mapping frameworks with relative ease, with little or no changes required on the classes that do the marshalling. This approach has the additional benefit of making it possible to do XML marshalling with a mix-and-match approach (e.g. some marshalling performed using JAXB, other using XMLBeans) in a non-intrusive fashion, leveraging the strength of each technology.

Consistent Exception Hierarchy. Spring provides a conversion from exceptions from the underlying O/X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. As can be expected, these runtime exceptions wrap the original exception so no information is lost.

8.2. Marshaller and Unmarshaller

As stated in the introduction, a *marshaller* serializes an object to XML, and an *unmarshaller* deserializes XML stream to an object. In this section, we will describe the two Spring interfaces used for this purpose.

8.2.1. Marshaller

Spring abstracts all marshalling operations behind the `org.springframework.oxm.Marshaller` interface, the main methods of which is listed below.

```
public interface Marshaller {

    /**
     * Marshals the object graph with the given root into the provided Result.
     */
    void marshal(Object graph, Result result)
        throws XmlMappingException, IOException;
}
```

The `Marshaller` interface has one main method, which marshals the given object to a given `javax.xml.transform.Result`. Result is a tagging interface that basically represents an XML output abstraction: concrete implementations wrap various XML representations, as indicated in the table below.

`javax.xml.transform.Result` implementation	Wraps XML representation
`javax.xml.transform.dom.DOMResult`	`org.w3c.dom.Node`
`javax.xml.transform.sax.SAXResult`	`org.xml.sax.ContentHandler`
`javax.xml.transform.stream.StreamResult`	`java.io.File`, `java.io.OutputStream`, or `java.io.Writer`

Note

Although the `marshal` method accepts a plain object as its first parameter, most `Marshaller` implementations cannot handle arbitrary objects. Instead, an object class must be mapped in a mapping file, registered with the marshaller, or have a common base class. Refer to the further sections in this chapter to determine how your O/X technology of choice manages this.

8.2.2. Unmarshaller

Similar to the `Marshaller`, there is the `org.springframework.oxm.Unmarshaller` interface.

```
public interface Unmarshaller {

    /**
     * Unmarshals the given provided Source into an object graph.
     */
    Object unmarshal(Source source)
        throws XmlMappingException, IOException;
}
```

This interface also has one method, which reads from the given `javax.xml.transform.Source` (an XML input abstraction), and returns the object read. As with Result, Source is a tagging interface that has three concrete implementations. Each wraps a different XML representation, as indicated in the table below.

`javax.xml.transform.Source` implementation	Wraps XML representation
`javax.xml.transform.dom.DOMSource`	`org.w3c.dom.Node`
`javax.xml.transform.sax.SAXSource`	`org.xml.sax.InputSource`, and `org.xml.sax.XMLReader`
`javax.xml.transform.stream.StreamSource`	`java.io.File`, `java.io.InputStream`, or `java.io.Reader`

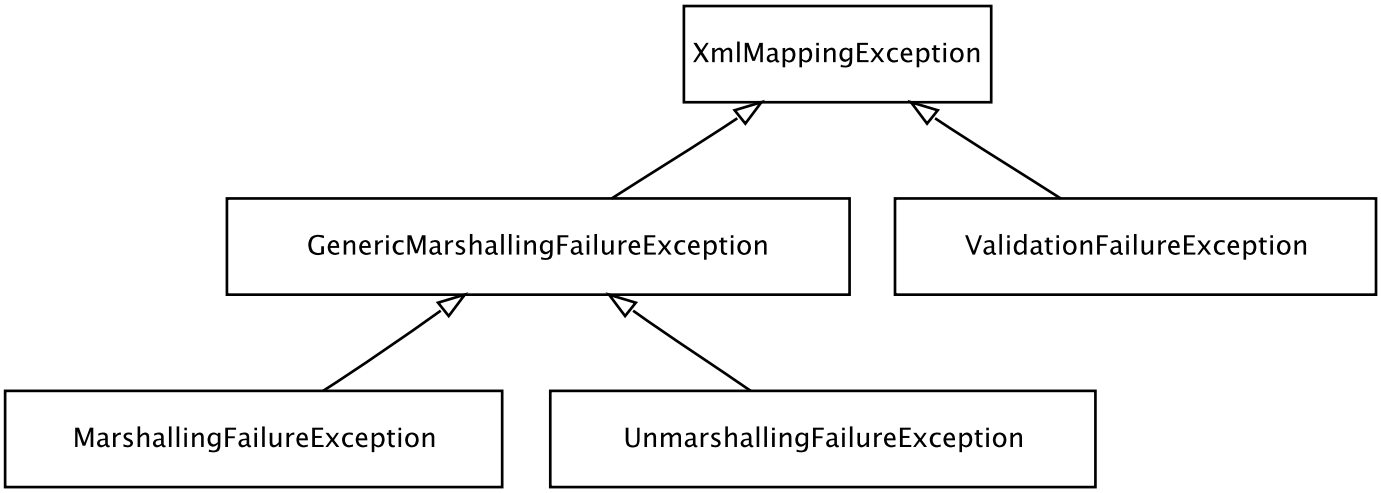
Even though there are two separate marshalling interfaces (`Marshaller` and `Unmarshaller`), all implementations found in Spring-WS implement both in one class. This means that you can wire up one marshaller class and refer to it both as a marshaller and an unmarshaller in your `applicationContext.xml`.

8.2.3. XmlMappingException

Spring converts exceptions from the underlying O/X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. As can be expected, these runtime exceptions wrap the original exception so no information will be lost.

Additionally, the `MarshallingFailureException` and `UnmarshallingFailureException` provide a distinction between marshalling and unmarshalling operations, even though the underlying O/X mapping tool does not do so.

The O/X Mapping exception hierarchy is shown in the following figure:



O/X Mapping exception hierarchy

8.3. Using Marshaller and Unmarshaller

Spring's OXM can be used for a wide variety of situations. In the following example, we will use it to marshal the settings of a Spring-managed application as an XML file. We will use a simple JavaBean to represent the settings:

```
public class Settings {
    private boolean fooEnabled;

    public boolean isFooEnabled() {
        return fooEnabled;
    }

    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

The application class uses this bean to store its settings. Besides a main method, the class has two methods: `saveSettings` saves the settings bean to a file named `settings.xml`, and `loadSettings` loads these settings again. A `main` method constructs a Spring application context, and calls these two methods.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;

public class Application {
    private static final String FILE_NAME = "settings.xml";
    private Settings settings = new Settings();
    private Marshaller marshaller;
    private Unmarshaller unmarshaller;
```

```
    public void setMarshaller(Marshaller marshaller) {
        this.marshaller = marshaller;
    }

    public void setUnmarshaller(Unmarshaller unmarshaller) {
        this.unmarshaller = unmarshaller;
    }

    public void saveSettings() throws IOException {
        FileOutputStream os = null;
        try {
            os = new FileOutputStream(FILE_NAME);
            this.marshaller.marshal(settings, new StreamResult(os));
        } finally {
            if (os != null) {
                os.close();
            }
        }
    }

    public void loadSettings() throws IOException {
        FileInputStream is = null;
        try {
            is = new FileInputStream(FILE_NAME);
            this.settings = (Settings) this.unmarshaller.unmarshal(new StreamSource(is));
        } finally {
            if (is != null) {
                is.close();
            }
        }
    }

    public static void main(String[] args) throws IOException {
        ApplicationContext appContext = new ClassPathXmlApplicationContext("applicationContext.xml");
        Application application = (Application) appContext.getBean("application");
        application.saveSettings();
        application.loadSettings();
    }
}
```

The `Application` requires both a marshaller and unmarshaller property to be set. We can do so using the following `applicationContext.xml`:

```
<beans>
    <bean id="application" class="Application">
        <property name="marshaller" ref="castorMarshaller" />
        <property name="unmarshaller" ref="castorMarshaller" />
    </bean>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>
</beans>
```

This application context uses Castor, but we could have used any of the other marshaller instances described later in this chapter. Note that Castor does not require any further configuration by default, so the bean definition is rather simple. Also note that the `CastorMarshaller` implements both `Marshaller` and `Unmarshaller`, so we can refer to the `castorMarshaller` bean in both the marshaller and unmarshaller property of the application.

This sample application produces the following `settings.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

8.4. JAXB

The JAXB binding compiler translates a W3C XML Schema into one or more Java classes, a `jaxb.properties`

file, and possibly other files, depending on the specific implementation of JAXB. Alternatively, JAXB2 offers a way to generate a schema from annotated Java classes.

Spring supports both the JAXB 1.0 and the JAXB 2.0 API as XML marshalling strategies, following the `Marshaller` and `Unmarshaller` interfaces described in Section 8.2, “Marshaller and Unmarshaller”. The corresponding integration classes reside in the org.springframework.oxm.jaxb package.

8.4.1. Jaxb1Marshaller

The `Jaxb1Marshaller` class implements both the Spring `Marshaller` and `Unmarshaller`interface. It requires a context path to operate, which you can set using the contextPath property. The context path is a list of colon (:) separated Java package names that contain schema derived classes. The marshaller has an additional validating property which defines whether to validate incoming XML.

The next sample bean configuration shows how to configure a `JaxbMarshaller` using the classes generated to `org.springframework.ws.samples.airline.schema`.

```
<beans>

    <bean id="jaxb1Marshaller" class="org.springframework.oxm.jaxb.Jaxb1Marshaller">
        <property name="contextPath" value="org.springframework.ws.samples.airline.schema"/>
    </bean>
    ...

</beans>
```

8.4.2. Jaxb2Marshaller

The `Jaxb2Marshaller` can be configured using the same contextPath property as the `Jaxb1Marshaller`. However, it also offers a classesToBeBound property, which allows you to set an array of classes to be supported by the marshaller. Schema validation is performed by specifying one or more schema resource to the bean, like so:

```
<beans>

    <bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>org.springframework.oxm.jaxb.Flight</value>
                <value>org.springframework.oxm.jaxb.Flights</value>
            </list>
        </property>
        <property name="schema" value="classpath:org/springframework/oxm/schema.xsd"/>
    </bean>
    ...

</beans>
```

8.5. Castor

Castor XML mapping is an open source XML binding framework. It allows you to transform the data contained in a java object model into/from an XML document. By default, it does not require any further configuration, though a mapping file can be used to have more control over the behavior of Castor.

For more information on Castor, refer to the *Castor web site*. The Spring integration classes reside in the org.springframework.oxm.castor package.

8.5.1. CastorMarshaller

As with JAXB, the `CastorMarshaller` implements both the `Marshaller` and `Unmarshaller` interface. It can be wired up as follows:

```
<beans>

    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />
    ...

</beans>
```

8.5.2. Mapping

Although it is possible to rely on Castor's default marshalling behavior, it might be necessary to have more control over it. This can be accomplished using a Castor mapping file. For more information, refer to Castor XML Mapping.

The mapping can be set using the mappingLocation resource property, indicated below with a classpath resource.

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" >
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

8.6. XMLBeans

XMLBeans is an XML binding tool that has full XML Schema support, and offers full XML Infoset fidelity. It takes a different approach to that of most other O/X mapping frameworks, in that all classes that are generated from an XML Schema are all derived from `XmlObject`, and contain XML binding information in them.

For more information on XMLBeans, refer to the *XMLBeans web site* . The Spring-WS integration classes reside in the org.springframework.oxm.xmlbeans package.

8.6.1. XmlBeansMarshaller

The `XmlBeansMarshaller` implements both the `Marshaller` and `Unmarshaller` interfaces. It can be configured as follows:

```
<beans>

    <bean id="xmlBeansMarshaller" class="org.springframework.oxm.xmlbeans.XmlBeansMarshaller" />
    ...

</beans>
```

Note

Note that the `XmlBeansMarshaller` can only marshal objects of type `XmlObject`, and not every `java.lang.Object`.

8.7. JiBX

The JiBX framework offers a solution similar to that which JDO provides for ORM: a binding definition defines the rules for how your Java objects are converted to or from XML. After preparing the binding and compiling the classes, a JiBX binding compiler enhances the class files, and adds code to handle converting instances of the classes from or to XML.

For more information on JiBX, refer to the *JiBX web site*. The Spring integration classes reside in the org.springframework.oxm.jibx package.

8.7.1. JibxMarshaller

The `JibxMarshaller` class implements both the `Marshaller` and `Unmarshaller` interface. To operate, it requires the name of the class to marshall in, which you can set using the targetClass property. Optionally, you can set the binding name using the bindingName property. In the next sample, we bind the `Flights` class:

```
<beans>

    <bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
        <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
    </bean>

    ...
```

A `JibxMarshaller` is configured for a single class. If you want to marshal multiple classes, you have to configure multiple `JibxMarshallers` with different targetClass property values.

8.8. XStream

XStream is a simple library to serialize objects to XML and back again. It does not require any mapping, and generates clean XML.

For more information on XStream, refer to the *XStream web site*. The Spring integration classes reside in the org.springframework.oxm.xstream package.

8.8.1. XStreamMarshaller

The `XStreamMarshaller` does not require any configuration, and can be configured in an application context directly. To further customize the XML, you can set an *alias map*, which consists of string aliases mapped to classes:

```
<beans>

    <bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
        <property name="aliases">
            <props>
                <prop key="Flight">org.springframework.oxm.xstream.Flight</prop>
            </props>
        </property>
    </bean>
    ...

</beans>
```

Note

Note that XStream is an XML serialization library, not a data binding library. Therefore, it has limited namespace support. As such, it is rather unsuitable for usage within Web services.

Part III. Other Resources

In addition to this reference documentation, there exist a number of other resources that may help you learn how to use Spring Web Services. These additional, third-party resources are enumerated in this section.

Bibliography

[waldo-94] Jim Waldo, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*. Springer Verlag. 1994.

[alpine] Steve Loughran and Edmund Smith. *Rethinking the Java SOAP Stack*. May 17, 2005. Copyright © 2005 IEEE Telephone Laboratories, Inc..

[effective-enterprise-java] Ted Neward. Scott Meyers. *Effective Enterprise Java*. Addison-Wesley. 2004.

[effective-xml] Elliotte Rusty Harold. Scott Meyers. *Effective XML*. Addison-Wesley. 2004.