Spring Web Services Reference Documentation

Arjen Poutsma, Rick Evans, Tareq Abed Rabbo, Greg Turnquist, Jay Bryant, Corneil du Plessis, Stéphane Nicoll

Version 4.0.16, 2025-10-21

Table of Contents

Preface

In the current age of Service Oriented Architectures, more and more people use 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 that contain all the data necessary to process the request. Conceptually, XML-based web services are better compared to message queues than to remoting solutions. Overall, XML should be considered the platform-neutral representation of data, the *common language* of SOA. When developing or using web services, the focus should be on this XML and not on Java.

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

I. Introduction

This first part of the reference documentation is an overview of Spring Web Services and the underlying concepts. Then, the concepts behind contract-first web service development are explained. Finally, the third section provides a tutorial.

Chapter 1. What is Spring Web Services?

1.1. Introduction

Spring Web Services (Spring-WS) is a product of the Spring community and is 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 by 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-WS are:

- Powerful mappings.
- XML API support.
- Flexible XML Marshalling.
- Reusing Your Spring expertise.
- Support for WS-Security.
- Integration with Spring Security.
- Apache license.

1.1.1. Powerful mappings

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

1.1.2. XML API support

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

1.1.3. Flexible XML Marshalling

Spring-WS builds on the Object/XML Mapping module in the Spring Framework, which supports JAXB 1 and 2, Castor, XMLBeans, JiBX, and XStream.

1.1.4. Reusing Your Spring expertise

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

1.1.5. Support for WS-Security

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

1.1.6. Integration with Spring Security

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

1.1.7. Apache license

You can confidently use Spring-WS in your project.

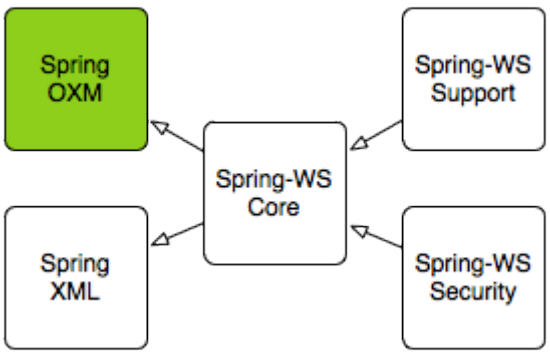
1.2. Runtime environment

Spring-WS requires a standard Java 17 Runtime Environment. Spring-WS is built on Spring Framework 6.x.

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

- The XML module (`spring-xml`) contains various XML support classes for Spring-WS. This module is mainly intended for the Spring-WS framework itself and not web service developers.
- The Core module (`spring-ws-core`) 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), the various support classes for implementing web service endpoints, and the client-side `WebServiceTemplate`.
- The Support module (`spring-ws-support`) contains additional transports (JMS, Email, and others).
- The Security module (`spring-ws-security`) provides a WS-Security implementation that integrates with the core web service package. It lets you sign, decrypt and encrypt, and add principal tokens to SOAP messages. Additionally, it lets you use your existing Spring Security implementation for authentication and authorization.

The following figure shows and the dependencies between the Spring-WS modules. Arrows indicate dependencies (that is, Spring-WS Core depends on Spring-XML and the Spring OXM).



1.3. Supported standards

Spring-WS supports the following standards:

- SOAP 1.1 and 1.2.
- WSDL 1.1 and 2.0 (XSD-based generation is supported only for WSDL 1.1).
- WS-I Basic Profile 1.0, 1.1, 1.2, and 2.0.
- WS-Addressing 1.0 and the August 2004 draft.
- SOAP Message Security 1.1, Username Token Profile 1.1, X.509 Certificate Token Profile 1.1, SAML Token Profile 1.1, Kerberos Token Profile 1.1, Basic Security Profile 1.1.

Chapter 2. Why Contract First?

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

What is WSDL?

WSDL stands for Web Service 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, see the WSDL specification.

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

2.1. Object/XML Impedance Mismatch

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

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

2.1.1. XSD Extensions

In Java, the only way to change the behavior of a class is to subclass it to add 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 way of a regular expression, allowing only three upper case letters. If this type is converted to Java, we 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.

2.1.2. 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, and others. Because all of these languages have different class libraries, you must use some common, cross-language format to communicate between them. That format is XML, which is supported by all of 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`:

```
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 probably ends 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 probably use either a `java.time.LocalDateTime` or `java.time.Instant`. However, both of these classes actually describe times, rather than dates. So, we 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`.

2.1.3. Cyclic Graphs

Imagine we have the following 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 take a naive approach to converting this to XML, we 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>
                   ...
```

Processing such a structure is likely to take a 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:

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

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

This solves the recursion 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 the 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.

2.2. 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.

- Fragility.
- Performance.
- Reusability.
- Versioning.

2.2.1. 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 use this approach, you 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.

Additionally, not all SOAP stacks generate the same web service contract from a Java contract. This means that 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 have to be instructed to obtain the new contract and potentially change their code to accommodate for any changes in the contract.

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

2.2.2. Performance

When a Java object 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, and so on. In the end, half of the objects on the heap in your virtual machine might be converted into XML, which results 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.

2.2.3. Reusability

Defining your schema in a separate file lets you reuse that file in different scenarios. Consider the definition of an `AirportCode` in a file called `airline.xsd`:

```
<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, by using an `import` statement.

2.2.4. 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 yet migrated.

If using contract-first, we can have a looser coupling between contract and implementation. Such a looser coupling lets us 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

This tutorial shows you how to write contract-first web services — that is, how to develop web services that start with the XML Schema or WSDL contract first followed by the Java code second. Spring-WS focuses on this development style, and this tutorial should 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 focuses on implementing this contract with Spring-WS .

The most important thing when doing contract-first web service development is to 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. Java being used to implement the web service is an implementation detail.

In this tutorial, we 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.

3.1. Messages

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

3.1.1. 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 saves a lot of parsing hassle. We have also added a namespace to the element, to make sure our elements can be used within other XML documents.

3.1.2. 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://example.com/employees/schemas`.

3.1.3. HolidayRequest

Both the `holiday` element and the `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. What matters is that all the data is there. In fact, the data is the only thing that is important: we take a data-driven approach.

3.2. Data Contract

Now that we have seen some examples of the XML data that we can 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 are easier than XML Schema. Unfortunately, they are not so widely supported across platforms. As a result, we 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 to

generate a schema that validates them all. The end result certainly needs to be polished up, but it is a great starting point.

Using the sample described earlier, 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 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 want to accept only a `<HolidayRequest/>`. By removing the wrapping element tags (thus keeping the types) and inlining the results, we can accomplish this, as follows:

```
<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 message to validate:

```
<HolidayRequest xmlns="http://mycompany.com/hr/schemas">
    <Holiday>
        <StartDate>this is not a date</StartDate>
        <EndDate>neither is this</EndDate>
    </Holiday>
    PlainText Section qName:lineannotation level:4, chunks:[<, !-- ... --, >]
attrs:[:]
</HolidayRequest>
```

Clearly, we must make sure that the start and end date are really dates. XML Schema has an excellent built-in `date` type that we can use. We also change the `NCName` s to `string` instances. 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 the following listing:

```
<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"/> ①
                <xs:element name="Employee" type="hr:EmployeeType"/> ①
            </xs:all>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="HolidayType">
        <xs:sequence>
            <xs:element name="StartDate" type="xs:date"/> ②
            <xs:element name="EndDate" type="xs:date"/> ②
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="EmployeeType">
        <xs:sequence>
            <xs:element name="Number" type="xs:integer"/>
            <xs:element name="FirstName" type="xs:string"/> ③
            <xs:element name="LastName" type="xs:string"/> ③
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

① `all` tells the XML parser that the order of `<Holiday/>` and `<Employee/>` is not significant.

② We use the `xs:date` data type (which consist of a year, a month, and a day) for `<StartDate/>` and `<EndDate/>`.

③ `xs:string` is used for the first and last names.

We store this file as `hr.xsd`.

3.3. 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 Implementing the Endpoint. The remainder of this section shows how to write WSDL by hand. You may want to skip to the next section.

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

```
<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, 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 message finishes 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 have just defined) and a `service` (which tells the client where to invoke it).

Adding a concrete part is pretty standard. To do so, 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 works), and determine the `location` URL where you want the request to arrive (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"
①
                schemaLocation="hr.xsd"/>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="HolidayRequest">
②
        <wsdl:part element="schema:HolidayRequest" name="HolidayRequest"/>
③
    </wsdl:message>
    <wsdl:portType name="HumanResource">
④
        <wsdl:operation name="Holiday">
            <wsdl:input message="tns:HolidayRequest" name="HolidayRequest"/>
②
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HumanResourceBinding" type="tns:HumanResource">
④⑤
        <soap:binding style="document"
⑥
            transport="http://schemas.xmlsoap.org/soap/http"/>
⑦
        <wsdl:operation name="Holiday">
            <soap:operation soapAction="http://mycompany.com/RequestHoliday"/>
⑧
            <wsdl:input name="HolidayRequest">
                <soap:body use="literal"/>
⑥
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HumanResourceService">
        <wsdl:port binding="tns:HumanResourceBinding" name="HumanResourcePort">
⑤
            <soap:address location="http://localhost:8080/holidayService/"/>
⑨
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

① We import the schema defined in Data Contract.

② We define the `HolidayRequest` message, which gets used in the `portType`.

③ The `HolidayRequest` type is defined in the schema.

④ We define the `HumanResource` port type, which gets used in the `binding`.

⑤ We define the `HumanResourceBinding` binding, which gets used in the `port`.

⑥ We use a document/literal style.

⑦ The literal `http://schemas.xmlsoap.org/soap/http` signifies a HTTP transport.

⑧ The `soapAction` attribute signifies the `SOAPAction` HTTP header that will be sent with every request.

⑨ The `http://localhost:8080/holidayService/` address is the URL where the web service can be invoked.

The preceding listing shows the final WSDL. We describe how to implement the resulting schema and WSDL in the next section.

3.4. Creating the project

In this section, we use Maven 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 set up our HolidayService.

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

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

The preceding command creates a new directory called `holidayService`. In this directory is a `src/main/webapp` directory, which contains the root of the WAR file. You can 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 special 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 preceding `WEB-INF/web.xml` file, you also need another, Spring-WS-specific, configuration file, named `WEB-INF/spring-ws-servlet.xml`. This file contains all the Spring-WS-specific beans, such as `EndPoints` and `WebServiceMessageReceivers` 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 define a `MessageDispatcherServlet` with the name `dynamite`, the name of the Spring-WS-specific configuration file becomes `WEB-INF/dynamite-servlet.xml`.

Once you had the project structure created, you can put the schema and the WSDL from the previous section into `WEB-INF/` folder.

3.5. Implementing the Endpoint

In Spring-WS, you implement endpoints to handle incoming XML messages. An endpoint is typically created by annotating a class with the `@Endpoint` annotation. In this endpoint class, you can create one or more methods that handle incoming request. The method signatures can be quite flexible. You can include almost any sort of parameter type related to the incoming XML message, as we explain later in this chapter.

3.5.1. Handling the XML Message

In this sample application, we use JDom 2 to handle the XML message. We also use XPath, because it lets us select particular parts of the XML JDOM tree without requiring strict schema conformance.

The following listing shows the class that defines our holiday endpoint:

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

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
```

```
import com.mycompany.hr.service.HumanResourceService;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.Namespace;
import org.jdom2.filter.Filters;
import org.jdom2.xpath.XPathExpression;
import org.jdom2.xpath.XPathFactory;

@Endpoint
①
public class HolidayEndpoint {

    private static final String NAMESPACE_URI = "http://mycompany.com/hr/schemas";

    private XPathExpression<Element> startDateExpression;

    private XPathExpression<Element> endDateExpression;

    private XPathExpression<Element> firstNameExpression;

    private XPathExpression<Element> lastNameExpression;

    private HumanResourceService humanResourceService;

    @Autowired
②
    public HolidayEndpoint(HumanResourceService humanResourceService) throws
JDOMException {
        this.humanResourceService = humanResourceService;

        Namespace namespace = Namespace.getNamespace("hr", NAMESPACE_URI);
        XPathFactory xPathFactory = XPathFactory.instance();
        startDateExpression = xPathFactory.compile("//hr:StartDate",
Filters.element(), null, namespace);
        endDateExpression = xPathFactory.compile("//hr:EndDate",
Filters.element(), null, namespace);
        firstNameExpression = xPathFactory.compile("//hr:FirstName",
Filters.element(), null, namespace);
        lastNameExpression = xPathFactory.compile("//hr:LastName",
Filters.element(), null, namespace);
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "HolidayRequest")
③
    public void handleHolidayRequest(@RequestPayload Element holidayRequest)
throws Exception {④
        Date startDate = parseDate(startDateExpression, holidayRequest);
        Date endDate = parseDate(endDateExpression, holidayRequest);
        String name = firstNameExpression.evaluateFirst(holidayRequest).getText()
+ " " + lastNameExpression.evaluateFirst(holidayRequest).getText();
```

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

    private Date parseDate(XPathExpression<Element> expression, Element element)
throws ParseException {
        Element result = expression.evaluateFirst(element);
        if (result != null) {
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
            return dateFormat.parse(result.getText());
        } else {
            throw new IllegalArgumentException("Could not evaluate [" + expression
+ "] on [" + element + "]");
        }
    }

}
```

① The `HolidayEndpoint` is annotated with `@Endpoint`. This marks the class as a special sort of `@Component`, suitable for handling XML messages in Spring-WS, and also makes it eligible for suitable for component scanning.

② The `HolidayEndpoint` requires the `HumanResourceService` business service to operate, so we inject the dependency in the constructor and annotate it with `@Autowired`. Next, we set up XPath expressions by using the JDOM2 API. There are four expressions: `//hr:StartDate` for extracting the `<StartDate>` text value, `//hr:EndDate` for extracting the end date, and two for extracting the names of the employee.

③ The `@PayloadRoot` annotation tells Spring-WS that the `handleHolidayRequest` method is suitable for handling XML messages. The sort of message that this method can handle is indicated by the annotation values. In this case, it can handle XML elements that have the `HolidayRequest` local part and the `http://mycompany.com/hr/schemas` namespace. More information about mapping messages to endpoints is provided in the next section.

④ The `handleHolidayRequest(..)` method is the main handling method, which gets passed the `<HolidayRequest/>` element from the incoming XML message. The `@RequestPayload` annotation indicates that the `holidayRequest` parameter should be mapped to the payload of the request message. We use the XPath expressions to extract the string values from the XML messages and convert these values to `Date` objects by using a `SimpleDateFormat` (the `parseData` method). With these values, we invoke a method on the business service. Typically, this results in a database transaction being started and some records being altered in the database. Finally, we define a `void` return type, which indicates to Spring-WS that we do not want to send a response message. If we want a response message, we could return a JDOM Element to represent 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, as explained in the next chapter. 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 do not need 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></version>
    </dependency>
    <dependency>
        <groupId>jdom</groupId>
        <artifactId>jdom</artifactId>
        <version>2.0.1</version>
    </dependency>
    <dependency>
        <groupId>jaxen</groupId>
        <artifactId>jaxen</artifactId>
        <version>1.1</version>
    </dependency>
</dependencies>
```

Here is how we would configure these classes in our `spring-ws-servlet.xml` Spring XML configuration file by using component scanning. We also instruct Spring-WS to use annotation-driven endpoints, with the `<sws:annotation-driven>` element.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:sws="http://www.springframework.org/schema/web-services"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/web-services
http://www.springframework.org/schema/web-services/web-services-2.0.xsd
  http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan base-package="com.mycompany.hr"/>

  <sws:annotation-driven/>

</beans>
```

3.5.2. Routing the Message to the Endpoint

As part of writing the endpoint, we also used the `@PayloadRoot` annotation to indicate which sort of messages can be handled by the `handleHolidayRequest` method. In Spring-WS, this process is the responsibility of an `EndpointMapping`. Here, we route messages based on their content by using a `PayloadRootAnnotationMethodEndpointMapping`. The following listing shows the annotation we used earlier:

```
@PayloadRoot(namespace = "http://mycompany.com/hr/schemas", localPart =
"HolidayRequest")
```

The annotation shown in the preceding example basically means that whenever an XML message is received with the namespace `http://mycompany.com/hr/schemas` and the `HolidayRequest` local name, it is routed to the `handleHolidayRequest` method. By using the `<sws:annotation-driven>` element in our configuration, we enable the detection of the `@PayloadRoot` annotations. It is possible (and quite common) to have multiple, related handling methods in an endpoint, each of them handling different XML messages.

There are other ways to map endpoints to XML messages, which is described in the next chapter.

3.5.3. Providing the Service and Stub implementation

Now that we have the endpoint, we need `HumanResourceService` and its implementation for use by `HolidayEndpoint`. The following listing shows the `HumanResourceService` interface:

```
package com.mycompany.hr.service;

import java.util.Date;

public interface HumanResourceService {
    void bookHoliday(Date startDate, Date endDate, String name);
}
```

For tutorial purposes, we use a simple stub implementation of the `HumanResourceService`:

```
package com.mycompany.hr.service;

import java.util.Date;

import org.springframework.stereotype.Service;

@Service                                                                    ①
public class StubHumanResourceService implements HumanResourceService {
```

```
    public void bookHoliday(Date startDate, Date endDate, String name) {
        System.out.println("Booking holiday for [" + startDate + "-" + endDate +
"] for [" + name + "] ");
    }
}
```

① The `StubHumanResourceService` is annotated with `@Service`. This marks the class as a business facade, which makes this a candidate for injection by `@Autowired` in `HolidayEndpoint`.

3.6. Publishing the WSDL

Finally, we need to publish the WSDL. As stated in Service Contract, we do not need to write a WSDL ourselves. Spring-WS can generate one based on some conventions. Here is how we define the generation:

```
<sws:dynamic-wsdl id="holiday"                                    ①
    portTypeName="HumanResource"                                  ③
    locationUri="/holidayService/"                                ④
    targetNamespace="http://mycompany.com/hr/definitions">        ⑤
  <sws:xsd location="/WEB-INF/hr.xsd"/>                           ②
</sws:dynamic-wsdl>
```

① The `id` determines the URL where the WSDL can be retrieved. In this case, the `id` is `holiday`, which means that the WSDL can be retrieved as `holiday.wsdl` in the servlet context. The full URL is `http://localhost:8080/holidayService/holiday.wsdl`.

② Next, we set the WSDL port type to be `HumanResource`.

③ We set the location where the service can be reached: `/holidayService/`. We use a relative URI, and we instruct the framework to transform it dynamically to an absolute URI. Hence, if the service is deployed to different contexts, we do not have to change the URI manually. For more information, see the section called “Automatic WSDL exposure”. For the location transformation to work, we need to add an init parameter to `spring-ws` servlet in `web.xml` (shown in the next listing).

④ We define the target namespace for the WSDL definition itself. Setting this attribute is not required. If not set, the WSDL has the same namespace as the XSD schema.

⑤ The `xsd` element refers to the human resource schema we defined in Data Contract. We placed the schema in the `WEB-INF` directory of the application.

The following listing shows how to add the init parameter:

```
<init-param>
  <param-name>transformWsdlLocations</param-name>
  <param-value>true</param-value>
```

```
</init-param>
```

You can create a WAR file by using `mvn install`. If you deploy the application (to Tomcat, Jetty, and so on) and point your browser at this location, you 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. If you wish to continue, 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.

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 a chapter on using WS-Security.

Chapter 4. Shared components

This chapter explores the components that are shared between client- and server-side Spring-WS development. These interfaces and classes represent the building blocks of Spring-WS, so you need to understand what they do, even if you do not use them directly.

4.1. Web Service Messages

This section describes the messages and message factories that Spring-WS uses.

4.1.1. `WebServiceMessage`

One of the core interfaces of Spring-WS 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 or Result implementation	Wrapped 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`

`SoapMessage` is a subclass of `WebServiceMessage`. It contains SOAP-specific methods, such as getting SOAP Headers, SOAP Faults, and so on. Generally, your code should not be dependent on `SoapMessage`, because the content of the SOAP Body (the payload of the message) can be obtained by using `getPayloadSource()` and `getPayloadResult()` in the `WebServiceMessage`. Only when it is necessary to perform SOAP-specific actions (such as adding a header, getting an attachment, and so on) 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 from an input stream. One concrete implementations of `WebServiceMessageFactory` is provided based on SAAJ, the SOAP with Attachments API for Java.

`SaajSoapMessageFactory`

The `SaajSoapMessageFactory` uses the SOAP with Attachments API for Java (SAAJ) 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[1]
IBM WebSphere 6	1.2
SUN Glassfish 1	1.3

[1]Weblogic 9 has a known bug in the SAAJ 1.2 implementation: it implements all the 1.2 interfaces but throws an `UnsupportedOperationException` when called. Spring-WS has a workaround: It uses SAAJ 1.1 when operating on WebLogic 9.

Additionally, Java SE 6 includes SAAJ 1.3. You can wire up a `SaajSoapMessageFactory` as follows:

```
<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 performant.

SOAP 1.1 or 1.2

`SaajSoapMessageFactory` has a `soapVersion` property, where you can inject a `SoapVersion` constant. By default, the version is 1.1, but you can set it to 1.2:

```
<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 preceding example, we define a `SaajSoapMessageFactory` that accepts only SOAP 1.2 messages.

WARNING

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 effectively deprecated, though they still work.

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 1.1 is currently 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-WS, 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 message content.

However, it is sometimes necessary to get access to the underlying transport, either on the client or the server side. For this, Spring-WS 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:

```
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-WS has two ways to use XPath within your application: the faster `XPathExpression` or the more flexible `XPathOperations`.

4.3.1. `XPathExpression`

The `XPathExpression` is an abstraction over a compiled XPath expression, such as the Java 5 `javax.xml.xpath.XPathExpression` interface or the Jaxen `XPath` class. To construct an expression in an application context, you can use `XPathExpressionFactoryBean`. The following example 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 preceding expression does not use namespaces, but we could set those by 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 to 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 = contactExpression.evaluate(document,
        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());
            }
        });
      PlainText Section qName; // do something with the list of Contact objects
   }
}
```

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

4.3.2. `XPathOperations`

The `XPathExpression` lets you evaluate only a single, pre-compiled expression. A more flexible, though slower, alternative is the `XPathOperations`. This class follows the common template pattern used throughout Spring (`JdbcTemplate`, `JmsTemplate`, and others). The following listing shows 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
    }

}
```

4.4. Message Logging and Tracing

When developing or debugging a web service, it can be quite useful to look at the content of a (SOAP) message when it arrives, or before it is sent. Spring-WS offer this functionality, through the standard Commons Logging interface.

To log all server-side messages, set the `org.springframework.ws.server.MessageTracing` logger level to `DEBUG` or `TRACE`. On the `DEBUG` level, only the payload root element is logged. On the `TRACE` level, the entire message content is logged. If you want to log only sent messages, use the `org.springframework.ws.server.MessageTracing.sent` logger. Similarly, you can use `org.springframework.ws.server.MessageTracing.received` to log only received messages.

On the client-side, similar loggers exist: `org.springframework.ws.client.MessageTracing.sent` and

`org.springframework.ws.client.MessageTracing.received`.

The following example of a `log4j2.properties` configuration file logs the full content of sent messages on the client side and only the payload root element for client-side received messages. On the server-side, the payload root is logged for both sent and received messages:

```
appender.console.name=STDOUT
appender.console.type=Console
appender.console.layout.type=PatternLayout
appender.console.layout.pattern=%-5p [%c{3}] %m%n

rootLogger=DEBUG,STDOUT
logger.org.springframework.ws.client.MessageTracing.sent=TRACE
logger.org.springframework.ws.client.MessageTracing.received=DEBUG
logger.org.springframework.ws.server.MessageTracing=DEBUG
```

With this configuration, a typical output is:

```
TRACE [client.MessageTracing.sent] Sent request [<SOAP-ENV:Envelope xmlns:SOAP-
ENV="...
DEBUG [server.MessageTracing.received] Received request [SaajSoapMessage
{http://example.com}request] ...
DEBUG [server.MessageTracing.sent] Sent response [SaajSoapMessage
{http://example.com}response] ...
DEBUG [client.MessageTracing.received] Received response [SaajSoapMessage
{http://example.com}response] ...
```

Chapter 5. Creating a Web service with Spring-WS

Spring-WS server-side support is designed around a `MessageDispatcher` that dispatches incoming messages to endpoints, with configurable endpoint mappings, response generation, and endpoint interception. Endpoints are typically annotated with the `@Endpoint` annotation and have one or more handling methods. These methods handle incoming XML request messages by inspecting parts of the message (typically the payload) and create some sort of response. You can annotate the method with another annotation, typically `@PayloadRoot`, to indicate what sort of messages it can handle.

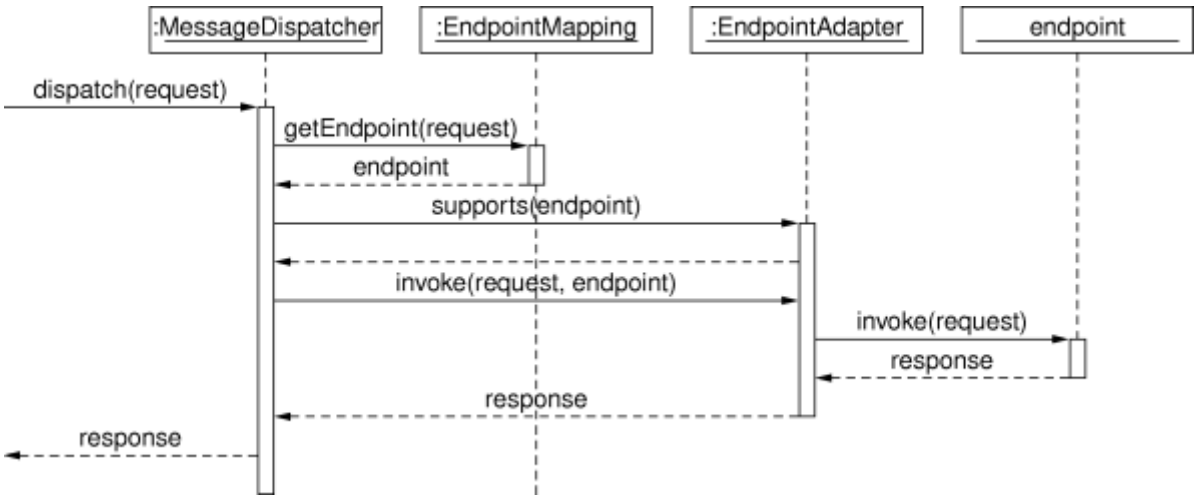
Spring-WS XML handling is extremely flexible. An endpoint can choose from a large number of XML handling libraries supported by Spring-WS, including:

- The DOM family: W3C DOM, JDOM, dom4j, and XOM.
- SAX or StAX: For faster performance.
- XPath: To extract information from the message.
- Marshalling techniques (JAXB, Castor, XMLBeans, JiBX, or XStream): To convert the XML to objects and vice-versa.

5.1. The `MessageDispatcher`

The server-side of Spring-WS is designed around a central class that dispatches incoming XML messages to endpoints. Spring-WS `MessageDispatcher` is extremely flexible, letting you 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 following sequence diagram shows the processing and dispatching flow of the `MessageDispatcher`:



When a `MessageDispatcher` is set up for use and a request comes in for that specific dispatcher, the `MessageDispatcher` starts processing the request. The following process describes how

`MessageDispatcher` handles a request:

1. The configured `EndpointMapping(s)` is searched for an appropriate endpoint. If an endpoint is found, the invocation chain associated with the endpoint (pre-processors, post-processors, and endpoints) is invoked to create a response.
2. An appropriate adapter is found 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 the 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 lets you 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 automatically detects all the types that are registered in the application context. You should set these properties only when you need to override detection.

The message dispatcher operates on a message context and not on a 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 `MessageDispatcherServlet`.

5.2. Transports

Spring-WS supports multiple transport protocols. The most common is the HTTP transport, for which a custom servlet is supplied, but you can also send messages over JMS and even email.

5.2.1. `MessageDispatcherServlet`

The `MessageDispatcherServlet` is a standard `Servlet` that conveniently extends from the standard Spring Web `DispatcherServlet` and wraps a `MessageDispatcher`. As a result, 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 must be mapped by a URL mapping in the same `web.xml` file. This is standard JavaEE servlet configuration. The following example shows such a `MessageDispatcherServlet` declaration and mapping:

```
<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 preceding example, all requests are handled by the `spring-ws MessageDispatcherServlet`. This is only the first step in setting up Spring-WS, 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 looks for a file named `[servlet-name]-servlet.xml` in the `WEB-INF` directory of your web application and creates the beans defined there in a Spring container. In the preceding example, it looks for `/WEB-INF/spring-ws-servlet.xml`. This file contains all the Spring-WS beans, such as endpoints, marshallers, and so on.

As an alternative for `web.xml`, you can configure Spring-WS programmatically. For this purpose, Spring-WS provides a number of abstract base classes that extend the `WebApplicationInitializer` interface found in the Spring Framework. If you also use `@Configuration` classes for your bean definitions, you should extend the `AbstractAnnotationConfigMessageDispatcherServletInitializer`:

```
public class MyServletInitializer
    extends AbstractAnnotationConfigMessageDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{MyRootConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{MyEndpointConfig.class};
    }

}
```

In the preceding example, we tell Spring that endpoint bean definitions can be found in the `MyEndpointConfig` class (which is a `@Configuration` class). Other bean definitions (typically services, repositories, and so on) can be found in the `MyRootConfig` class. By default, the `AbstractAnnotationConfigMessageDispatcherServletInitializer` maps the servlet to two patterns: `/services` and `*.wsdl`, though you can change this by overriding the `getServletMappings()` method. For more details on the programmatic configuration of the `MessageDispatcherServlet`, refer to the

Javadoc of `AbstractMessageDispatcherServletInitializer` and `AbstractAnnotationConfigMessageDispatcherServletInitializer`.

Automatic WSDL exposure

The `MessageDispatcherServlet` automatically detects any `WsdlDefinition` beans defined in its Spring container. All the `WsdlDefinition` detected beans are also exposed through a `WsdlDefinitionHandlerAdapter`. This is a convenient way to expose your WSDL to clients by defining some beans.

By way of an example, consider the following `<static-wsdl>` definition, defined in the Spring-WS configuration file (`/WEB-INF/[servlet-name]-servlet.xml`). Take notice of the value of the `id` attribute, because it is used when exposing the WSDL.

```
<sws:static-wsdl id="orders" location="orders.wsdl"/>
```

Alternatively, it can be a `@Bean` method in a `@Configuration` class:

```
@Bean
public SimpleWsdl11Definition orders() {
    return new SimpleWsdl11Definition(new ClassPathResource("orders.wsdl"));
}
```

You can access the WSDL defined in the `orders.wsdl` file on the classpath through `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
```

NOTE All `WsdlDefinition` bean definitions are exposed by the `MessageDispatcherServlet` under their bean name with a suffix of`.wsdl`. So, if the bean name is `echo`, the host name is `server`, and the Servlet context (war name) is `spring-ws`, the WSDL can be found at `http://server/spring-ws/echo.wsdl`.

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

Note that this `location` transformation feature is off by default. To switch this feature on, you need to specify an initialization parameter to the `MessageDispatcherServlet`:

```
<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>
```

If you use `AbstractAnnotationConfigMessageDispatcherServletInitializer`, enabling transformation is as simple as overriding the `isTransformWsdlLocations()` method to return `true`.

Consult the class-level Javadoc on the `WsdlDefinitionHandlerAdapter` class to learn more about the whole transformation process.

As an alternative to writing the WSDL by hand and exposing it with `<static-wsdl>`, Spring-WS can also generate a WSDL from an XSD schema. This is the approach shown in Publishing the WSDL. The next application context snippet shows how to create such a dynamic WSDL file:

```
<sws:dynamic-wsdl id="orders"
    portTypeName="Orders"
    locationUri="http://localhost:8080/ordersService/">
  <sws:xsd location="Orders.xsd"/>
</sws:dynamic-wsdl>
```

Alternatively, you can use the Java `@Bean` method:

```
@Bean
public DefaultWsdl11Definition orders() {
    DefaultWsdl11Definition definition = new DefaultWsdl11Definition();
    definition.setPortTypeName("Orders");
    definition.setLocationUri("http://localhost:8080/ordersService/");
    definition.setSchema(new SimpleXsdSchema(new
```

```
ClassPathResource("Orders.xsd")));
    return definition;
}
```

The `<dynamic-wsdl>` element depends on the `DefaultWsdl11Definition` class. This definition class uses WSDL providers in the `org.springframework.ws.wsdl.wsdl11.provider` package and the `ProviderBasedWsdl4jDefinition` class to generate a WSDL the first time it is requested. See the class-level Javadoc of these classes to see how you can extend this mechanism, if necessary.

The `DefaultWsdl11Definition` (and therefore, the `<dynamic-wsdl>` tag) builds a WSDL from an XSD schema by using conventions. It iterates over all `element` elements found in the schema and creates a `message` for all elements. Next, it creates a WSDL `operation` for all messages 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` attributes on `<dynamic-wsdl />`, respectively. It also builds a `portType`, `binding`, and `service` based on the operations.

For instance, if our `Orders.xsd` schema defines the `GetOrdersRequest` and `GetOrdersResponse` elements, `<dynamic-wsdl>` creates a `GetOrdersRequest` and `GetOrdersResponse` message and a `GetOrders` operation, which is put in a `Orders` port type.

To use multiple schemas, either by includes or imports, you can put Commons XMLSchema on the class path. If Commons XMLSchema is on the class path, the `<dynamic-wsdl>` element follows all XSD imports and includes and inlines them in the WSDL as a single XSD. With Java config, you can use `CommonsXsdSchemaCollection` as shown in the following example:

```
@Bean
public DefaultWsdl11Definition ordersWsdlDefinition() throws Exception {
    DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
    // ... configuration
    CommonsXsdSchemaCollection schemas = new CommonsXsdSchemaCollection(
            new ClassPathResource("xsd/order/main.xsd"));
    schemas.setInline(true);
    wsdl11Definition.setSchemaCollection(schemas);
    return wsdl11Definition;
}
```

This greatly simplifies the deployment of the schemas, while still making it possible to edit them separately.

WARNING	Even though it can be handy to create the WSDL at runtime from your XSDs, there are a couple of drawbacks to this approach. First, though we try to keep the WSDL generation process consistent between releases, there is still the possibility that it changes (slightly). Second, the generation is a bit slow, though, once generated, the WSDL is cached for later reference.

Therefore, you should use `<dynamic-wsdl>` only during the development stages of your project. We

recommend using your browser to download the generated WSDL, store it in the project, and expose it with `<static-wsdl>`. This is the only way to be really sure that the WSDL does not change over time.

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 delegate only 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.WebServiceMessageReceiverHandlerAdapt
er"/>

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

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

    ...

    <bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandler
Adapter"/>

</beans>
```

Note that, by explicitly 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 `RequestMappingHandlerAdapter` at the end.

In a similar fashion, you can wire a `WsdlDefinitionHandlerAdapter` to make sure the `DispatcherServlet` can handle implementations of the `WsdlDefinition` interface:

```
<beans>

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

    <bean
```

```
class="org.springframework.ws.transport.http.WsdlDefinitionHandlerAdapter"/>

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
           <props>
             <prop key="*.wsdl">myServiceDefinition</prop>
           </props>
        </property>
        <property name="defaultHandler" ref="messageDispatcher"/>
    </bean>

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

    <bean id="myServiceDefinition"
class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">
       <prop name="wsdl" value="/WEB-INF/myServiceDefinition.wsdl"/>
    </bean>

    ...

</beans>
```

5.2.3. JMS transport

Spring-WS supports server-side JMS handling through the JMS functionality provided in the Spring framework. Spring-WS provides the `WebServiceMessageListener` to plug in to a `MessageListenerContainer`. This message listener requires a `WebServiceMessageFactory` and `MessageDispatcher` to operate. The following configuration example shows this:

```
<beans>

    <bean id="connectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL"
value="vm://localhost?broker.persistent=false"/>
    </bean>

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

    <bean
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
        <property name="connectionFactory" ref="connectionFactory"/>
        <property name="destinationName" value="RequestQueue"/>
        <property name="messageListener">
            <bean
class="org.springframework.ws.transport.jms.WebServiceMessageListener">
```

```
                <property name="messageFactory" ref="messageFactory"/>
                <property name="messageReceiver" ref="messageDispatcher"/>
            </bean>
        </property>
    </bean>

    <bean id="messageDispatcher"
class="org.springframework.ws.soap.server.SoapMessageDispatcher">
        <property name="endpointMappings">
            <bean

class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodE
ndpointMapping">
                <property name="defaultEndpoint">
                    <bean class="com.example.MyEndpoint"/>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

5.2.4. Email Transport

In addition to HTTP and JMS, Spring-WS also provides server-side email handling. This functionality is provided through the `MailMessageReceiver` class. This class monitors a POP3 or IMAP folder, converts the email to a `WebServiceMessage`, and sends any response by using SMTP. You can configure the host names through the `storeUri`, which indicates the mail folder to monitor for requests (typically a POP3 or IMAP folder), and a `transportUri`, which indicates the server to use for sending responses (typically an SMTP server).

You can configure how the `MailMessageReceiver` monitors incoming messages with a pluggable strategy: the `MonitoringStrategy`. By default, a polling strategy is used, where the incoming folder is polled for new messages every five minutes. You can change this interval by setting the `pollingInterval` property on the strategy. By default, all `MonitoringStrategy` implementations delete the handled messages. You can change this setting by setting the `deleteMessages` property.

As an alternative to the polling approaches, which are quite inefficient, there is a monitoring strategy that uses IMAP IDLE. The IDLE command is an optional expansion of the IMAP email protocol that lets the mail server send new message updates to the `MailMessageReceiver` asynchronously. If you use an IMAP server that supports the IDLE command, you can plug the `ImapIdleMonitoringStrategy` into the `monitoringStrategy` property.

The following piece of configuration shows how to use the server-side email support, overriding the default polling interval to check every 30 seconds (30.000 milliseconds):

```
<beans>
```

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

    <bean id="messagingReceiver"
class="org.springframework.ws.transport.mail.MailMessageReceiver">
        <property name="messageFactory" ref="messageFactory"/>
        <property name="from" value="Spring-WS SOAP Server
&lt;server@example.com&gt;"/>
        <property name="storeUri"
value="imap://server:s04p@imap.example.com/INBOX"/>
        <property name="transportUri" value="smtp://smtp.example.com"/>
        <property name="messageReceiver" ref="messageDispatcher"/>
        <property name="monitoringStrategy">
            <bean
class="org.springframework.ws.transport.mail.monitor.PollingMonitoringStrategy">
                <property name="pollingInterval" value="30000"/>
            </bean>
        </property>
    </bean>

    <bean id="messageDispatcher"
class="org.springframework.ws.soap.server.SoapMessageDispatcher">
        <property name="endpointMappings">
            <bean

class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodE
ndpointMapping">
                <property name="defaultEndpoint">
                    <bean class="com.example.MyEndpoint"/>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

5.2.5. Embedded HTTP Server transport

NOTE This should only be used for testing purposes.

Spring-WS provides a transport based on Sun's JRE HTTP server. The embedded HTTP Server is a standalone server that is simple to configure. It offers a lighter alternative to conventional servlet containers.

When using the embedded HTTP server, you need no external deployment descriptor (`web.xml`). You need only define an instance of the server and configure it to handle incoming requests. `SimpleHttpServerFactoryBean` wires things up, and the most important property is `contexts`, which maps context paths to corresponding `HttpHandler` instances.

Spring-WS provides two implementations of the `HttpHandler` interface: `WsdlDefinitionHttpHandler`

and `WebServiceMessageReceiverHttpHandler`. The former maps an incoming GET request to a `WsdlDefinition`. The latter is responsible for handling POST requests for web services messages and, thus, needs a `WebServiceMessageFactory` (typically a `SaajSoapMessageFactory`) and a `WebServiceMessageReceiver` (typically the `SoapMessageDispatcher`) to accomplish its task.

To draw parallels with the servlet world, the `contexts` property plays the role of servlet mappings in `web.xml` and the `WebServiceMessageReceiverHttpHandler` is the equivalent of a `MessageDispatcherServlet`.

The following snippet shows a configuration example of the HTTP server transport:

```
<beans>

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

    <bean id="messageReceiver"
class="org.springframework.ws.soap.server.SoapMessageDispatcher">
        <property name="endpointMappings" ref="endpointMapping"/>
    </bean>

    <bean id="endpointMapping"
class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodE
ndpointMapping">
        <property name="defaultEndpoint" ref="stockEndpoint"/>
    </bean>

    <bean id="httpServer"
class="org.springframework.ws.transport.http.SimpleHttpServerFactoryBean">
        <property name="contexts">
            <map>
                <entry key="/StockService.wsdl" value-ref="wsdlHandler"/>
                <entry key="/StockService" value-ref="soapHandler"/>
            </map>
        </property>
    </bean>

    <bean id="soapHandler"
class="org.springframework.ws.transport.http.WebServiceMessageReceiverHttpHandler"
>
        <property name="messageFactory" ref="messageFactory"/>
        <property name="messageReceiver" ref="messageReceiver"/>
    </bean>

    <bean id="wsdlHandler"
class="org.springframework.ws.transport.http.WsdlDefinitionHttpHandler">
        <property name="definition" ref="wsdlDefinition"/>
    </bean>
</beans>
```

5.2.6. XMPP transport

Spring-WS has support for XMPP, otherwise known as Jabber. The support is based on the Smack library.

Spring-WS support for XMPP is very similar to the other transports: There is a a `XmppMessageSender` for the `WebServiceTemplate` and a `XmppMessageReceiver` to use with the `MessageDispatcher`.

The following example shows how to set up the server-side XMPP components:

```
<beans>

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

    <bean id="connection"
class="org.springframework.ws.transport.xmpp.support.XmppConnectionFactoryBean">
        <property name="host" value="jabber.org"/>
        <property name="username" value="username"/>
        <property name="password" value="password"/>
    </bean>

    <bean id="messagingReceiver"
class="org.springframework.ws.transport.xmpp.XmppMessageReceiver">
        <property name="messageFactory" ref="messageFactory"/>
        <property name="connection" ref="connection"/>
        <property name="messageReceiver" ref="messageDispatcher"/>
    </bean>

    <bean id="messageDispatcher"
class="org.springframework.ws.soap.server.SoapMessageDispatcher">
        <property name="endpointMappings">
            <bean

class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodE
ndpointMapping">
                <property name="defaultEndpoint">
                    <bean class="com.example.MyEndpoint"/>
                </property>
            </bean>
        </property>
    </bean>

</beans>
```

5.2.7. MTOM

MTOM is the mechanism for sending binary data to and from Web Services. You can look at how to

implement this with Spring WS through the MTOM sample.

5.3. Endpoints

Endpoints are the central concept in Spring-WS 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 (typically) invoke a method on the business service. The result of that service invocation is represented as a response message. Spring-WS has a wide variety of endpoints and uses various ways to handle the XML message and to create a response.

You can create an endpoint by annotating a class with the `@Endpoint` annotation. In the class, you define one or more methods that handle the incoming XML request, by using a wide variety of parameter types (such as DOM elements, JAXB2 objects, and others). You can indicate the sort of messages a method can handle by using another annotation (typically `@PayloadRoot`).

Consider the following sample endpoint:

```
package samples;

import org.w3c.dom.Element;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.soap.SoapHeader;

@Endpoint ①
public class AnnotationOrderEndpoint {

  private final OrderService orderService;

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

  @PayloadRoot(localPart = "order", namespace = "http://samples") ④
  public void order(@RequestPayload Element orderElement) { ②
    Order order = createOrder(orderElement);
    orderService.createOrder(order);
  }

  @PayloadRoot(localPart = "orderRequest", namespace = "http://samples") ④
  @ResponsePayload
  public Order getOrder(@RequestPayload OrderRequest orderRequest, SoapHeader
header) { ③
    checkSoapHeaderForSomething(header);
    return orderService.getOrder(orderRequest.getId());
  }
```

```
}
```

① The class is annotated with `@Endpoint`, marking it as a Spring-WS endpoint.

② The `order` method takes an `Element` (annotated with `@RequestPayload`) as a parameter. This means that the payload of the message is passed on this method as a DOM element. The method has a `void` return type, indicating that no response message is sent. For more information about endpoint methods, see `@Endpoint` handling methods.

③ The `getOrder` method takes an `OrderRequest` (also annotated with `@RequestPayload`) as a parameter. This parameter is a JAXB2-supported object (it is annotated with `@XmlRootElement`). This means that the payload of the message is passed to this method as a unmarshalled object. The `SoapHeader` type is also given as a parameter. On invocation, this parameter contains the SOAP header of the request message. The method is also annotated with `@ResponsePayload`, indicating that the return value (the `Order`) is used as the payload of the response message. For more information about endpoint methods, see `@Endpoint` handling methods.

④ The two handling methods of this endpoint are marked with `@PayloadRoot`, indicating what sort of request messages can be handled by the method: the `getOrder` method is invoked for requests with a `orderRequest` local name and a `http://samples` namespace URI. The order method is invoked for requests with a `order` local name. For more information about `@PayloadRoot`, see Endpoint mappings.

To enable the support for `@Endpoint` and related Spring-WS annotations, you need to add the following to your Spring application context:

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

  <sws:annotation-driven />

</beans>
```

Alternatively, if you use `@Configuration` classes instead of Spring XML, you can annotate your configuration class with `@EnableWs`:

```
@EnableWs
@Configuration
public class EchoConfig {
```

```
    // @Bean definitions go here

}
```

To customize the `@EnableWs` configuration, you can implement `WsConfigurer` and override individual methods:

```
@Configuration
@EnableWs
public class EchoConfig extends WsConfigurerAdapter {

  @Override
  public void addInterceptors(List<EndpointInterceptor> interceptors) {
    interceptors.add(new MyInterceptor());
  }

  @Override
  public void addArgumentResolvers(List<MethodArgumentResolver> argumentResolvers)
{
    argumentResolvers.add(new MyArgumentResolver());
  }

}
```

If `WsConfigurer` does not expose some more advanced setting that needs to be configured, consider removing `@EnableWs` and extending directly from `WsConfigurationSupport` or `DelegatingWsConfiguration`.

```
@Configuration
public class EchoConfig extends WsConfigurationSupport {

  @Override
  public void addInterceptors(List<EndpointInterceptor> interceptors) {
    interceptors.add(new MyInterceptor());
  }

  @Bean
  @Override
  public PayloadRootAnnotationMethodEndpointMapping
payloadRootAnnotationMethodEndpointMapping() {
      // Create or delegate to "super" to create and
      // customize properties of PayloadRootAnnotationMethodEndpointMapping
  }
```

```
}
```

In the next couple of sections, a more elaborate description of the `@Endpoint` programming model is given.

NOTE	Endpoints, like any other Spring Bean, are scoped as a singleton by default. That is, one instance of the bean definition is created per container. Being a singleton implies that more than one thread can use it at the same time, so the endpoint has to be thread safe. If you want to use a different scope, such as prototype, see the Spring Framework reference documentation.

Note that all abstract base classes provided in Spring-WS are thread safe, unless otherwise indicated in the class-level Javadoc.

5.4. `@Endpoint` handling methods

For an endpoint to actually handle incoming XML messages, it needs to have one or more handling methods. Handling methods can take wide range of parameters and return types. However, they typically have one parameter that contains the message payload, and they return the payload of the response message (if any). This section covers which parameter and return types are supported.

To indicate what sort of messages a method can handle, the method is typically annotated with either the `@PayloadRoot` or the `@SoapAction` annotation. You can learn more about these annotations in Endpoint mappings.

The following example shows a handling method:

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

The `order` method takes an `Element` (annotated with `@RequestPayload`) as a parameter. This means that the payload of the message is passed on this method as a DOM element. The method has a `void` return type, indicating that no response message is sent.

5.4.1. Handling Method Parameters

The handling method typically has one or more parameters that refer to various parts of the incoming XML message. Most commonly, the handling method has a single parameter that maps to the payload of the message, but it can also map to other parts of the request message, such as a SOAP header. This section describes the parameters you can use in your handling method signatures.

To map a parameter to the payload of the request message, you need to annotate this parameter with the `@RequestPayload` annotation. This annotation tells Spring-WS that the parameter needs to be bound to the request payload.

The following table describes the supported parameter types. It shows the supported types, whether the parameter should be annotated with `@RequestPayload`, and any additional notes.

Name	Supported parameter types	`@RequestPayload` required?	Additional notes
TrAX	`javax.xml.transform.Source` and sub-interfaces (`DOMSource`, `SAXSource`, `StreamSource`, and `StAXSource`)	Yes	Enabled by default.
W3C DOM	`org.w3c.dom.Element`	Yes	Enabled by default
dom4j	`org.dom4j.Element`	Yes	Enabled when dom4j is on the classpath.
JDOM	`org.jdom.Element`	Yes	Enabled when JDOM is on the classpath.
XOM	`nu.xom.Element`	Yes	Enabled when XOM is on the classpath.
StAX	`javax.xml.stream.XMLStreamReader` and `javax.xml.stream.XMLEventReader`	Yes	Enabled when StAX is on the classpath.
XPath	Any boolean, double, `String`, `org.w3c.Node`, `org.w3c.dom.NodeList`, or type that can be converted from a `String` by a Spring conversion service, and that is annotated with `@XPathParam`.	No	Enabled by default, see the section called `XPathParam`.
Message context	`org.springframework.ws.context.MessageContext`	No	Enabled by default.

Name	Supported parameter types	`@RequestPayload` required?	Additional notes
SOAP	`org.springframework.ws.soap.SoapMessage`, `org.springframework.ws.soap.SoapBody`, `org.springframework.ws.soap.SoapEnvelope`, `org.springframework.ws.soap.SoapHeader`, and `org.springframework.ws.soap.SoapHeaderElement`s when used in combination with the `@SoapHeader` annotation.	No	Enabled by default.
JAXB2	Any type that is annotated with `javax.xml.bind.annotation.XmlRootElement`, and `javax.xml.bind.JAXBElement`.	Yes	Enabled when JAXB2 is on the classpath.
OXM	Any type supported by a Spring OXM `Unmarshaller`.	Yes	Enabled when the `unmarshaller` attribute of `<sws:annotation-driven/>` is specified.

The next few examples show possible method signatures. The following method is invoked with the payload of the request message as a DOM `org.w3c.dom.Element`:

```
public void handle(@RequestPayload Element element)
```

The following method is invoked with the payload of the request message as a `javax.xml.transform.dom.DOMSource`. The `header` parameter is bound to the SOAP header of the request message.

```
public void handle(@RequestPayload DOMSource domSource, SoapHeader header)
```

The following method is invoked with the payload of the request message unmarshalled into a `MyJaxb2Object` (which is annotated with `@XmlRootElement`). The payload of the message is also given as a DOM `Element`. The whole message context is passed on as the third parameter.

```
public void handle(@RequestPayload MyJaxb2Object requestObject, @RequestPayload
Element element, Message messageContext)
```

As you can see, there are a lot of possibilities when it comes to defining how to handle method signatures. You can even extend this mechanism to support your own parameter types. See the Javadoc of `DefaultMethodEndpointAdapter` and `MethodArgumentResolver` to see how.

`@XPathParam`

One parameter type needs some extra explanation: `@XPathParam`. The idea here is that you annotate one or more method parameters with an XPath expression and that each such annotated parameter is bound to the evaluation of the expression. The following example shows how to do so:

```
package samples;

import javax.xml.transform.Source;

import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.Namespace;
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")
  @Namespace(prefix = "s", uri="http://samples")
  public Order getOrder(@XPathParam("/s:orderRequest/@id") int orderId) {
    Order order = orderService.getOrder(orderId);
    // create Source from order and return it
  }

}
```

Since we use the `s` prefix in our XPath expression, we must bind it to the `http://samples` namespace. This is accomplished with the `@Namespace` annotation. Alternatively, we could have placed this annotation on the type-level to use the same namespace mapping for all handler methods or even the package-level (in `package-info.java`) to use it for multiple endpoints.

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

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

In addition to this list, you can use any type that can be converted from a `String` by a Spring conversion service.

5.4.2. Handling method return types

To send a response message, the handling needs to specify a return type. If no response message is required, the method can declare a `void` return type. Most commonly, the return type is used to create the payload of the response message. However, you can also map to other parts of the response message. This section describes the return types you can use in your handling method signatures.

To map the return value to the payload of the response message, you need to annotate the method with the `@ResponsePayload` annotation. This annotation tells Spring-WS that the return value needs to be bound to the response payload.

The following table describes the supported return types. It shows the supported types, whether the parameter should be annotated with `@ResponsePayload`, and any additional notes.

Name	**Supported return types**	`@ResponsePayload` **required?**	**Additional notes**
No response	`void`	No	Enabled by default.
TrAX	`javax.xml.transform.Source` and sub-interfaces (`DOMSource`, `SAXSource`, `StreamSource`, and `StAXSource`)	Yes	Enabled by default.
W3C DOM	`org.w3c.dom.Element`	Yes	Enabled by default
dom4j	`org.dom4j.Element`	Yes	Enabled when dom4j is on the classpath.
JDOM	`org.jdom.Element`	Yes	Enabled when JDOM is on the classpath.
XOM	`nu.xom.Element`	Yes	Enabled when XOM is on the classpath.

Name	Supported return types	`@ResponsePayload` required?	Additional notes
JAXB2	Any type that is annotated with `javax.xml.bind.annotation.XmlRootElement`, and `javax.xml.bind.JAXBElement`.	Yes	Enabled when JAXB2 is on the classpath.
OXM	Any type supported by a Spring OXM `Marshaller`.	Yes	Enabled when the `marshaller` attribute of `<sws:annotation-driven/>` is specified.

There are a lot of possibilities when it comes to defining handling method signatures. It is even possible to extend this mechanism to support your own parameter types. See the class-level Javadoc of `DefaultMethodEndpointAdapter` and `MethodReturnValueHandler` to see how.

5.5. Endpoint mappings

The endpoint mapping is responsible for mapping incoming messages to appropriate endpoints. Some endpoint mappings are enabled by default — for example, the `PayloadRootAnnotationMethodEndpointMapping` or the `SoapActionAnnotationMethodEndpointMapping`. However, we first need to 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 are applied to the request and response. When a request comes in, the `MessageDispatcher` hands it over to the endpoint mapping to let it inspect the request and come up with an appropriate `EndpointInvocationChain`. Then the `MessageDispatcher` invokes the endpoint and any interceptors in the chain.

The concept of configurable endpoint mappings that can optionally contain interceptors (which can, in turn, manipulate the request, the response, or both) is extremely powerful. A lot of supporting functionality can be built into custom `EndpointMapping` implementations. For example, a custom endpoint mapping could choose an endpoint based not only 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 Intercepting Requests — the `EndpointInterceptor` Interface.

As explained in Endpoints, the `@Endpoint` style lets you handle multiple requests in one endpoint class. This is the responsibility of the `MethodEndpointMapping`. 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` You can enable both methods by using `<sws:annotation-driven/>` in your application context.

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 with this qualified name for the payload root element, the method is invoked.

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

`AbstractEndpointMapping` implementations provides a `defaultEndpoint` property that configures the endpoint to use when a configured mapping does not result in a matching endpoint.

5.5.1. WS-Addressing

WS-Addressing specifies a transport-neutral routing mechanism. It is based on the `To` and `Action` SOAP headers, which indicate the destination and intent of the SOAP message, respectively. Additionally, WS-Addressing lets you define a return address (for normal messages and for faults) and a unique message identifier, which can be used for correlation. For more information on WS-Addressing, see https://en.wikipedia.org/wiki/WS-Addressing. The following example shows a WS-Addressing message:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <SOAP-ENV:Header>
    <wsa:MessageID>urn:uuid:21363e0d-2645-4eb7-8afd-2f5ee1bb25cf</wsa:MessageID>
    <wsa:ReplyTo>
      <wsa:Address>http://example.com/business/client1</wsa:Address>
    </wsa:ReplyTo>
    <wsa:To S:mustUnderstand="true">http://example/com/fabrikam</wsa:To>
    <wsa:Action>http://example.com/fabrikam/mail/Delete</wsa:Action>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <f:Delete xmlns:f="http://example.com/fabrikam">
      <f:maxCount>42</f:maxCount>
    </f:Delete>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the preceding example, the destination is set to `http://example/com/fabrikam`, while the action is set to `http://example.com/fabrikam/mail/Delete`. Additionally, there is a message identifier and a reply-to address. By default, this address is the "anonymous" address, indicating that a response should be sent by using the same channel as the request (that is, the HTTP response), but it can also be another address, as indicated in this example.

In Spring-WS, WS-Addressing is implemented as an endpoint mapping. By using this mapping, you associate WS-Addressing actions with endpoints, similar to the `SoapActionAnnotationMethodEndpointMapping` described earlier.

Using `AnnotationActionEndpointMapping`

The `AnnotationActionEndpointMapping` is similar to the `SoapActionAnnotationMethodEndpointMapping` but uses WS-Addressing headers instead of the SOAP Action transport header.

To use the `AnnotationActionEndpointMapping`, annotate the handling methods with the `@Action` annotation, similar to the `@PayloadRoot` and `@SoapAction` annotations described in `@Endpoint` handling methods and Endpoint mappings. The following example shows how to do so:

```
package samples;

import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.soap.addressing.server.annotation.Action

@Endpoint
public class AnnotationOrderEndpoint {

    private final OrderService orderService;

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

    @Action("http://samples/RequestOrder")
    public Order getOrder(OrderRequest orderRequest) {
        return orderService.getOrder(orderRequest.getId());
    }

    @Action("http://samples/CreateOrder")
    public void order(Order order) {
        orderService.createOrder(order);
    }

}
```

The preceding mapping routes requests that have a WS-Addressing `Action` of `http://samples/RequestOrder` to the `getOrder` method. Requests with `http://samples/CreateOrder` are routed to the `order` method.

By default, the `AnnotationActionEndpointMapping` supports both the 1.0 (May 2006), and the August 2004 editions of WS-Addressing. These two versions are most popular and are interoperable with Axis 1 and 2, JAX-WS, XFire, Windows Communication Foundation (WCF), and Windows Services Enhancements (WSE) 3.0. If necessary, specific versions of the spec can be injected into the `versions` property.

In addition to the `@Action` annotation, you can annotate the class with the `@Address` annotation. If set, the value is compared to the `To` header property of the incoming message.

Finally, there is the `messageSenders` property, which is required for sending response messages to non-anonymous, out-of-bound addresses. You can set `MessageSender` implementations in this property, the same as you would on the `WebServiceTemplate`. See URIs and Transports.

5.5.2. 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.

Endpoint interceptors are typically defined by using a `<sws:interceptors>` element in your application context. In this element, you can define endpoint interceptor beans that apply to all endpoints defined in that application context. Alternatively, you can use `<sws:payloadRoot>` or `<sws:soapAction>` elements to specify for which payload root name or SOAP action the interceptor should apply. The following example shows how to do so:

```
<sws:interceptors>
  <bean class="samples.MyGlobalInterceptor"/>
  <sws:payloadRoot namespaceUri="http://www.example.com">
    <bean class="samples.MyPayloadRootInterceptor"/>
  </sws:payloadRoot>
  <sws:soapAction value="http://www.example.com/SoapAction">
    <bean class="samples.MySoapActionInterceptor1"/>
    <ref bean="mySoapActionInterceptor2"/>
  </sws:soapAction>
</sws:interceptors>

<bean id="mySoapActionInterceptor2" class="samples.MySoapActionInterceptor2"/>
```

In the preceding example, we define one "global" interceptor (`MyGlobalInterceptor`) that intercepts all requests and responses. We also define an interceptor that applies only to XML messages that have the `http://www.example.com` as a payload root namespace. We could have defined a `localPart` attribute in addition to the `namespaceUri` to further limit the messages to which the interceptor applies. Finally, we define two interceptors that apply when the message has a `http://www.example.com/SoapAction` SOAP action. Notice how the second interceptor is actually a reference to a bean definition outside the `<interceptors>` element. You can use bean references anywhere inside the `<interceptors>` element.

When you use `@Configuration` classes, you can extend from `WsConfigurerAdapter` to add interceptors:

```
@Configuration
@EnableWs
public class MyWsConfiguration extends WsConfigurerAdapter {

  @Override
  public void addInterceptors(List<EndpointInterceptor> interceptors) {
```

```
    interceptors.add(new MyPayloadRootInterceptor());
  }

}
```

Interceptors must implement `EndpointInterceptor`. This interface defines three methods, one that can be used for handling the request message **before** the actual endpoint is processed, one that can be used for handling a normal response message, and one that can be used for handling fault messages. The second two are called **after** the endpoint is processed. 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 processing 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 processing 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 that you can use in your Web service. Additionally, there is the `XwsSecurityInterceptor`, which is described in `XwsSecurityInterceptor`.

`PayloadLoggingInterceptor` and `SoapEnvelopeLoggingInterceptor`

When developing a web service, it can be useful to log the incoming and outgoing XML messages. Spring WS facilitates this with the `PayloadLoggingInterceptor` and `SoapEnvelopeLoggingInterceptor` classes. The former logs only the payload of the message. The latter logs the entire SOAP envelope, including SOAP headers. The following example shows how to define the `PayloadLoggingInterceptor` in an endpoint mapping:

```
  <sws:interceptors>
    <bean
class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingIntercepto
r"/>
  </sws:interceptors>
```

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.

You could use the `WsConfigurerAdapter` approach, as described earlier, for the `PayloadLoggingInterceptor` as well.

`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, responses, or both.

NOTE	While request validation may sound like a good idea, it 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 fulfill 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."

The following example 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 by setting the `schemas` property.

```
<bean id="validatingInterceptor"

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

Of course, you could use the `WsConfigurerAdapter` approach, as described earlier, for the `PayloadValidatingInterceptor` as well.

Using `PayloadTransformingInterceptor`

To transform the payload to another XML format, Spring-WS offers the `PayloadTransformingInterceptor`. This endpoint interceptor is based on XSLT style sheets and is especially useful when supporting multiple versions of a web service, because you can transform the older message format to the newer format. The following example uses the `PayloadTransformingInterceptor`:

```
<bean id="transformingInterceptor"

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

In the preceding example, we transform requests by using `/WEB-INF/oldRequests.xslt` and response

messages by using `/WEB-INF/oldResponses.xslt`. Note that, since endpoint interceptors are registered at the endpoint-mapping level, you can create an endpoint mapping that applies to the "old style" messages and add the interceptor to that mapping. Hence, the transformation applies only to these "old style" message.

You could use the `WsConfigurerAdapter` approach, as described earlier, for the `PayloadTransformingInterceptor` as well.

5.6. Handling Exceptions

Spring-WS provides `EndpointExceptionResolver` implementations to ease the pain of unexpected exceptions occurring while your message is being processed by an endpoint that 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 creates a SOAP 1.1 Server or SOAP 1.2 Receiver fault and uses the exception message as the fault string. You can subclass it to customize the fault, as shown in the following example:

```
public class CustomSoapExceptionResolver extends SimpleSoapExceptionResolver {

    private final Transformer transformer;

    public CustomSoapExceptionResolver(Transformer transformer) {
        this.transformer = transformer;
    }

    @Override
    protected void customizeFault(MessageContext messageContext, Object endpoint,
Exception exception,
            SoapFault fault) {
        SoapFaultDetail faultDetail = fault.addFaultDetail();
        try {
            this.transformer.transform(new StringSource("""
                    <ns2:YourCustomException xmlns:ns2="http://serviceendpoint/">
                        <errorCode>Your custom error code</errorCode>
                        <systemMessage>A system message</systemMessage>
```

```
                    </ns2:YourCustomException >
                   """), faultDetail.getResult());
        }
        catch (TransformerException ex) {
            throw new IllegalArgumentException("Failed to write detail", ex);
        }

    }
}
```

The `SimpleSoapExceptionResolver` is the default, but it can be overridden by explicitly adding another resolver.

5.6.1. SoapFaultMappingExceptionResolver

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

```
<beans>
    <bean id="exceptionResolver"

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

The key values and default endpoint use a format of `faultCode,faultString,locale`, where only the fault code is required. If the fault string is not set, it defaults to the exception message. If the language is not set, it defaults to English. The preceding configuration maps exceptions of type `ValidationFailureException` to a client-side SOAP fault with a fault string of `Invalid request`, as follows:

```
<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 returns the default fault: a server-side fault with the exception message as the fault string.

5.6.2. Using `SoapFaultAnnotationExceptionResolver`

You can also annotate exception classes with the `@SoapFault` annotation, to indicate the SOAP fault that should be returned whenever that exception is thrown. 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. The following example shows such an 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 results 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>
```

5.7. Server-side Testing

When it comes to testing your Web service endpoints, you have two possible approaches:

- Write Unit Tests, where you provide (mock) arguments for your endpoint to consume. The

advantage of this approach is that it is quite easy to accomplish (especially for classes annotated with `@Endpoint`). The disadvantage is that you are not really testing the exact content of the XML messages that are sent over the wire.

- Write Integrations Tests, which do test the contents of the message.

The first approach can easily be accomplished with mocking frameworks such as Mockito, EasyMock, and others. The next section focuses on writing integration tests.

5.7.1. Writing server-side integration tests

Spring-WS has support for creating endpoint integration tests. In this context, an endpoint is a class that handles (SOAP) messages (see Endpoints).

The integration test support lives in the `org.springframework.ws.test.server` package. The core class in that package is the `MockWebServiceClient`. The underlying idea is that this client creates a request message and then sends it over to the endpoints that are configured in a standard `MessageDispatcherServlet` application context (see `MessageDispatcherServlet`). These endpoints handle the message and create a response. The client then receives this response and verifies it against registered expectations.

The typical usage of the `MockWebServiceClient` is: .

1. Create a `MockWebServiceClient` instance by calling `MockWebServiceClient.createClient(ApplicationContext)` or `MockWebServiceClient.createClient(WebServiceMessageReceiver, WebServiceMessageFactory)`.
2. Send request messages by calling `sendRequest(RequestCreator)`, possibly by using the default `RequestCreator` implementations provided in `RequestCreators` (which can be statically imported).
3. Set up response expectations by calling `andExpect(ResponseMatcher)`, possibly by using the default `ResponseMatcher` implementations provided in `ResponseMatchers` (which can be statically imported). Multiple expectations can be set up by chaining `andExpect(ResponseMatcher)` calls.

NOTE	`MockWebServiceClient` (and related classes) offers a "fluent" API, so you can typically use the code-completion features in your IDE to guide you through the process of setting up the mock server.

NOTE	You can rely on the standard logging features available in Spring-WS in your unit tests. Sometimes, it might be useful to inspect the request or response message to find out why a particular tests failed. See Message Logging and Tracing for more information.

Consider, for example, the following web service endpoint class:

```
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;
```

```
@Endpoint                                                          ①
public class CustomerEndpoint {

  @ResponsePayload                                                 ②
  public CustomerCountResponse getCustomerCount(
      @RequestPayload CustomerCountRequest request) {
    CustomerCountResponse response = new CustomerCountResponse();
    response.setCustomerCount(10);
    return response;
  }

}
```

① The `CustomerEndpoint` in annotated with `@Endpoint`. See Endpoints.

② The `getCustomerCount()` method takes a `CustomerCountRequest` as its argument and returns a `CustomerCountResponse`. Both of these classes are objects supported by a marshaller. For instance, they can have a `@XmlRootElement` annotation to be supported by JAXB2.

The following example shows a typical test for `CustomerEndpoint`:

```
import javax.xml.transform.Source;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.xml.transform.StringSource;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.ws.test.server.MockWebServiceClient;
import static org.springframework.ws.test.server.RequestCreators.*;
import static org.springframework.ws.test.server.ResponseMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
①
@ContextConfiguration("spring-ws-servlet.xml")
public class CustomerEndpointIntegrationTest {

  @Autowired
  private ApplicationContext applicationContext;
②

  private MockWebServiceClient mockClient;

  @Before
  public void createClient() {
```

```
    mockClient = MockWebServiceClient.createClient(applicationContext);
③
  }

  @Test
  public void customerEndpoint() throws Exception {
    Source requestPayload = new StringSource("""
      <customerCountRequest xmlns='http://springframework.org/spring-ws'>
        <customerName>John Doe</customerName>
      </customerCountRequest>
    """);
    Source responsePayload = new StringSource("""
      <customerCountResponse xmlns='http://springframework.org/spring-ws'>
        <customerCount>10</customerCount>
      </customerCountResponse>
    """);

    mockClient.sendRequest(withPayload(requestPayload)).
④
      andExpect(payload(responsePayload));
  }
}
```

① This test uses the standard testing facilities provided in the Spring Framework. This is not required but is generally the easiest way to set up the test.

② The application context is a standard Spring-WS application context (see `MessageDispatcherServlet`), read from `spring-ws-servlet.xml`. In this case, the application context contains a bean definition for `CustomerEndpoint` (or perhaps a `<context:component-scan />` is used).

③ In a `@Before` method, we create a `MockWebServiceClient` by using the `createClient` factory method.

④ We send a request by calling `sendRequest()` with a `withPayload()` `RequestCreator` provided by the statically imported `RequestCreators` (see Using `RequestCreator` and `RequestCreators`). We also set up response expectations by calling `andExpect()` with a `payload()` `ResponseMatcher` provided by the statically imported `ResponseMatchers` (see Using `ResponseMatcher` and `ResponseMatchers`).

This part of the test might look a bit confusing, but the code completion features of your IDE are of great help. After typing `sendRequest(`, your IDE can provide you with a list of possible request creating strategies, provided you statically imported `RequestCreators`. The same applies to `andExpect()`, provided you statically imported `ResponseMatchers`.

5.7.2. Using `RequestCreator` and `RequestCreators`

Initially, the `MockWebServiceClient` needs to create a request message for the endpoint to consume. The client uses the `RequestCreator` strategy interface for this purpose:

```
public interface RequestCreator {

  WebServiceMessage createRequest(WebServiceMessageFactory messageFactory)
    throws IOException;

}
```

You can write your own implementations of this interface, creating a request message by using the message factory, but you certainly do not have to. The `RequestCreators` class provides a way to create a `RequestCreator` based on a given payload in the `withPayload()` method. You typically statically import `RequestCreators`.

5.7.3. Using `ResponseMatcher` and `ResponseMatchers`

When the request message has been processed by the endpoint and a response has been received, the `MockWebServiceClient` can verify whether this response message meets certain expectations. The client uses the `ResponseMatcher` strategy interface for this purpose:

```
public interface ResponseMatcher {

    void match(WebServiceMessage request, WebServiceMessage response)
      throws IOException, AssertionError;

}
```

Once again, you can write your own implementations of this interface, throwing `AssertionError` instances when the message does not meet your expectations, but you certainly do not have to, as the `ResponseMatchers` class provides standard `ResponseMatcher` implementations for you to use in your tests. You typically statically import this class.

The `ResponseMatchers` class provides the following response matchers:

`ResponseMatchers` **method**	**Description**
`payload()`	Expects a given response payload. May include XMLUnit Placeholders.
`validPayload()`	Expects the response payload to validate against given XSD schemas.
`xpath()`	Expects a given XPath expression to exist, not exist, or evaluate to a given value.
`soapHeader()`	Expects a given SOAP header to exist in the response message.

`ResponseMatchers` **method**	**Description**
`soapEnvelope()`	Expects a given SOAP payload. May include XMLUnit Placeholders.
`noFault()`	Expects that the response message does not contain a SOAP Fault.
`mustUnderstandFault()`, `clientOrSenderFault()`, `serverOrReceiverFault()`, and `versionMismatchFault()`	Expects the response message to contain a specific SOAP Fault.

You can set up multiple response expectations by chaining `andExpect()` calls:

```
mockClient.sendRequest(...).
 andExpect(payload(expectedResponsePayload)).
 andExpect(validPayload(schemaResource));
```

For more information on the response matchers provided by `ResponseMatchers`, see the Javadoc.

Chapter 6. Using Spring-WS on the Client

Spring-WS provides a client-side Web service API that allows for consistent, XML-driven access to web services. It also caters to 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

- Sending and receiving of XML messages.
- Marshalling objects to XML before sending.
- Allowing for multiple transport options.

6.1. Using the Client-side API

This section describes how to use the client-side API. For how to use the server-side API, see Creating a Web service with Spring-WS.

6.1.1. `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.

URIs and Transports

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

HTTP transports

There are three implementations of the `WebServiceMessageSender` interface for sending messages over HTTP. The default implementation is the `HttpUrlConnectionMessageSender`, which uses the facilities provided by Java itself. The alternatives are either `JdkHttpClientMessageSender` that uses the JDK's `HttpClient`, or `HttpComponents5MessageSender`, which uses the Apache HttpClient. Use the latter if you need more advanced and easy-to-use functionality (such as authentication, HTTP connection pooling, and so forth).

To use the HTTP transport, either set the `defaultUri` to something like `http://example.com/services` or supply the `uri` parameter for one of the methods.

The following example shows how to use default configuration for HTTP transports:

```
<beans>

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

    <bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory"/>
        <property name="defaultUri" value="http://example.com/WebService"/>
    </bean>

</beans>
```

The following example shows how to override the default configuration and how to use Apache HttpClient to authenticate with HTTP authentication:

```
<bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
    <constructor-arg ref="messageFactory"/>
    <property name="messageSender">
        <bean
class="org.springframework.ws.transport.http.HttpComponents5MessageSender">
            <property name="credentials">
                <bean
class="org.apache.hc.client5.http.auth.UsernamePasswordCredentials">
                    <constructor-arg value="john"/>
                    <constructor-arg value="secret"/>
                </bean>
            </property>
        </bean>
    </property>
    <property name="defaultUri" value="http://example.com/WebService"/>
</bean>
```

JMS transport

For sending messages over JMS, Spring-WS provides `JmsMessageSender`. This class uses the facilities of the Spring framework to transform the `WebServiceMessage` into a JMS `Message`, send it on its way on a `Queue` or `Topic`, and receive a response (if any).

To use `JmsMessageSender`, you need to set the `defaultUri` or `uri` parameter to a JMS URI, which — at a minimum — consists of the `jms:` prefix and a destination name. Some examples of JMS URIs are: `jms:SomeQueue`, `jms:SomeTopic?priority=3&deliveryMode=NON_PERSISTENT`, and

`jms:RequestQueue?replyToName=ResponseName`. For more information on this URI syntax, see the Javadoc for `JmsMessageSender`.

By default, the `JmsMessageSender` sends JMS `BytesMessage`, but you can override this to use `TextMessages` by using the `messageType` parameter on the JMS URI — for example, `jms:Queue?messageType=TEXT_MESSAGE`. Note that `BytesMessages` are the preferred type, because `TextMessages` do not support attachments and character encodings reliably.

The following example shows how to use the JMS transport in combination with an Artemis connection factory:

```
<beans>

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

    <bean id="connectionFactory"
class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
        <property name="brokerURL"
value="vm://localhost?broker.persistent=false"/>
    </bean>

    <bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory"/>
        <property name="messageSender">
            <bean class="org.springframework.ws.transport.jms.JmsMessageSender">
                <property name="connectionFactory" ref="connectionFactory"/>
            </bean>
        </property>
        <property name="defaultUri"
value="jms:RequestQueue?deliveryMode=NON_PERSISTENT"/>
    </bean>

</beans>
```

Email Transport

Spring-WS also provides an email transport, which you can use to send web service messages over SMTP and retrieve them over either POP3 or IMAP. The client-side email functionality is contained in `MailMessageSender`. This class creates an email message from the request `WebServiceMessage` and sends it over SMTP. It then waits for a response message to arrive at the incoming POP3 or IMAP server.

To use the `MailMessageSender`, set the `defaultUri` or `uri` parameter to a `mailto` URI — for example, `mailto:john@example.com` or `mailto:server@localhost?subject=SOAP%20Test`. Make sure that the message sender is properly configured with a `transportUri`, which indicates the server to use for sending requests (typically a SMTP server), and a `storeUri`, which indicates the server to poll for

responses (typically a POP3 or IMAP server).

The following example shows how to use the email transport:

```
<beans>

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

    <bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory"/>
        <property name="messageSender">
            <bean class="org.springframework.ws.transport.mail.MailMessageSender">
                <property name="from" value="Spring-WS SOAP Client
&lt;client@example.com&gt;"/>
                <property name="transportUri"
value="smtp://client:s04p@smtp.example.com"/>
                <property name="storeUri"
value="imap://client:s04p@imap.example.com/INBOX"/>
            </bean>
        </property>
        <property name="defaultUri"
value="mailto:server@example.com?subject=SOAP%20Test"/>
    </bean>

</beans>
```

XMPP Transport

Spring-WS also provides a XMPP (Jabber) transport, which you can use to send and receive web service messages over XMPP. The client-side XMPP functionality is contained in `XmppMessageSender`. This class creates an XMPP message from the request `WebServiceMessage` and sends it over XMPP. It then listens for a response message to arrive.

To use the `XmppMessageSender`, set the `defaultUri` or `uri` parameter to a `xmpp` URI — for example, `xmpp:johndoe@jabber.org`. The sender also requires an `XMPPConnection` to work, which can be conveniently created by using the `org.springframework.ws.transport.xmpp.support.XmppConnectionFactoryBean`.

The following example shows how to use the XMPP transport:

```
<beans>

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

```
    <bean id="connection"
class="org.springframework.ws.transport.xmpp.support.XmppConnectionFactoryBean">
        <property name="host" value="jabber.org"/>
        <property name="username" value="username"/>
        <property name="password" value="password"/>
    </bean>

    <bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory"/>
        <property name="messageSender">
            <bean class="org.springframework.ws.transport.xmpp.XmppMessageSender">
                <property name="connection" ref="connection"/>
            </bean>
        </property>
        <property name="defaultUri" value="xmpp:user@jabber.org"/>
    </bean>

</beans>
```

Message factories

In addition to a message sender, the `WebServiceTemplate` requires a web service message factory. By default, `SaajSoapMessageFactory` is used.

6.1.2. 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 that marshal and unmarshal objects to XML. The following example 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</message>";

    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/AnotherWebS
ervice",
            source, result);
    }

}
```

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

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

</beans>
```

The preceding 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.

Note that the `WebServiceTemplate` class is thread-safe once configured (assuming that all of its dependencies are also thread-safe, which is the case for all of the dependencies that ship with Spring-WS), so multiple objects can use the same shared `WebServiceTemplate` instance. The `WebServiceTemplate` exposes a zero-argument constructor and `messageFactory` and `messageSender` bean properties that you can use to construct the instance (by using a Spring container or plain Java code). Alternatively, consider deriving from Spring-WS `WebServiceGatewaySupport` convenience base class, which exposes bean properties to enable easy configuration. (You do not have to extend this base class, it is provided as a convenience class only).

6.1.3. Sending and Receiving POJOs — Marshalling and Unmarshalling

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, see the Spring Framework reference documentation).

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. To use the marshalling functionality, you have to set a marshaller and an unmarshaller with the `marshaller` and `unmarshaller` properties of the `WebServiceTemplate` class.

6.1.4. Using `WebServiceMessageCallback`

To accommodate setting 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 following example 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 doWithMessage(WebServiceMessage message) {
            ((SoapMessage)message).setSoapAction("http://tempuri.org/Action");
        }
    });
}
```

NOTE	Note that you can also use the `org.springframework.ws.soap.client.core.SoapActionCallback` to set the SOAP action header.

WS-Addressing

In addition to the server-side WS-Addressing support, Spring-WS also has support for this specification on the client-side.

For setting WS-Addressing headers on the client, you can use `ActionCallback`. This callback takes the desired action header as a parameter. It also has constructors for specifying the WS-Addressing version and a `To` header. If not specified, the `To` header defaults to the URL of the connection being made.

The following example sets the `Action` header to `http://samples/RequestOrder`:

```
webServiceTemplate.marshalSendAndReceive(o, new
ActionCallback("http://samples/RequestOrder"));
```

6.1.5. Using `WebServiceMessageExtractor`

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

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

6.2. Client-side Testing

When it comes to testing your Web service clients (that is, classes that use the `WebServiceTemplate` to access a Web service), you have two possible approaches:

- Write unit tests, which mock away the `WebServiceTemplate` class, `WebServiceOperations` interface, or the complete client class. The advantage of this approach is that it s easy to accomplish. The disadvantage is that you are not really testing the exact content of the XML messages that are sent over the wire, especially when mocking out the entire client class.
- Write integrations tests, which do test the contents of the message.

The first approach can easily be accomplished with mocking frameworks, such as Mockito, EasyMock, and others. The next section focuses on writing integration tests.

6.2.1. Writing Client-side Integration Tests

Spring-WS has support for creating for creating Web service client integration tests. In this context, a client is a class that uses the `WebServiceTemplate` to access a web service.

The integration test support lives in the `org.springframework.ws.test.client` package. The core class in that package is the `MockWebServiceServer`. The underlying idea is that the web service template connects to this mock server and sends it a request message, which the mock server then verifies against the registered expectations. If the expectations are met, the mock server then prepares a response message, which is sent back to the template.

The typical usage of the `MockWebServiceServer` is: .

1. Create a `MockWebServiceServer` instance by calling `MockWebServiceServer.createServer(WebServiceTemplate)`, `MockWebServiceServer.createServer(WebServiceGatewaySupport)`, or `MockWebServiceServer.createServer(ApplicationContext)`.
2. Set up request expectations by calling `expect(RequestMatcher)`, possibly by using the default `RequestMatcher` implementations provided in `RequestMatchers` (which can be statically imported). Multiple expectations can be set up by chaining `andExpect(RequestMatcher)` calls.
3. Create an appropriate response message by calling `andRespond(ResponseCreator)`, possibly by using the default `ResponseCreator` implementations provided in `ResponseCreators` (which can be statically imported).
4. Use the `WebServiceTemplate` as normal, either directly of through client code.
5. Call `MockWebServiceServer.verify()` to make sure that all expectations have been met.

NOTE	`MockWebServiceServer` (and related classes) offers a 'fluent' API, so you can typically use the code-completion features in your IDE to guide you through the process of setting up the mock server.

NOTE	You can rely on the standard logging features available in Spring-WS in your unit tests. Sometimes, it might be useful to inspect the request or response message to find out why a particular tests failed. See Message Logging and Tracing for more information.

Consider, for example, the following Web service client class:

```
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;

public class CustomerClient extends WebServiceGatewaySupport {
①

  public int getCustomerCount() {
    CustomerCountRequest request = new CustomerCountRequest();
②
    request.setCustomerName("John Doe");
```

```
    CustomerCountResponse response =
      (CustomerCountResponse)
getWebServiceTemplate().marshalSendAndReceive(request);    ③
    return response.getCustomerCount();
  }

}
```

① The `CustomerClient` extends `WebServiceGatewaySupport`, which provides it with a `webServiceTemplate` property.

② `CustomerCountRequest` is an object supported by a marshaller. For instance, it can have an `@XmlRootElement` annotation to be supported by JAXB2.

③ The `CustomerClient` uses the `WebServiceTemplate` offered by `WebServiceGatewaySupport` to marshal the request object into a SOAP message and sends that to the web service. The response object is unmarshalled into a `CustomerCountResponse`.

The following example shows a typical test for `CustomerClient`:

```
import javax.xml.transform.Source;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.xml.transform.StringSource;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertEquals;

import org.springframework.ws.test.client.MockWebServiceServer;
import static org.springframework.ws.test.client.RequestMatchers.*;
import static org.springframework.ws.test.client.ResponseCreators.*;

@RunWith(SpringJUnit4ClassRunner.class)
①
@ContextConfiguration("integration-test.xml")
public class CustomerClientIntegrationTest {

  @Autowired
  private CustomerClient client;
②

  private MockWebServiceServer mockServer;
③

  @Before
```

```
  public void createServer() throws Exception {
    mockServer = MockWebServiceServer.createServer(client);
  }

  @Test
  public void customerClient() throws Exception {
    Source requestPayload = new StringSource("""
      <customerCountRequest xmlns='http://springframework.org/spring-ws'>
        <customerName>John Doe</customerName>
      </customerCountRequest>
    """);
    Source responsePayload = new StringSource("""
      <customerCountResponse xmlns='http://springframework.org/spring-ws'>
        <customerCount>10</customerCount>
      </customerCountResponse>
    """);


mockServer.expect(payload(requestPayload)).andRespond(withPayload(responsePayload)
);④

    int result = client.getCustomerCount();
⑤
    assertEquals(10, result);

    mockServer.verify();
⑥
  }

}
```

① This test uses the standard testing facilities provided in the Spring Framework. This is not required but is generally the easiest way to set up the test.

② The `CustomerClient` is configured in `integration-test.xml` and wired into this test using `@Autowired`.

③ In a `@Before` method, we create a `MockWebServiceServer` by using the `createServer` factory method.

④ We define expectations by calling `expect()` with a `payload()` `RequestMatcher` provided by the statically imported `RequestMatchers` (see Using `RequestMatcher` and `RequestMatchers`). We also set up a response by calling `andRespond()` with a `withPayload()` `ResponseCreator` provided by the statically imported `ResponseCreators` (see Using `ResponseCreator` and `ResponseCreators`). This part of the test might look a bit confusing, but the code-completion features of your IDE are of great help. After you type `expect(`, your IDE can provide you with a list of possible request matching strategies, provided you statically imported `RequestMatchers`. The same applies to `andRespond(`, provided you statically imported `ResponseCreators`.

⑤ We call `getCustomerCount()` on the `CustomerClient`, thus using the `WebServiceTemplate`. The template has been set up for “testing mode” by now, so no real (HTTP) connection is made by this method call. We also make some JUnit assertions based on the result of the method

call.

⑥ We call `verify()` on the `MockWebServiceServer`, verifying that the expected message was actually received.

6.2.2. Using `RequestMatcher` and `RequestMatchers`

To verify whether the request message meets certain expectations, the `MockWebServiceServer` uses the `RequestMatcher` strategy interface. The contract defined by this interface is as follows:

```
public interface RequestMatcher {

  void match(URI uri, WebServiceMessage request)
    throws IOException, AssertionError;

}
```

You can write your own implementations of this interface, throwing `AssertionError` exceptions when the message does not meet your expectations, but you certainly do not have to. The `RequestMatchers` class provides standard `RequestMatcher` implementations for you to use in your tests. You typically statically import this class.

The `RequestMatchers` class provides the following request matchers:

`RequestMatchers` **method**	**Description**
`anything()`	Expects any sort of request.
`payload()`	Expects a given request payload. May include XMLUnit Placeholders
`validPayload()`	Expects the request payload to validate against given XSD schemas.
`xpath()`	Expects a given XPath expression to exist, not exist, or evaluate to a given value.
`soapHeader()`	Expects a given SOAP header to exist in the request message.
`soapEnvelope()`	Expects a given SOAP payload. May include XMLUnit Placeholders
`connectionTo()`	Expects a connection to the given URL.

You can set up multiple request expectations by chaining `andExpect()` calls:

```
mockServer.expect(connectionTo("http://example.com")).
 andExpect(payload(expectedRequestPayload)).
```

```
  andExpect(validPayload(schemaResource)).
  andRespond(...);
```

For more information on the request matchers provided by `RequestMatchers`, see the Javadoc.

6.2.3. Using `ResponseCreator` and `ResponseCreators`

When the request message has been verified and meets the defined expectations, the `MockWebServiceServer` creates a response message for the `WebServiceTemplate` to consume. The server uses the `ResponseCreator` strategy interface for this purpose:

```
public interface ResponseCreator {

  WebServiceMessage createResponse(URI uri, WebServiceMessage request,
                                   WebServiceMessageFactory messageFactory)
    throws IOException;

}
```

Once again, you can write your own implementations of this interface, creating a response message by using the message factory, but you certainly do not have to, as the `ResponseCreators` class provides standard `ResponseCreator` implementations for you to use in your tests. You typically statically import this class.

The `ResponseCreators` class provides the following responses:

`ResponseCreators` **method**	**Description**
`withPayload()`	Creates a response message with a given payload.
`withError()`	Creates an error in the response connection. This method gives you the opportunity to test your error handling.
`withException()`	Throws an exception when reading from the response connection. This method gives you the opportunity to test your exception handling.
`withMustUnderstandFault()`, `withClientOrSenderFault()`, `withServerOrReceiverFault()`, or `withVersionMismatchFault()`	Creates a response message with a given SOAP fault. This method gives you the opportunity to test your Fault handling.

For more information on the request matchers provided by `RequestMatchers`, see the Javadoc.

Chapter 7. Securing Your Web services with Spring-WS

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

- **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 that 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 encrypted data back into a readable form.

These three areas are implemented by using the `XwsSecurityInterceptor` or `Wss4jSecurityInterceptor`, which we describe in `XwsSecurityInterceptor` and Using `Wss4jSecurityInterceptor`, respectively.

NOTE	WS-Security (especially encryption and signing) requires substantial amounts of memory and can decrease performance. If performance is important to you, you might want to consider not using WS-Security or using HTTP-based security.

7.1. `XwsSecurityInterceptor`

The `XwsSecurityInterceptor` is an `EndpointInterceptor` (see 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 Endpoint mappings). This means that you can be selective about adding WS-Security support. Some endpoint mappings require it, while others do not.

NOTE	XWSS requires a SUN SAAJ reference implementation. The WSS4J interceptor does not have these requirements (see Using `Wss4jSecurityInterceptor`).

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 is explained in the following sections, but a more in-depth tutorial is available. You can set the policy with the `policyConfiguration` property, which requires a Spring resource. The policy file can contain multiple elements — for example, require a username token on incoming messages and sign all outgoing messages. It contains a

`SecurityConfiguration` element (not a `JAXRPCSecurity` element) as its root.

Additionally, the security interceptor requires one or more `CallbackHandler` instances to operate. These handlers are used to retrieve certificates, private keys, validate user credentials, and so on. Spring-WS offers handlers for most common security concerns — for example, authenticating against a Spring Security authentication manager and signing outgoing messages based on a X509 certificate. The following sections indicate what callback handler to use for which security concern. You can set the callback handlers by using the `callbackHandler` or `callbackHandlers` property.

The following example that shows how to wire up the `XwsSecurityInterceptor`:

```
<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 by using the `securityPolicy.xml` file on the classpath. It uses two callback handlers that are defined later in the file.

7.1.1. Keystores

For most cryptographic operations, you an use the standard `java.security.KeyStore` objects. These operations include certificate verification, message signing, signature verification, and encryption. They exclude 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 a certificate chain for the corresponding public key. Within the field of WS-Security, this accounts for message signing and message decryption.
- **Symmetric Keys**: Symmetric (or secret) keys are also used for message encryption and decryption — 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 does indeed belong to the owner of the certificate. Within WS-Security, these certificates are used for certificate validation, signature verification, and encryption.

Using `keytool`

The `keytool` program, a key and certificate management utility, is supplied with your Java Virtual Machine. You can use this tool to create new keystores, add new private keys and certificates to them, and so on. It is beyond the scope of this document to provide a full reference of the `keytool` command, check the standard reference or invoke `keytool -help` on the command line.

Using `KeyStoreFactoryBean`

To easily load a keystore by 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. The following listing configures a `KeyStoreFactoryBean`:

```
<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>
```

WARNING If you do not specify the location property, a new, empty keystore is created, which is most likely not what you want.

KeyStoreCallbackHandler

To use the keystores within a `XwsSecurityInterceptor`, you 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` is used. For determining trust relationships, the `trustStore` is used. The following table indicates this:

Cryptographic operation	Keystore used
Certificate validation	First `keyStore`, then `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`

Cryptographic operation	Keystore used
Signing	`keyStore`
Signature verification	`trustStore`

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

If the `symmetricStore` is not set, it defaults to the `keyStore`. If the key or trust store is not set, the callback handler uses the standard Java mechanism to load or create it. See `KeyStoreCallbackHandler` for more details.

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

```
<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 can use a key store:

```
<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 indicate where the `KeyStoreCallbackHandler` can be used and which properties to set for particular cryptographic operations.

7.1.2. Authentication

As stated in the introduction to this chapter, 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.

Plain Text Username Authentication

The simplest form of username authentication uses plain text passwords. In this scenario, the SOAP message contains 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	Plain text passwords are not very secure. Therefore, you should always add additional security measures to your transport layer if you use 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`. Fore more details, check the official reference of possible child elements. The following listing shows how to include a `RequireUsernameToken` element:

```
<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` returns a SOAP fault to the sender. If it is present, it fires a `PasswordValidationCallback` with a `PlainTextPasswordRequest` to the registered handlers. Within Spring-WS, there are three classes that handle this particular callback:

- `SimplePasswordValidationCallbackHandler`
- `SpringPlainTextPasswordValidationCallbackHandler`
- `JaasPlainTextPasswordValidationCallbackHandler`

Using `SimplePasswordValidationCallbackHandler`

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

```
<bean id="passwordValidationHandler"

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

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

Using `SpringPlainTextPasswordValidationCallbackHandler`

The `SpringPlainTextPasswordValidationCallbackHandler` uses Spring Security to authenticate users. It is beyond the scope of this document to describe Spring Security, but it is a full-fledged security framework. You can read more about it in the Spring Security reference documentation.

The `SpringPlainTextPasswordValidationCallbackHandler` requires an `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 by using the `authenticationManager` property:

```
<beans>
  <bean id="springSecurityHandler"

class="org.springframework.ws.soap.security.xwss.callback.SpringPlainTextPasswordV
alidationCallbackHandler">
    <property name="authenticationManager" ref="authenticationManager"/>
  </bean>

  <bean id="authenticationManager" class="
org.springframework.security.authentication.ProviderManager">
      <constructor-arg>
          <bean
class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
              <property name="userDetailsService" ref="userDetailsService"/>
          </bean>
      </constructor-arg>
  </bean>

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

Using `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 tutorials are available.

The `JaasPlainTextPasswordValidationCallbackHandler` requires only a `loginContextName` to operate. It creates a new JAAS `LoginContext` by using this name and handles the standard JAAS `NameCallback` and `PasswordCallback` by 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.JaasPlainTextPasswo
rdValidationCallbackHandler">
    <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 tutorial mentioned earlier.

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. This method is comparable 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` attribute should be set to `true`. For more details, check the official reference of possible child elements. The following listing shows how to define a `RequireUsernameToken` element:

```
<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` returns a SOAP fault to the sender. If it is present, it fires a `PasswordValidationCallback` with a `DigestPasswordRequest` to the registered handlers. Within Spring-WS, two classes handle this particular callback: `SimplePasswordValidationCallbackHandler` and `SpringDigestPasswordValidationCallbackHandler`.

Using `SimplePasswordValidationCallbackHandler`

The `SimplePasswordValidationCallbackHandler` can handle both plain text passwords as well as password digests. It is described in Using `SimplePasswordValidationCallbackHandler`.

Using `SpringDigestPasswordValidationCallbackHandler`

The `SpringDigestPasswordValidationCallbackHandler` requires a Spring Security `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 by using the `userDetailsService` property. Additionally, you can set a `userCache` property, to cache loaded user details. The following example shows how to do so:

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

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

Certificate Authentication

A more secure way of authentication uses X509 certificates. In this scenario, the SOAP message contains a `BinarySecurityToken`, which contains a Base 64-encoded version of a X509 certificate. The certificate is used by the recipient to authenticate. The certificate stored in the message is also used to sign the message (see 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 are covered in Verifying Signatures. For more details, check the official reference of possible child elements. The following listing shows how to define a `RequireSignature` element:

```
<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` returns a SOAP fault to the sender. If it is present, it fires a `CertificateValidationCallback`. Two handlers within Spring-WS handle this callback for authentication purposes:

- `KeyStoreCallbackHandler`
- `SpringCertificateValidationCallbackHandler`

NOTE

In most cases, certificate authentication should be preceded by certificate validation, since you want to authenticate against only 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 `SpringCertificateValidationCallbackHandler` 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.XwsSecurityIntercep
tor">
    <property name="policyConfiguration"
value="classpath:securityPolicy.xml"/>
    <property name="callbackHandlers">
        <list>
            <ref bean="keyStoreHandler"/>
            <ref bean="springSecurityHandler"/>
        </list>
    </property>
</bean>
```

Using this setup, the interceptor first determines if the certificate in the message is valid by using the keystore and then authenticating against it.

Using `KeyStoreCallbackHandler`

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

1. The handler checks 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 checks whether 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 continues with the final step.
3. A certification path for the certificate is created. This basically means that the handler determines 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 valid.

To use the `KeyStoreCallbackHandler` for certificate validation purposes, you most likely need to 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 the setup shown in the preceding example, the certificate that is to be validated must be in the trust store itself or the trust store must contain a certificate authority that issued the certificate.

Using `SpringCertificateValidationCallbackHandler`

The `SpringCertificateValidationCallbackHandler` requires an Spring Security `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 that can handle this token (usually an instance of `X509AuthenticationProvider`). If authentication is successful, the token is stored in the `SecurityContextHolder`. You can set the authentication manager by using the `authenticationManager` property:

```
<beans>
    <bean id="springSecurityCertificateHandler"

class="org.springframework.ws.soap.security.xwss.callback.SpringCertificateValidat
ionCallbackHandler">
        <property name="authenticationManager" ref="authenticationManager"/>
    </bean>

    <bean id="authenticationManager"
```

```
        class="org.springframework.security.providers.ProviderManager">
        <property name="providers">
            <bean
class="org.springframework.ws.soap.security.x509.X509AuthenticationProvider">
                <property name="x509AuthoritiesPopulator">
                    <bean
class="org.springframework.ws.soap.security.x509.populator.DaoX509AuthoritiesPopul
ator">
                        <property name="userDetailsService"
ref="userDetailsService"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

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

In this case, we use a custom user details service to obtain authentication details based on the certificate. See the Spring Security reference documentation for more information about authentication against X509 certificates.

7.1.3. Digital Signatures

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

Verifying Signatures

As with 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 that was expected to be signed and various other sub-elements. You can also define the private key alias to use, whether to use a symmetric instead of a private key, and many other properties. For more details, check the official reference of possible child elements. The following listing configures a `RequireSignature` element:

```
<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` returns a SOAP fault to the sender. If it is present, it fires a `SignatureVerificationKeyCallback` to the registered handlers. Within Spring-WS, one class handles this particular callback: `KeyStoreCallbackHandler`.

Using `KeyStoreCallbackHandler`

As described in KeyStoreCallbackHandler, `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>
```

Signing Messages

When signing a message, the `XwsSecurityInterceptor` adds the `BinarySecurityToken` to the message. It also adds 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 that was expected to be signed and various other sub-elements. You can also define the private key alias to use, whether to use a symmetric instead of a private key, and many other properties. For more details, check the official reference of possible child elements. The following example includes a `Sign` element:

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

The `XwsSecurityInterceptor` fires a `SignatureKeyCallback` to the registered handlers. Within Spring-WS, the `KeyStoreCallbackHandler` class handles this particular callback.

Using `KeyStoreCallbackHandler`

As described in KeyStoreCallbackHandler, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` to handle 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. The following example uses a `KeyStoreCallbackHandler`:

```
<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>
```

7.1.4. Decryption and Encryption

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

Decryption

To decrypt incoming SOAP messages, the security policy file should contain a `RequireEncryption` element. This element can further carry a `EncryptionTarget` element that 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. For more details, check the official reference of the other elements. The following example uses a `RequireEncryption` element:

```
<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` returns a SOAP fault to the sender. If it is present, it fires a `DecryptionKeyCallback` to the registered handlers. Within Spring-WS, the `KeyStoreCallbackHandler` class handles this particular callback.

Using `KeyStoreCallbackHandler`

As described in KeyStoreCallbackHandler, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` to handle 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 uses the `symmetricStore`. The following example uses `KeyStoreCallbackHandler`:

```
<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>
```

Encryption

To encrypt outgoing SOAP messages, the security policy file should contain an `Encrypt` element. This element can further carry a `EncryptionTarget` element that indicates which part of the message should be encrypted and a `SymmetricKey` to indicate that a shared secret instead of the regular public key should be used to encrypt the message. For more details, check the official reference of the other elements. The following example uses an `Encrypt` element:

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

The `XwsSecurityInterceptor` fires an `EncryptionKeyCallback` to the registered handlers to retrieve the encryption information. Within Spring-WS, the `KeyStoreCallbackHandler` class handles this particular callback.

Using `KeyStoreCallbackHandler`

As described in KeyStoreCallbackHandler, the `KeyStoreCallbackHandler` uses a `java.security.KeyStore` to handle various cryptographic callbacks, including encryption. For encryption based on public keys, the handler uses the `trustStore` property. For encryption based on symmetric keys, it uses `symmetricStore`. The following example uses `KeyStoreCallbackHandler`:

```
<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>
```

7.1.5. Security Exception Handling

When a securement or validation action fails, the `XwsSecurityInterceptor` throws a `WsSecuritySecurementException` or `WsSecurityValidationException` respectively. These exceptions bypass the standard exception handling mechanism but are handled by the interceptor itself.

`WsSecuritySecurementException` exceptions are handled by the `handleSecurementException` method of the `XwsSecurityInterceptor`. By default, this method logs an error and stops further processing of the message.

Similarly, `WsSecurityValidationException` exceptions are handled by the `handleValidationException` method of the `XwsSecurityInterceptor`. By default, this method creates a SOAP 1.1 Client or SOAP 1.2 sender fault and sends that back as a response.

NOTE	Both `handleSecurementException` and `handleValidationException` are protected methods, which you can override to change their default behavior.

7.2. Using `Wss4jSecurityInterceptor`

The `Wss4jSecurityInterceptor` is an `EndpointInterceptor` (see Intercepting Requests — the `EndpointInterceptor` Interface) that is based on Apache's WSS4J.

WSS4J implements the following standards:

- OASIS Web Services Security: SOAP Message Security 1.0 Standard 200401, March 2004.
- Username Token profile V1.0.
- X.509 Token Profile V1.0.

This interceptor supports messages created by the `SaajSoapMessageFactory`.

7.2.1. Configuring `Wss4jSecurityInterceptor`

WSS4J uses no external configuration file. The interceptor is entirely configured by properties. The validation and securement actions invoked by this interceptor are specified via `validationActions` and `securementActions` properties, respectively. Actions are passed as a space-separated strings. The following listing shows an example configuration:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="validationActions" value="UsernameToken Encrypt"/>
    ...
    <property name="securementActions" value="Encrypt"/>
    ...
</bean>
```

The following table shows the available validation actions:

Validation action	Description
`UsernameToken`	Validates username token
`Timestamp`	Validates the timestamp
`Encrypt`	Decrypts the message
`Signature`	Validates the signature
`NoSecurity`	No action performed

The following table shows the available securement actions:

Securement action	Description
`UsernameToken`	Adds a username token
`UsernameTokenSignature`	Adds a username token and a signature username token secret key
`Timestamp`	Adds a timestamp
`Encrypt`	Encrypts the response
`Signature`	Signs the response
`NoSecurity`	No action performed

The order of the actions is significant and is enforced by the interceptor. If its security actions were performed in a different order than the one specified by `validationActions`, the interceptor rejects an incoming SOAP message.

7.2.2. Handling Digital Certificates

For cryptographic operations that require interaction with a keystore or certificate handling (signature, encryption, and decryption operations), WSS4J requires an instance of

`org.apache.ws.security.components.crypto.Crypto`.

`Crypto` instances can be obtained from WSS4J's `CryptoFactory` or more conveniently with the Spring-WS `CryptoFactoryBean`.

CryptoFactoryBean

Spring-WS provides a convenient factory bean, `CryptoFactoryBean`, that constructs and configures `Crypto` instances through strongly typed properties (preferred) or through a `Properties` object.

By default, `CryptoFactoryBean` returns instances of `org.apache.ws.security.components.crypto.Merlin`. You can change this by setting the `cryptoProvider` property (or its equivalent `org.apache.ws.security.crypto.provider` string property).

The following example configuration uses `CryptoFactoryBean`:

```
<bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
    <property name="keyStorePassword" value="mypassword"/>
    <property name="keyStoreLocation"
value="file:/path_to_keystore/keystore.jks"/>
</bean>
```

7.2.3. Authentication

This section addresses how to do authentication with `Wss4jSecurityInterceptor`.

Validating Username Token

Spring-WS provides a set of callback handlers to integrate with Spring Security. Additionally, a simple callback handler, `SimplePasswordValidationCallbackHandler`, is provided to configure users and passwords with an in-memory `Properties` object.

Callback handlers are configured through the `validationCallbackHandler` of the `Wss4jSecurityInterceptor` property.

Using `SimplePasswordValidationCallbackHandler`

`SimplePasswordValidationCallbackHandler` validates plain text and digest username tokens against an in-memory `Properties` object. You can configure it as follows:

```
<bean id="callbackHandler"

class="org.springframework.ws.soap.security.wss4j.callback.SimplePasswordValidatio
nCallbackHandler">
    <property name="users">
        <props>
```

```
            <prop key="Bert">Ernie</prop>
        </props>
    </property>
</bean>
```

Using `SpringSecurityPasswordValidationCallbackHandler`

The `SpringSecurityPasswordValidationCallbackHandler` validates plain text and digest passwords by using a Spring Security `UserDetailService` to operate. It uses this service to retrieve the password (or a digest of the password) of the user specified in the token. The password (or a 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 by using the `userDetailsService`. Additionally, you can set a `userCache` property, to cache loaded user details, as follows:

```
<beans>
    <bean
class="org.springframework.ws.soap.security.wss4j.callback.SpringDigestPasswordVal
idationCallbackHandler">
        <property name="userDetailsService" ref="userDetailsService"/>
    </bean>

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

Adding Username Token

Adding a username token to an outgoing message is as simple as adding `UsernameToken` to the `securementActions` property of the `Wss4jSecurityInterceptor` and specifying `securementUsername` and `securementPassword`.

The password type can be set by setting the `securementPasswordType` property. Possible values are `PasswordText` for plain text passwords or `PasswordDigest` for digest passwords, which is the default.

The following example generates a username token with a digest password:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="securementActions" value="UsernameToken"/>
    <property name="securementUsername" value="Ernie"/>
    <property name="securementPassword" value="Bert"/>
</bean>
```

If the plain text password type is chosen, it is possible to instruct the interceptor to add `Nonce` and `Created` elements by setting the `securementUsernameTokenElements` property. The value must be a list that contains the desired elements' names separated by spaces (case-sensitive).

The following example generates a username token with a plain text password, a `Nonce`, and a `Created` element:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="securementActions" value="UsernameToken"/>
    <property name="securementUsername" value="Ernie"/>
    <property name="securementPassword" value="Bert"/>
    <property name="securementPasswordType" value="PasswordText"/>
    <property name="securementUsernameTokenElements" value="Nonce Created"/>
</bean>
```

Certificate Authentication

As certificate authentication is akin to digital signatures, WSS4J handles it as part of the signature validation and securement. Specifically, the `securementSignatureKeyIdentifier` property must be set to `DirectReference` in order to instruct WSS4J to generate a `BinarySecurityToken` element containing the X509 certificate and to include it in the outgoing message. The certificate's name and password are passed through the `securementUsername` and `securementPassword` properties, respectively, as the following example shows:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="securementActions" value="Signature"/>
    <property name="securementSignatureKeyIdentifier" value="DirectReference"/>
    <property name="securementUsername" value="mycert"/>
    <property name="securementPassword" value="certpass"/>
    <property name="securementSignatureCrypto">
      <bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
        <property name="keyStorePassword" value="123456"/>
        <property name="keyStoreLocation" value="classpath:/keystore.jks"/>
      </bean>
    </property>
</bean>
```

For the certificate validation, regular signature validation applies:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Signature"/>
    <property name="validationSignatureCrypto">
```

```
      <bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
        <property name="keyStorePassword" value="123456"/>
        <property name="keyStoreLocation" value="classpath:/keystore.jks"/>
      </bean>
    </property>
</bean>
```

At the end of the validation, the interceptor automatically verifies the validity of the certificate by delegating to the default WSS4J implementation. If needed, you can change this behavior by redefining the `verifyCertificateTrust` method.

For more detail, see to Digital Signatures.

7.2.4. Security Timestamps

This section describes the various timestamp options available in the `Wss4jSecurityInterceptor`.

Validating Timestamps

To validate timestamps, add `Timestamp` to the `validationActions` property. You can override timestamp semantics specified by the initiator of the SOAP message by setting `timestampStrict` to `true` and specifying a server-side time-to-live in seconds (default: 300) by setting the `timeToLive` property. The interceptor always rejects already expired timestamps, whatever the value of `timeToLive` is.

In the following example, the interceptor limits the timestamp validity window to 10 seconds, rejecting any valid timestamp token outside that window:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Timestamp"/>
    <property name="timestampStrict" value="true"/>
    <property name="timeToLive" value="10"/>
</bean>
```

Adding Timestamps

Adding `Timestamp` to the `securementActions` property generates a timestamp header in outgoing messages. The `timestampPrecisionInMilliseconds` property specifies whether the precision of the generated timestamp is in milliseconds. The default value is `true`. The following listing adds a timestamp:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="securementActions" value="Timestamp"/>
    <property name="timestampPrecisionInMilliseconds" value="true"/>
```

```
</bean>
```

7.2.5. Digital Signatures

This section describes the various signature options available in the `Wss4jSecurityInterceptor`.

Verifying Signatures

To instruct the `Wss4jSecurityInterceptor`, `validationActions` must contain the `Signature` action. Additionally, the `validationSignatureCrypto` property must point to the keystore containing the public certificates of the initiator:

```
<bean id="wsSecurityInterceptor"
class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Signature"/>
    <property name="validationSignatureCrypto">
        <bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
            <property name="keyStorePassword" value="123456"/>
            <property name="keyStoreLocation" value="classpath:/keystore.jks"/>
        </bean>
    </property>
</bean>
```

Signing Messages

Signing outgoing messages is enabled by adding the `Signature` action to the `securementActions`. The alias and the password of the private key to use are specified by the `securementUsername` and `securementPassword` properties, respectively. `securementSignatureCrypto` must point to the keystore that contains the private key:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="securementActions" value="Signature"/>
    <property name="securementUsername" value="mykey"/>
    <property name="securementPassword" value="123456"/>
    <property name="securementSignatureCrypto">
        <bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
            <property name="keyStorePassword" value="123456"/>
            <property name="keyStoreLocation" value="classpath:/keystore.jks"/>
        </bean>
    </property>
</bean>
```

Furthermore, you can define the signature algorithm by setting the `securementSignatureAlgorithm` property.

You can customize the key identifier type to use by setting the `securementSignatureKeyIdentifier` property. Only `IssuerSerial` and `DirectReference` are valid for the signature.

The `securementSignatureParts` property controls which part of the message is signed. The value of this property is a list of semicolon-separated element names that identify the elements to sign. The general form of a signature part is `{}{namespace}Element`. Note that the first empty brackets are used for encryption parts only. The default behavior is to sign the SOAP body.

The following example shows how to sign the `echoResponse` element in the Spring-WS echo sample:

```
<property name="securementSignatureParts"
    value="{}{http://www.springframework.org/spring-
ws/samples/echo}echoResponse"/>
```

To specify an element without a namespace, use the string, `Null` (case sensitive), as the namespace name.

If no other element in the request has a local name of `Body`, the SOAP namespace identifier can be empty (`{}`).

Signature Confirmation

Signature confirmation is enabled by setting `enableSignatureConfirmation` to `true`. Note that the signature confirmation action spans over the request and the response. This implies that `secureResponse` and `validateRequest` must be set to `true` (which is the default value) even if there are no corresponding security actions. The following example sets the `enableSignatureConfirmation` property to `true`:

```
<bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Signature"/>
    <property name="enableSignatureConfirmation" value="true"/>
    <property name="validationSignatureCrypto">
        <bean
class="org.springframework.ws.soap.security.wss4j.support.CryptoFactoryBean">
            <property name="keyStorePassword" value="123456"/>
            <property name="keyStoreLocation" value="file:/keystore.jks"/>
        </bean>
    </property>
</bean>
```

7.2.6. Decryption and Encryption

This section describes the various decryption and encryption options available in the `Wss4jSecurityInterceptor`.

Decryption

Decryption of incoming SOAP messages requires that the `Encrypt` action be added to the `validationActions` property. The rest of the configuration depends on the key information that appears in the message. (This is because WSS4J needs only a Crypto for encrypted keys, whereas embedded key name validation is delegated to a callback handler).

To decrypt messages with an embedded encrypted symmetric key (the `xenc:EncryptedKey` element), `validationDecryptionCrypto` needs to point to a keystore that contains the decryption private key. Additionally, `validationCallbackHandler` has to be injected with a `KeyStoreCallbackHandler` that specifies the key's password:

```
<bean
class="org.springframework.ws.soap.security.wss4j2.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Encrypt"/>
    <property name="validationDecryptionCrypto">
        <bean
class="org.springframework.ws.soap.security.wss4j2.support.CryptoFactoryBean">
            <property name="keyStorePassword" value="123456"/>
            <property name="keyStoreLocation" value="classpath:/keystore.jks"/>
        </bean>
    </property>
    <property name="validationCallbackHandler">
        <bean
class="org.springframework.ws.soap.security.wss4j2.callback.KeyStoreCallbackHandle
r">
            <property name="privateKeyPassword" value="mykeypass"/>
        </bean>
    </property>
</bean>
```

To support decryption of messages with an embedded key name (`ds:KeyName` element), you can configure a `KeyStoreCallbackHandler` that points to the keystore with the symmetric secret key. The `symmetricKeyPassword` property indicates the key's password, the key name being the one specified by `ds:KeyName` element:

```
<bean
class="org.springframework.ws.soap.security.wss4j2.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Encrypt"/>
    <property name="validationCallbackHandler">
        <bean
```

```
class="org.springframework.ws.soap.security.wss4j2.callback.KeyStoreCallbackHandle
r">
            <property name="keyStore">
                <bean
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
                    <property name="location" value="classpath:keystore.jks"/>
                    <property name="type" value="JCEKS"/>
                    <property name="password" value="123456"/>
                </bean>
            </property>
            <property name="symmetricKeyPassword" value="mykeypass"/>
        </bean>
    </property>
</bean>
```

Encryption

Adding `Encrypt` to the `securementActions` enables encryption of outgoing messages. You can set the certificate's alias to use for the encryption by setting the `securementEncryptionUser` property. The keystore where the certificate resides is accessed through the `securementEncryptionCrypto` property. As encryption relies on public certificates, no password needs to be passed. The following example uses the `securementEncryptionCrypto` property:

```
<bean
class="org.springframework.ws.soap.security.wss4j2.Wss4jSecurityInterceptor">
    <property name="securementActions" value="Encrypt"/>
    <property name="securementEncryptionUser" value="mycert"/>
    <property name="securementEncryptionCrypto">
        <bean
class="org.springframework.ws.soap.security.wss4j2.support.CryptoFactoryBean">
            <property name="keyStorePassword" value="123456"/>
            <property name="keyStoreLocation" value="file:/keystore.jks"/>
        </bean>
    </property>
</bean>
```

You can customize encryption in several ways: The key identifier type to use is defined by the `securementEncryptionKeyIdentifier` property. Possible values are `IssuerSerial`,`X509KeyIdentifier`, `DirectReference`,`Thumbprint`, `SKIKeyIdentifier`, and `EmbeddedKeyName`.

If you choose the `EmbeddedKeyName` type, you need to specify the secret key to use for the encryption. The alias of the key is set in the `securementEncryptionUser` property, as for the other key identifier types. However, WSS4J requires a callback handler to fetch the secret key. Thus, you must provide `securementCallbackHandler` with a `KeyStoreCallbackHandler` that points to the appropriate keystore. By default, the `ds:KeyName` element in the resulting WS-Security header takes the value of the `securementEncryptionUser` property. To indicate a different name, you can set the

securementEncryptionEmbeddedKeyName with the desired value. In the next example, the outgoing message is encrypted with a key aliased secretKey, whereas myKey appears in ds:KeyName element:

```
<bean
class="org.springframework.ws.soap.security.wss4j2.Wss4jSecurityInterceptor">
    <property name="securementActions" value="Encrypt"/>
    <property name="securementEncryptionKeyIdentifier" value="EmbeddedKeyName"/>
    <property name="securementEncryptionUser" value="secretKey"/>
    <property name="securementEncryptionEmbeddedKeyName" value="myKey"/>
    <property name="securementCallbackHandler">
        <bean
class="org.springframework.ws.soap.security.wss4j2.callback.KeyStoreCallbackHandle
r">
            <property name="symmetricKeyPassword" value="keypass"/>
            <property name="keyStore">
                <bean
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
                    <property name="location" value="file:/keystore.jks"/>
                    <property name="type" value="jceks"/>
                    <property name="password" value="123456"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

The securementEncryptionKeyTransportAlgorithm property defines which algorithm to use to encrypt the generated symmetric key. Supported values are http://www.w3.org/2001/04/xmlenc#rsa-1_5, which is the default, and http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

You can set the symmetric encryption algorithm to use by setting the securementEncryptionSymAlgorithm property. Supported values are http://www.w3.org/2001/04/xmlenc#aes128-cbc (default), http://www.w3.org/2001/04/xmlenc#tripledes-cbc, http://www.w3.org/2001/04/xmlenc#aes256-cbc, and http://www.w3.org/2001/04/xmlenc#aes192-cbc.

Finally, the securementEncryptionParts property defines which parts of the message are encrypted. The value of this property is a list of semicolon-separated element names that identify the elements to encrypt. An encryption mode specifier and a namespace identification, each inside a pair of curly brackets, may precede each element name. The encryption mode specifier is either {Content} or {Element} See the W3C XML Encryption specification about the differences between Element and Content encryption. The following example identifies the echoResponse from the echo sample:

```
<property name="securementEncryptionParts"
    value="{Content}{http://www.springframework.org/spring-
ws/samples/echo}echoResponse"/>
```

Be aware that the element name, the namespace identifier, and the encryption modifier are case-sensitive. You can omit the encryption modifier and the namespace identifier. If you do, the encryption mode defaults to `Content`, and the namespace is set to the SOAP namespace.

To specify an element without a namespace, use the value, `Null` (case sensitive), as the namespace name. If no list is specified, the handler encrypts the SOAP Body in `Content` mode by default.

7.2.7. Security Exception Handling

The exception handling of the `Wss4jSecurityInterceptor` is identical to that of the `XwsSecurityInterceptor`. See Security Exception Handling for more information.

III. Other Resources

In addition to this reference documentation, a number of other resources 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 & Edmund Smith. *Rethinking the Java SOAP Stack*. May 17, 2005. © 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