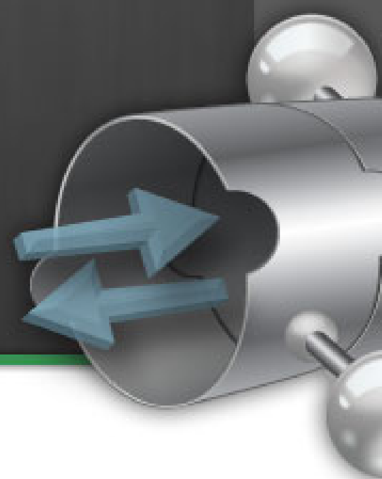
talend*
*open integration solutions



Talend Enterprise Service Factory

User Guide

Covers Apache CXF 2.5.x/2.6.x series

Talend Enterprise Service Factory: User Guide

Publication date 3 May 2012
Copyright © 2011-2012 Talend Inc.

Copyleft

This documentation is provided under the terms of the Creative Commons Public License (CCPL). For more information about what you can and cannot do with this documentation in accordance with the CCPL, please read: http://creativecommons.org/licenses/by-nc-sa/2.0/

This document may include documentation produced at The Apache Software Foundation which is licensed under The Apache License 2.0.

Notices

Talend and Talend ESB are trademarks of Talend, Inc.

Apache CXF, CXF, Apache Karaf, Karaf, Apache Camel, Camel, Apache Maven, Maven, Apache Archiva, Archiva are trademarks of The Apache Foundation.

Eclipse Equinox is a trademark of the Eclipse Foundation, Inc. SoapUI is a trademark of SmartBear Software. Hyperic is a trademark of VMware, Inc. Nagios is a trademark of Nagios Enterprises, LLC.

All other brands, product names, company names, trademarks and service marks are the properties of their respective owners.

Table of Contents

1. Introduction to Service Creation with Talend ESB 1
2. JAX-WS Development 2
    2.1. JAX-WS Overview 2
        2.1.1. Spring Integration 2
        2.1.2. Transports 2
        2.1.3. Support for Various Databindings between XML and Java 3
        2.1.4. Bindings 3
        2.1.5. Message Interception and Modification 3
        2.1.6. JAX-WS Handlers 3
        2.1.7. Interceptors 3
        2.1.8. Transmitting Binary Data 3
        2.1.9. WS-* Support 4
        2.1.10. Invokers 4
    2.2. JAX-WS Service Development Options 4
        2.2.1. JAX-WS Annotated Services from Java 4
        2.2.2. JAX-WS Annotated Services from WSDL 5
        2.2.3. Developing a Service using JAX-WS 5
        2.2.4. JAX-WS Configuration 19
        2.2.5. JAX-WS Providers 28
        2.2.6. WebserviceContext 34
    2.3. JAX-WS Client Development Options 34
        2.3.1. WSDL2Java generated Client 34
        2.3.2. JAX-WS Proxy 35
        2.3.3. JAX-WS Dispatch APIs 35
        2.3.4. Usage Modes 36
        2.3.5. Data Types 37
        2.3.6. Working with Dispatch Objects 38
        2.3.7. Developing a Consumer 41
    2.4. Data Binding Options 56
        2.4.1. Aegis 57
        2.4.2. JAXB 66
        2.4.3. MTOM Attachments with JAXB 69
        2.4.4. SDO 72
        2.4.5. XMLBeans 73
    2.5. CXF Transports 74
        2.5.1. HTTP Transport 74
        2.5.2. JMS Transport 94
    2.6. WS-* Support 110
        2.6.1. WS-Addressing 110
        2.6.2. WS-Policy 111
        2.6.3. WS-ReliableMessaging 121
        2.6.4. WS-SecureConversation 122
        2.6.5. WS-Security 123
        2.6.6. WS-SecurityPolicy 133
        2.6.7. WS-Trust 136
    2.7. CXF Customizations 139
        2.7.1. Annotations 139
        2.7.2. Dynamic Clients 142
    2.8. CXF Command-Line Tools 144
        2.8.1. WSDL to Java 144
        2.8.2. Java to WS 146
    2.9. JAX-WS Development With Eclipse 147
3. JAX-RS Development 153
    3.1. JAX-RS Overview 153
        3.1.1. Root Resources and Sub Resources 153
        3.1.2. Path, HTTP Method and MediaType annotations 155

3.1.3. Request Message, Parameters and Contexts ... 156
3.1.4. Responses from Resource Methods ... 157
3.1.5. Exception Handling ... 157
3.1.6. Custom JAX-RS Providers ... 158
3.2. Client API ... 158
3.2.1. HTTP Centric API ... 158
3.2.2. Proxy API ... 159
3.2.3. Reading and Writing HTTP Messages ... 160
3.2.4. Exception Handling ... 161
3.3. Working With Attachments ... 161
3.3.1. Reading Attachments ... 161
3.3.2. Writing Attachments ... 162
3.3.3. Uploading files ... 164
3.3.4. Forms and multiparts ... 164
3.3.5. XOP support ... 166
3.4. Configuration ... 166
3.4.1. Configuration of Endpoints ... 166
3.4.2. Configuration of Clients ... 168
3.5. Tutorials ... 168
3.5.1. Creating a Basic JAX-RS endpoint ... 168
4. JAX-RS and OAuth2 ... 173
4.1. Introduction to OAuth2 ... 173
4.2. Developing OAuth2 Servers ... 174
4.2.1. Authorization Service ... 174
4.2.2. AccessTokenService ... 177
4.2.3. Writing OAuthDataProvider ... 178
4.2.4. OAuth Server JAX-RS endpoints ... 179
4.3. Protecting resources with OAuth2 filters ... 180
4.4. How to get the user login name ... 180
4.5. Client-side support ... 181
4.6. OAuth2 without Explicit Authorization ... 182
4.7. OAuth2 without a Browser ... 182
4.8. Controlling the Access to Resource Server ... 182
4.8.1. Sharing the same access path between end users and clients ... 183
4.8.2. Providing different access points to end users and clients ... 184
5. Combining JAX-WS and JAX-RS ... 185
5.1. Using Java-First Approach ... 185
5.2. Using Document-First Approach ... 186
6. Talend ESB Service Recommended Project Structure ... 187
7. Talend ESB Service Examples ... 188
8. Configuring JMX Integration ... 190
8.1. Example Configuration ... 191
8.2. How to get web service performance metrics ... 192

List of Examples

2.1. Implementation of the Greeter Service ... 7
2.2. Simple SEI ... 9
2.3. Implementation for SEI ... 9
2.4. Interface with the @WebService Annotation ... 11
2.5. Annotated Service Implementation Class ... 12
2.6. Specifying an RPC/LITERAL SOAP Binding ... 13
2.7. SEI with Annotated Methods ... 16
2.8. Fully Annotated SEI ... 17
2.9. Outline of a Generated Service Class ... 46
2.10. The Greeter Service Endpoint Interface ... 47
2.11. Setting a Request Context Property on the Client Side ... 49
2.12. Reading a Response Context Property on the Client Side ... 50
2.13. Template for an Asynchronous Binding Declaration ... 52
2.14. Service Endpoint Interface with Methods for Asynchronous Invocations ... 53
2.15. Polling Approach for an Asynchronous Operation Call ... 54
2.16. The javax.xml.ws.AsyncHandler Interface ... 55
2.17. The TestAsyncHandler Callback Class ... 55
2.18. Callback Approach for an Asynchronous Operation Call ... 56
2.19. HTTP Consumer Configuration Namespace ... 76
2.20. http-conf:conduit Element ... 76
2.21. HTTP Consumer Endpoint Configuration ... 79
2.22. HTTP conduit configuration disabling HTTP URL hostname verification (usage of localhost, etc) ... 80
2.23. HTTP Consumer WSDL Element's Namespace ... 80
2.24. WSDL to Configure an HTTP Consumer Endpoint ... 81
2.25. Adding the Configuration Namespace ... 86
2.26. http-conf:destination Element ... 86
2.27. HTTP Service Provider Endpoint Configuration ... 87
2.28. HTTP Provider WSDL Element's Namespace ... 88
2.29. WSDL to Configure an HTTP Service Provider Endpoint ... 88
2.30. JMS Extension Namespace ... 95
2.31. JMS Configuration Namespaces ... 95
2.32. JMS WSDL Port Specification ... 97
2.33. Addressing Information in a Configuration File ... 98
2.34. Configuration for a JMS Consumer Endpoint ... 99
2.35. Configuration for a JMS Service Endpoint ... 101
2.36. JMS Session Pool Configuration ... 102
2.37. JMS Consumer Endpoint Runtime Configuration ... 103
2.38. JMS Service Endpoint Runtime Configuration ... 103

Chapter 1. Introduction to Service Creation with Talend ESB

Talend ESB provides users with an easy-to-use solution for service enablement. Talend ESB incorporates the industry leading open source Apache CXF implementation of JAX-WS and helps you create new services and also service enable your existing applications and interfaces. It provides a lightweight, modular architecture that is based on the popular Spring Framework, so it works with your application, regardless of the platform on which it is running. It can be run as a stand-alone Java applications, as part of a servlet engine, such as Tomcat, as an OSGi bundle on an OSGi container such as Equinox, or within a JEE server.

Talend ESB supports the creation of SOAP and REST web services, with full WS-*functionality, including support for WS- Addressing, WS-Reliable Messaging, and WS-Security over both HTTP and JMS transports. Developers use a declarative, policy-centric approach to enable different qualities of service through configuration, rather than code.

CXF has been certified and tested against the broadest set of vendor implementations for the various WS standards. Users benefit from this interoperability testing, which reduces the overall cost and complexity for application integration.

The Talend ESB distribution goes beyond Apache CXF, with support for OSGi containers along with illustrative examples, freely available for download. CXF development tools include support for Maven plug-ins, WSDL document creation, and Spring configuration generation.

Chapter 2. JAX-WS Development

2.1. JAX-WS Overview

CXF implements the JAX-WS APIs which make building web services easy. JAX-WS encompasses many different areas: Generating WSDL from Java classes and generating Java classes from WSDL, a Provider API which allows you to create simple messaging receiving server endpoints, and a Dispatch API which allows you to send raw XML messages to server endpoints. Apache CXF supports a variety of web service specifications including WS-Addressing, WS-Policy, WS-ReliableMessaging and WS-Security. Architectural aspects of CXF include the following:

2.1.1. Spring Integration

Spring is a first class citizen with Apache CXF. CXF supports the Spring 2.0 XML syntax, making it trivial to declare endpoints which are backed by Spring and inject clients into your application.

2.1.2. Transports

CXF works with many different transports. Currently CXF includes support for HTTP, JMS, and Local (that is, "in-JVM") transports. The local transport is unique in that it will not work across machines, but simply sends messages in memory. You can also configure the local transport to avoid serialization by using the Object binding or the colocation feature if desired. You can also write your own transport.

2.1.3. Support for Various Databindings between XML and Java

CXF provides support for multiple databindings, including JAXB, XML Beans, and Aegis Databinding (2.0.x), is our own databinding library that makes development of code-first web services incredibly easy. Unlike JAXB, you don't need annotations at all. It also works correctly with a variety of datatypes such as Lists, Maps, Dates, etc. right out of the box. If you're building a prototype web services that's really invaluable as it means you have to do very little work to get up and running.

2.1.4. Bindings

Bindings map a particular service's messages to a particular protocol. CXF includes support for several different bindings. The SOAP binding, which is the default, maps messages to SOAP and can be used with the various WS-* modules inside CXF. The Pure XML binding avoids serialization of a SOAP envelope and just sends a raw XML message. There is also an HTTP Binding which maps a service to HTTP using RESTful semantics.

2.1.5. Message Interception and Modification

Many times you may want to provide functionality for your application that works at a low level with XML messages. This commonly occurs through functionality referred to as Handlers or Interceptors. Handlers/ Interceptors are useful for:

- Performing authentication based on Headers
- Processing custom headers
- Transforming a message (i.e. via XSLT or GZip)
- Redirecting a message
- Getting access to the raw I/O or XML stream

2.1.6. JAX-WS Handlers

If you are using the JAX-WS frontend, JAX-WS supports the concept of logical and protocol handlers. Protocol handlers allow you to manipulate the message in its raw, often XML-based, form - i.e. a SAAJ SOAPMessage. Logical handlers allow you to manipulate the message after its already been bound from the protocol to the JAXB object that your service will receive.

2.1.7. Interceptors

Interceptors provide access to all the features that CXF has to offer - allowing you to do just about anything, including manipulating the raw bytes or XML of the message.

2.1.8. Transmitting Binary Data

CXF provides facilities to transmit binary data efficiently via a standard called MTOM. Normally binary data inside an XML message must be Base64 encoded. This results in processing overhead and increases message

size by 30%. If you use MTOM, CXF will send/receive MIME messages with the message stored as a MIME attachment, just like email. This results in much more efficient communication and allows you to transmit messages much larger than memory.

2.1.9. WS-* Support

CXF supports a variety of web service specifications including WS-Addressing, WS-Policy, WS-ReliableMessaging and WS-Security.

2.1.10. Invokers

Invokers allow you to customize how a particular method or backend service object is executed. This is particularly useful if your underlying service objects are not plain javabeans and instead need to be created or looked up via a custom factory.

2.2. JAX-WS Service Development Options

2.2.1. JAX-WS Annotated Services from Java

The JAX-WS APIs include a set of annotations which allow you to build services using annotated classes. These services are based on a single class which contains a set of operations.

Here's a simple example:

```
@WebService
public class Hello {
  public String sayHi(String name) {
    return "Hello " + name;
  }
}
```

JAX-WS includes many more annotations as well such as:

- @WebMethod - allows you to customize the operation name, exclude the operation from inclusion in the service, etc
- @WebParam - allows you to customize a parameter's name, namespace, direction (IN or OUT), etc
- @WebResult - allows you to customize the return value of the web service call

Data is marshalled from XML to Java and vice versa via the JAXB data-binding.

Services are publish via one of two means:

- The JAX-WS standard Endpoint APIs
- CXF's XML configuration format - i.e. <jaxws:endpoint ... />

2.2.2. JAX-WS Annotated Services from WSDL

If you have existing WSDLs for your service or wish to write your WSDL first and then generate classes, CXF has many tools to help you do this.

The WSDL2Java tool will generate a JAX-WS annotated service and server stub from your WSDL. You can run it one of three ways:

- The command line
- The Maven plugin
- With the WSDL2Java API

Note that CXF generally restricts WSDL support to WSI-BP, not the full WSDL 1.1 specification.

2.2.3. Developing a Service using JAX-WS

You can develop a service using one of two approaches:

- Start with a WSDL contract and generate Java objects to implement the service.
- Start with a Java object and service enable it using annotations. For new development the preferred path is to design your services in WSDL and then generate the code to implement them. This approach enforces the concept that a service is an abstract entity that is implementation neutral. It also means you can spend more time working out the exact interface your service requires before you start coding.

However, there are many cases where you may need to service enable an existing application. While JAX-WS eases the process, it does require that you make some changes to source code of your application. You will need to add annotations to the source. It also requires that you migrate your code to Java 5.0.

2.2.3.1. WSDL First Development

Using the WSDL first model of service development, you start with a WSDL document that defines the service you wish to implement. This WSDL document could be obtained from another developer, a system architect, a UDDI registry, or you could write it yourself. The document must contain at least a fully specified logical interface before you can begin generating code from it.

Once you have a WSDL document, the process for developing a JAX-WS service is three steps:

1. Generate starting point code.
2. Implement the service's operations.
3. Publish the implemented service.

Generating the Starting Point Code

JAX-WS specifies a detailed mapping from a service defined in WSDL to the Java classes that will implement that service. The logical interface, defined by the `wsdl:portType` element, is mapped to a service endpoint interface (SEI). Any complex types defined in the WSDL are mapped into Java classes following the mapping defined by the Java Architecture for XML Binding (JAXB) specification. The endpoint defined by the `wsdl:service` element is also generated into a Java class that is used by consumers to access endpoints implementing the service.

The **wsdl2java** command automates the generation of this code. It also provides options for generating starting point code for your implementation and an ant based makefile to build the application. **wsdl2java** provides a number of arguments for controlling the generated code.

Running wsdl2java

You can generate the code needed to develop your service using the following command: wsdl2java -ant -impl -server -d *outputDir* myService.wsdl

The command does the following:

- The `-ant` argument generates a Ant makefile, called `build.xml` , for your application.
- The `-impl` argument generates a shell implementation class for each portType element in the WSDL document.
- The `-server` argument generates a simple `main()` to launch your service as a stand alone application.
- The `-d outputDir` argument tells **wsdl2java** to write the generated code to a directory called outputDir.
- `myService.wsdl` is the WSDL document from which code is generated.

Generated code

Table1 [6] describes the files generated for creating a service.

Table 1: Generated Classes for a Service

File	Description
`portTypeName.java`	The SEI. This file contains the interface your service implements. You should not edit this file.
`serviceName.java`	The endpoint. This file contains the Java class your clients will use to make requests on the service.
`portTypeNameImpl.java`	The skeleton implementation class. You will modify this file to implement your service.
`portTypeName_portTypeName...` `ImplPort_Server.java`	A basic server `main()` that allows you to deploy your service as a stand alone process.

Implementing the Service

Once the starting point code is generated, you must provide the business logic for each of the operations defined in the service's interface.

Generating the implementation code

You generate the implementation class for your service with **wsdl2java** 's `-impl` flag.

Tip

If your service's contract included any custom types defined in XML Schema, you will also need to ensure that the classes for the types are also generated and available.

Generated code

The service implementation code consists of two files:

- `portTypeName.java` is the service interface(SEI) for the service.
- `portTypeNameImpl.java` is the class you will use to implement the operations defined for the service.

Implement the operation's logic

You provide the business logic for your service's operations by completing the stub methods in `portTypeNameImpl.java` . For the most part, you use standard Java to implement the business logic. If your service uses custom XML Schema types, you will need to use the generated classes for each type to manipulate them. There are also some CXF specific APIs that you can use to access some advanced features.

Example

For example, an implementation class for a service that defined the operations `sayHi` and `greetMe` may look like the below example.

Example 2.1. Implementation of the Greeter Service

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
 targetNamespace = "http://apache.org/hello_world_soap_http",
 endpointInterface = "org.apache.hello_world_soap_http.Greeter")

public class GreeterImpl implements Greeter {

   public String greetMe(String me) {
      System.out.println("Executing operation greetMe");
      System.out.println("Message received: " + me + "\n");
      return "Hello " + me;
   }

   public String sayHi() {
      System.out.println("Executing operation sayHi\n");
      return "Bonjour";
   }
}
```

2.2.3.2. Java First Development

To create a service starting from Java you need to do the following:

1. Create a Service Endpoint Interface (SEI) that defines the methods you wish to expose as a service.

Tip

You can work directly from a Java class, but working from an interface is the recommended approach. Interfaces are better for sharing with the developers who will be responsible for developing the applications consuming your service. The interface is smaller and does not provide any of the service's implementation details.

2. Add the required annotations to your code.
3. Generate the WSDL contract for your service.

Tip

If you intend to use the SEI as the service's contract, it is not necessary to generate a WSDL contract

4. Publish the service.

Creating the SEI

The service endpoint interface (SEI) is the piece of Java code that is shared between a service and the consumers that make requests on it. When starting with a WSDL contract, the SEI is generated by the code generators. However, when starting from Java, it is the up to a developer to create the SEI.

There are two basic patterns for creating an SEI:

- Green field development You are developing a new service from the ground up. When starting fresh, it is best to start by creating the SEI first. You can then distribute the SEI to any developers that are responsible for implementing the services and consumers that use the SEI.

Note

The recommended way to do green field service development is to start by creating a WSDL contract that defines the service and its interfaces.

- Service enablement In this pattern, you typically have an existing set of functionality that is implemented as a Java class and you want to service enable it. This means that you will need to do two things:
  1. Create an SEI that contains **only** the operations that are going to be exposed as part of the service.
  2. Modify the existing Java class so that it implements the SEI.

Note

You can add the JAX-WS annotations to a Java class, but that is not recommended.

Writing the interface

The SEI is a standard Java interface. It defines a set of methods that a class will implement. It can also define a number of member fields and constants to which the implementing class has access.

In the case of an SEI the methods defined are intended to be mapped to operations exposed by a service. The SEI corresponds to a `wsdl:portType` element. The methods defined by the SEI correspond to `wsdl:operation` elements in the `wsdl:portType` element.

Tip

JAX-WS defines an annotation that allows you to specify methods that are not exposed as part of a service. However, the best practice is to leave such methods out of the SEI.

The below shows a simple SEI for a stock updating service.

Example 2.2. Simple SEI

```
package org.apache.cxf;

public interface QuoteReporter
{
  public Quote getQuote(String ticker);
}
```

Implementing the interface

Because the SEI is a standard Java interface, the class that implements it is just a standard Java class. If you started with a Java class you will need to modify it to implement the interface. If you are starting fresh, the implementation class will need to implement the SEI.

The below shows a class for implementing the above [9] interface. .

Example 2.3. Implementation for SEI

```
package org.apache.cxf;

import java.util.*;

public class StockQuoteReporter implements QuoteReporter
{
  ...
public Quote getQuote(String ticker)
  {
    Quote retVal = new Quote();
    retVal.setID(ticker);
    retVal.setVal(Board.check(ticker));[1]
    Date retDate = new Date();
    retVal.setTime(retDate.toString());
    return(retVal);
  }
}
```

Annotating the Code

JAX-WS relies on the annotation feature of Java 5. The JAX-WS annotations are used to specify the metadata used to map the SEI to a fully specified service definition. Among the information provided in the annotations are the following:

- The target namespace for the service.
- The name of the class used to hold the request message.
- The name of the class used to hold the response message.
- If an operation is a one way operation.
- The binding style the service uses.
- The name of the class used for any custom exceptions.
- The namespaces under which the types used by the service are defined.

Tip

Most of the annotations have sensible defaults and do not need to be specified. However, the more information you provide in the annotations, the better defined your service definition. A solid service definition increases the likely hood that all parts of a distributed application will work together.

Required Annotations

In order to create a service from Java code you are only required to add one annotation to your code. You must add the `@WebService()` annotation on both the SEI and the implementation class.

The @WebService annotation

The `@WebService` annotation is defined by the `javax.jws.WebService` interface and it is placed on an interface or a class that is intended to be used as a service. `@WebService` has the following properties:

Property	Description
name	Specifies the name of the service interface. This property is mapped to the name attribute of the `wsdl:portType` element that defines the service's interface in a WSDL contract. The default is to append `PortType` to the name of the implementation class.
targetNamespace	Specifies the target namespace under which the service is defined. If this property is not specified, the target namespace is derived from the package name.
serviceName	Specifies the name of the published service. This property is mapped to the `name` attribute of the `wsdl:service` element that defines the published service. The default is to use the name of the service's implementation class. **Note:** Not allowed on the SEI
wsdlLocation	Specifies the URI at which the service's WSDL contract is stored. The default is the URI at which the service is deployed.
endpointInterface	Specifies the full name of the SEI that the implementation class implements. This property is only used when the attribute is used on a service implementation class. **Note:** Not allowed on the SEI
portName	Specifies the name of the endpoint at which the service is published. This property is mapped to the `name` attribute of the `wsdl:port` element that

Property	Description
	specifies the endpoint details for a published service. The default is the append `Port` to the name of the service's implementation class. **Note:** Not allowed on the SEI

Tip

You do not need to provide values for any of the `@WebService` annotation's properties. However, it is recommended that you provide as much information as you can.

Annotating the SEI

The SEI requires that you add the `@WebService` annotation. Since the SEI is the contract that defines the service, you should specify as much detail as you can about the service in the `@WebService` annotation's properties.

The below shows the interface defined in above [9] with the `@WebService` annotation.

Example 2.4. Interface with the @WebService Annotation

```
package com.mycompany.demo;

import javax.jws.*;

@WebService(name="quoteUpdater",
      targetNamespace="http://cxf.apache.org",
      wsdlLocation="http://somewhere.com/quoteExampleService?wsdl")
public interface QuoteReporter
{
  public Quote getQuote(@WebParam(name="ticker") String ticker);
}
```

The `@WebService` annotation above does the following:

1. Specifies that the value of the name attribute of the `wsdl:portType` element defining the service interface is `quoteUpdater` .
2. Specifies that the target namespace of the service is http://cxf.apache.org.
3. Specifies that the service will use the pre-defined WSDL contract published at http://somewhere.com/quoteExampleService?wsdl.

The @WebParam annotation is necessary as java interfaces do not store the Parameter name in the .class file. So if you leave out the annotation your parameter will be named arg0.

Annotating the service implementation

In addition to annotating the SEI with the `@WebService` annotation, you also have to annotate the service implementation class with the `@WebService` annotation. When adding the annotation to the service implementation class you only need to specify the `endpointInterface` property. As shown in the next example the property needs to be set to the full name of the SEI.

Example 2.5. Annotated Service Implementation Class

```
package org.apache.cxf;

import javax.jws.*;

@WebService(endpointInterface="org.apache.cxf.quoteReporter",
            targetNamespace="http://cxf.apache.org",
            portName="StockQuotePort",
            serviceName="StockQuoteReporter",
            )
public class StockQuoteReporter implements QuoteReporter
{
public Quote getQuote(String ticker)
  {
  ...
  }
}
```

Optional Annotations

While the `@WebService` annotation is sufficient for service enabling a Java interface or a Java class, it does not provide a lot of information about how the service will be exposed as an endpoint. The JAX-WS programming model uses a number of optional annotations for adding details about your service, such as the binding it uses, to the Java code. You add these annotations to the service's SEI.

Tip

The more details you provide in the SEIthe easier it will be for developers to implement applications that can use the functionality it defines. It will also provide for better generated WSDL contracts.

Defining the Binding Properties with Annotations

If you are using a SOAP binding for your service, you can use JAX-WS annotations to specify a number of the bindings properties. These properties correspond directly to the properties you can specify in a service's WSDL contract.

The @SOAPBinding annotation

The `@SOAPBinding` annotation is defined by the `javax.jws.soap.SOAPBinding` interface. It provides details about the SOAP binding used by the service when it is deployed. If the `@SOAPBinding` annotation is not specified, a service is published using a wrapped doc/literal SOAP binding.

You can put the `@SOAPBinding` annotation on the SEI and any of the SEI's methods. When it is used on a method, setting of the method's `@SOAPBinding` annotation take precedent.

The following table shows the properties for the @SOAPBinding annotation.

Property	Values	Description
style	`Style.DOCUMENT` (default) `Style.RPC`	Specifies the style of the SOAP message. If `RPC` style is specified, each

Property	Values	Description
		message part within the SOAP body is a parameter or return value and will appear inside a wrapper element within the `soap:body` element. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. If `DOCUMENT` style is specified, the contents of the SOAP body must be a valid XML document, but its form is not as tightly constrained.
use	`Use.LITERAL` (default) `Use.ENCODED`	Specifies how the data of the SOAP message is streamed.
parameterStyle	`ParameterStyle.BARE` `ParameterStyle.WRAPPED` (default)	Specifies how the method parameters, which correspond to message parts in a WSDL contract, are placed into the SOAP message body. A parameter style of `BARE` means that each parameter is placed into the message body as a child element of the message root. A parameter style of `WRAPPED` means that all of the input parameters are wrapped into a single element on a request message and that all of the output parameters are wrapped into a single element in the response message. If you set the style to RPC you must use the `WRAPPED` parameter style.

The below shows an SEI that uses rpc/literal SOAP messages.

Example 2.6. Specifying an RPC/LITERAL SOAP Binding

```
package org.eric.demo;

import javax.jws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;

@WebService(name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface QuoteReporter
{
  ...
}
```

Defining Operation Properties with Annotations

When the runtime maps your Java method definitions into XML operation definitions it fills in details such as:

- what the exchanged messages look like in XML.
- if the message can be optimized as a one way message.

- the namespaces where the messages are defined.

The @WebMethod annotation

The `@WebMethod` annotation is defined by the `javax.jws.WebMethod` interface. It is placed on the methods in the SEI. The `@WebMethod` annotation provides the information that is normally represented in the `wsdl:operation` element describing the operation to which the method is associated.

The following table describes the properties of the `@WebMethod` annotation.

Property	Description
operationName	Specifies the value of the associated `wsdl:operation` element's name. The default value is the name of the method.
action	Specifies the value of the `soapAction` attribute of the `soap:operation` element generated for the method. The default value is an empty string.
exclude	Specifies if the method should be excluded from the service interface. The default is `false` .

The @RequestWrapper annotation

The `@RequestWrapper` annotation is defined by the `javax.xml.ws.RequestWrapper` interface. It is placed on the methods in the SEI. As the name implies, `@RequestWrapper` specifies the Java class that implements the wrapper bean for the method parameters that are included in the request message sent in a remote invocation. It is also used to specify the element names, and namespaces, used by the runtime when marshalling and unmarshalling the request messages.

The following table describes the properties of the @RequestWrapper annotation.

Property	Description
localName	Specifies the local name of the wrapper element in the XML representation of the request message. The default value is the name of the method or the value of the `@WebMethod` annotation's `operationName` property.
targetNamespace	Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.
className	Specifies the full name of the Java class that implements the wrapper element.

Tip

Only the `className` property is required.

The @ResponseWrapper annotation

The `@ResponseWrapper` annotation is defined by the `javax.xml.ws.ResponseWrapper` interface. It is placed on the methods in the SEI. As the name implies, `@ResponseWrapper` specifies the Java class that implements the wrapper bean for the method parameters that are included in the response message sent in a remote invocation. It is also used to specify the element names, and namespaces, used by the runtime when marshalling and unmarshalling the response messages.

The following table describes the properties of the `@ResponseWrapper` annotation.

Property	Description
localName	Specifies the local name of the wrapper element in the XML representation of the response message. The default value is the name of the method with `Response` appended or the value of the `@WebMethod` annotation's `operationName` property with `Response` appended.
targetNamespace	Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.
className	Specifies the full name of the Java class that implements the wrapper element.

Tip

Only the `className` property is required.

The @WebFault annotation

The `@WebFault` annotation is defined by the `javax.xml.ws.WebFault` interface. It is placed on exceptions that are thrown by your SEI. The `@WebFault` annotation is used to map the Java exception to a `wsdl:fault` element. This information is used to marshall the exceptions into a representation that can be processed by both the service and its consumers.

The following table describes the properties of the `@WebFault` annotation.

Property	Description
name	Specifies the local name of the fault element.
targetNamespace	Specifies the namespace under which the fault element is defined. The default value is the target namespace of the SEI.
faultName	Specifies the full name of the Java class that implements the exception.

Important

The `name` property is required.

The @Oneway annotation

The `@Oneway` annotation is defined by the `javax.jws.Oneway` interface. It is placed on the methods in the SEI that will not require a response from the service. The `@Oneway` annotation tells the run time that it can optimize the execution of the method by not waiting for a response and not reserving any resources to process a response.

Example

The next example shows an SEI whose methods are annotated.

Example 2.7. SEI with Annotated Methods

```
package org.apache.cxf;

import javax.jws.*;
import javax.xml.ws.*;

@WebService(name="quoteReporter")
public interface QuoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.mycompany.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.mycompany.com/types",
                   className="org.eric.demo.Quote")
  public Quote getQuote(String ticker);
}
```

Defining Parameter Properties with Annotations

The method parameters in the SEI coresspond to the `wsdl:message` elements and their `wsdl:part` elements. JAX-WS provides annotations that allow you to describe the `wsdl:part` elements that are generated for the method parameters.

The @WebParam annotation

The `@WebParam` annotation is defined by the `javax.jws.WebParam` interface. It is placed on the parameters on the methods defined in the SEI. The `@WebParam` annotation allows you to specify the direction of the parameter, if the parameter will be placed in the SOAP header, and other properties of the generated `wsdl:part` .

The following table describes the properties of the `@WebParam` annotation.

Property	**Description**
name	Specifies the name of the parameter as it appears in the WSDL. For RPC bindings, this is name of the `wsdl:part` representing the parameter. For document bindings, this is the local name of the XML element representing the parameter. Per the JAX-WS specification, the default is *argN* , where *N* is replaced with the zero-based argument index (i.e., *arg0* , *arg1* , etc.)
targetNamespace	Specifies the namespace for the parameter. It is only used with document bindings where the parameter maps to an XML element. The defaults is to use the service's namespace.
mode	Specifies the direction of the parameter: Mode.IN (default), Mode.OUT, Mode.INOUT
header	Specifies if the parameter is passed as part of the SOAP header. Values of true or false (default).
partName	Specifies the value of the name attribute of the `wsdl:part` element for the parameter when the binding is document.

The @WebResult annotation

The `@WebResult` annotation is defined by the `javax.jws.WebResult` interface. It is placed on the methods defined in the SEI. The `@WebResult` annotation allows you to specify the properties of the generated `wsdl:part` that is generated for the method's return value.

The following table describes the properties of the `@WebResult` annotation.

Property	Description
name	Specifies the name of the return value as it appears in the WSDL. For RPC bindings, this is name of the `wsdl:part` representing the return value. For document bindings, this is the local name of the XML element representing the return value. The default value is return.
targetNamespace	Specifies the namespace for the return value. It is only used with document bindings where the return value maps to an XML element. The defaults is to use the service's namespace.
header	Specifies if the return value is passed as part of the SOAP header.
partName	Specifies the value of the name attribute of the `wsdl:part` element for the return value when the binding is document.

Example

This example shows an SEI that is fully annotated.

Example 2.8. Fully Annotated SEI

```
package org.apache.cxf;

import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;

@WebService(name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface QuoteReporter
{
   @WebMethod(operationName="getStockQuote")
   @RequestWrapper(targetNamespace="http://demo.mycompany.com/types",
                  className="java.lang.String")
   @ResponseWrapper(targetNamespace="http://demo.mycompany.com/types",
                   className="org.eric.demo.Quote")
   @WebResult(targetNamespace="http://demo.mycompany.com/types",
             name="updatedQuote")
   public Quote getQuote(
      @WebParam(targetNamespace="http://demo.mycompany.com/types",
      name="stockTicker", mode=Mode.IN)
      String ticker
   );
}
```

Generating WSDL

Once you have annotated your code, you can generate a WSDL contract for your service using the **java2wsdl** command.

Generated WSDL from an SEI

The below example shows the WSDL contract generated for the SEI shown above.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://demo.eric.org/"
                  xmlns:tns="http://demo.eric.org/"
          xmlns:ns1=""
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:ns2="http://demo.eric.org/types"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema>
      <xs:complexType name="quote">
        <xs:sequence>
          <xs:element name="ID" type="xs:string" minOccurs="0"/>
          <xs:element name="time" type="xs:string" minOccurs="0"/>
          <xs:element name="val" type="xs:float"/>
        </xs:sequence>
      </xs:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getStockQuote">
    <wsdl:part name="stockTicker" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getStockQuoteResponse">
    <wsdl:part name="updatedQuote" type="tns:quote">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="quoteReporter">
    <wsdl:operation name="getStockQuote">
      <wsdl:input name="getQuote" message="tns:getStockQuote">
    </wsdl:input>
      <wsdl:output name="getQuoteResponse"
        message="tns:getStockQuoteResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="quoteReporterBinding" type="tns:quoteReporter">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getStockQuote">
      <soap:operation style="rpc"/>
      <wsdl:input name="getQuote">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="getQuoteResponse">
        <soap:body use="literal"/>
```

```
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="quoteReporterService">
    <wsdl:port name="quoteReporterPort"
       binding="tns:quoteReporterBinding">
      <soap:address location=
         "http://localhost:9000/quoteReporterService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

2.2.4. JAX-WS Configuration

The following sections list JAX-WS specific configuration items.

2.2.4.1. Configuring an Endpoint

A JAX-WS Endpoint can be configured in XML in addition to using the JAX-WS APIs. Once you've created your server implementation , you simply need to provide the class name and an address. Here is a simple example:

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

<jaxws:endpoint id="classImpl"
    implementor="org.apache.cxf.jaxws.service.Hello"
    endpointName="e:HelloEndpointCustomized"
    serviceName="s:HelloServiceCustomized"
    address="http://localhost:8080/test"
    xmlns:e="http://service.jaxws.cxf.apache.org/endpoint"
    xmlns:s="http://service.jaxws.cxf.apache.org/service"/>

</beans>
```

Be sure to include the JAX-WS `schemaLocation` attribute specified on the root beans element. This allows CXF to validate the file and is required. Also note the namespace declarations at the end of the <jaxws:endpoint/> tag--these are required because the combined "{namespace}localName" syntax is presently not supported for this tag's attribute values.

The `jaxws:endpoint` element (which appears to create an EndpointImpl under the covers) supports many additional attributes:

Name	Value
endpointName	The endpoint name this service is implementing, it maps to the wsdl:port@name. In the format of "ns:ENDPOINT_NAME" where ns is a namespace prefix valid at this scope.

Name	Value
publish	Whether the endpoint should be published now, or whether it will be published at a later point.
serviceName	The service name this service is implementing, it maps to the wsdl:service@name. In the format of "ns:SERVICE_NAME" where ns is a namespace prefix valid at this scope.
wsdlLocation	The location of the WSDL. Can be on the classpath, file system, or be hosted remotely.
bindingUri	The URI, or ID, of the message binding for the endpoint to use. For SOAP the binding URI(ID) is specified by the JAX-WS specification. For other message bindings the URI is the namespace of the WSDL extensions used to specify the binding.
address	The service publish address
bus	The bus name that will be used in the jaxws endpoint.
implementor	The implementor of jaxws endpoint. You can specify the implementor class name here, or just the ref bean name in the format of "#REF_BEAN_NAME"
implementorClass	The implementor class name, it is really useful when you specify the implementor with the ref bean which is wrapped by using Spring AOP
createdFromAPI	This indicates that the endpoint bean was already created using jaxws API's thus at runtime when parsing the bean spring can use these values rather than the default ones. It's important that when this is true, the "name" of the bean is set to the port name of the endpoint being created in the form "{http://service.target.namespace} PortName".
publishedEndpointUrl	The URL that is placed in the address element of the wsdl when the wsdl is retrieved. If not specified, the address listed above is used. This parameter allows setting the "public" URL that may not be the same as the URL the service is deployed on. (for example, the service is behind a proxy of some sort).

It also supports many child elements:

Name	Value
jaxws:executor	A Java executor which will be used for the service. This can be supplied using the Spring <bean class="MyExecutor"/> syntax.
jaxws:inInterceptors	The incoming interceptors for this endpoint. A list of <bean>s or <ref>s. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of <bean>s or <ref>s. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:outInterceptors	The outgoing interceptors for this endpoint. A list of <bean>s or <ref>s. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of <bean>s or <ref>s. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:handlers	The JAX-WS handlers for this endpoint. A list of <bean>s or <ref>s. Each should implement javax.xml.ws.handler.Handler or javax.xml.ws.handler.soap.SOAPHandler (Note that `@HandlerChain` annotations on the service bean appear to be ignored)

Name	Value
jaxws:properties	A properties map which should be supplied to the JAX-WS endpoint. See below.
jaxws:dataBinding	You can specify the which DataBinding will be use in the endpoint , This can be supplied using the Spring <bean class="MyDataBinding"/> syntax.
jaxws:binding	You can specify the BindingFactory for this endpoint to use. This can be supplied using the Spring <bean class="MyBindingFactory"/> syntax.
jaxws:features	The features that hold the interceptors for this endpoint. A list of <bean>s or <ref>s
jaxws:invoker	The invoker which will be supplied to this endpoint. This can be supplied using the Spring <bean class="MyInvoker"/> syntax.
jaxws:schemaLocations	The schema locations for endpoint to use. A list of <schemaLocation>s
jaxws:serviceFactory	The service factory for this endpoint to use. This can be supplied using the Spring <bean class="MyServiceFactory"/> syntax

Here is a more advanced example which shows how to provide interceptors and properties:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
        xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://cxf.apache.org/bindings/soap
       http://cxf.apache.org/schemas/configuration/soap.xsd
       http://cxf.apache.org/jaxws
       http://cxf.apache.org/schemas/jaxws.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>

  <jaxws:endpoint
    id="helloWorld"
    implementor="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <jaxws:inInterceptors>
      <bean class="com.acme.SomeInterceptor"/>
      <ref bean="anotherInterceptor"/>
    </jaxws:inInterceptor>
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>

  <bean id="anotherInterceptor" class="com.acme.SomeInterceptor"/>

  <jaxws:endpoint id="simpleWithBinding"
    implementor="#greeter"
    address="http://localhost:8080/simpleWithAddress">
    <jaxws:binding>
       <soap:soapBinding mtomEnabled="true" version="1.2"/>
    </jaxws:binding>
  </jaxws:endpoint>
```

```
  <jaxws:endpoint id="inlineInvoker"
     address="http://localhost:8080/simpleWithAddress">
    <jaxws:implementor>
      <bean class="org.apache.hello_world_soap_http.GreeterImpl"/>
    </jaxws:implementor>
    <jaxws:invoker>
      <bean class="org.apache.cxf.jaxws.spring.NullInvoker"/>
    </jaxws:invoker>
  </jaxws:endpoint>

</beans>
```

If you are a Spring user, you'll notice that the `jaxws:properties` element follows the Spring Map syntax.

2.2.4.2. Configuring a Spring Client (Option 1)

Important

This technique lets you add a Web Services client to your Spring application. You can inject it into other Spring beans, or manually retrieve it from the Spring context for use by non-Spring-aware client code.

The easiest way to add a Web Services client to a Spring context is to use the `<jaxws:client>` element (similar to the `<jaxws:endpoint>` element used for the server side). Here's a simple example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       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.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="helloClient"
                  serviceClass="demo.spring.HelloWorld"
                  address="http://localhost:9002/HelloWorld" />
</beans>
```

The attributes available on `<jaxws:client>` include:

Name	Type	Description
id	String	A unique identified for the client, which is how other beans in the context will reference it
address	URL	The URL to connect to in order to invoke the service
serviceClass	Class	The fully-qualified name of the interface that the bean should implement (typically, same as the service interface used on the server side)
serviceName	QName	The name of the service to invoke, if this address/WSDL hosts several. It maps to the wsdl:service@name. In the format of "ns:SERVICE_NAME" where ns is a namespace prefix valid at this scope.

Name	Type	Description
endpointName	QName	The name of the endpoint to invoke, if this address/ WSDL hosts several. It maps to the wsdl:port@name. In the format of "ns:ENDPOINT_NAME" where ns is a namespace prefix valid at this scope.
bindingId	URI	The URI, or ID, of the message binding for the endpoint to use. For SOAP the binding URI(ID) is specified by the JAX-WS specification. For other message bindings the URI is the namespace of the WSDL extensions used to specify the binding.
bus	Bean Reference	The bus name that will be used in the jaxws endpoint (defaults to `cxf`).
username	String	
password	String	
wsdlLocation	URL	A URL to connect to in order to retrieve the WSDL for the service. This is not required.
createdFromAPI	boolean	This indicates that the client bean was already created using jaxws API's thus at runtime when parsing the bean spring can use these values rather than the default ones. It's important that when this is true, the "name" of the bean is set to the port name of the endpoint being created in the form "{http://service.target.namespace} PortName".

It also supports many child elements:

Name	Description
jaxws:inInterceptors	The incoming interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:outInterceptors	The outgoing interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
jaxws:features	The features that hold the interceptors for this endpoint. A list of `<bean>` or `<ref>` elements
jaxws:handlers	The JAX-WS handlers for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement javax.xml.ws.handler.Handler or javax.xml.ws.handler.soap.SOAPHandler . These are more portable than CXF interceptors, but may cause the full message to be loaded in as a DOM (slower for large messages).
jaxws:properties	A properties map which should be supplied to the JAX-WS endpoint. See below.
jaxws:dataBinding	You can specify the which DataBinding will be use in the endpoint , This can be supplied using the Spring <bean class="MyDataBinding"/> syntax.

Name	Description
jaxws:binding	You can specify the BindingFactory for this endpoint to use. This can be supplied using the Spring <bean class="MyBindingFactory"/> syntax.
jaxws:conduitSelector	

Here is a more advanced example which shows how to provide interceptors, JAX-WS handlers, and properties:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       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.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">

  <!-- Interceptors extend
          e.g. org.apache.cxf.phase.AbstractPhaseInterceptor -->
  <bean id="anotherInterceptor" class="..." />

  <!-- Handlers implement
          e.g. javax.xml.ws.handler.soap.SOAPHandler -->
  <bean id="jaxwsHandler" class="..." />

  <!-- The SOAP client bean -->
  <jaxws:client id="helloClient"
                serviceClass="demo.spring.HelloWorld"
                address="http://localhost:9002/HelloWorld">
    <jaxws:inInterceptors>
      <bean class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
      <ref bean="anotherInterceptor"/>
    </jaxws:inInterceptor>
    <jaxws:handlers>
      <ref bean="jaxwsHandler" />
    </jaxws:handlers>
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:client>
</beans>
```

2.2.4.3. Configuring a Spring Client (Option 2)

Important

Building a Client using this configuration is only applicable for those wishing to inject a Client into their Spring ApplicationContext.

This approach requires more explicit Spring bean configuration than the previous option, and may require more configuration data depending on which features are used. To configure a client this way, you'll need to declare a proxy factory bean and also a client bean which is created by that proxy factory. Here is an example:

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

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

  <bean id="proxyFactory"
     class="org.apache.cxf.jaxws.JaxWsProxyFactoryBean">
    <property name="serviceClass" value="demo.spring.HelloWorld"/>
    <property name="address" value="http://localhost:9002/HelloWorld"/>
  </bean>

  <bean id="client" class="demo.spring.HelloWorld"
    factory-bean="proxyFactory" factory-method="create"/>

</beans>
```

The JaxWsProxyFactoryBean in this case takes two properties. The service class, which is the interface of the Client proxy you wish to create. The address is the address of the service you wish to call.

The second bean definition is for the client. In this case it implements the HelloWorld interface and is created by the proxyFactory <bean> by calling the create() method. You can then reference this "client" bean and inject it anywhere into your application. Here is an example of a very simple Java class which accesses the client bean:

```
include org.springframework.context.support.ClassPathXmlApplicationContext;

public final class HelloWorldClient {

     private HelloWorldClient() { }

     public static void main(String args[]) throws Exception {
         ClassPathXmlApplicationContext context =
             new ClassPathXmlApplicationContext(
             new String[]{"my/path/to/client-beans.xml"});

         HelloWorld client = (HelloWorld)context.getBean("client");

         String response = client.sayHi("Dan");
         System.out.println("Response: " + response);
         System.exit(0);
     }
}
```

The JaxWsProxyFactoryBean supports many other properties:

Name	Description
clientFactoryBean	The ClientFactoryBean used in construction of this proxy.
password	The password which the transport should use.
username	The username which the transport should use.
wsdlURL	The wsdl URL the client should use to configure itself.
wsdlLocation	Appears to be the same as wsdlURL?
serviceName	The name of the service to invoke, if this address/WSDL hosts several. It maps to the wsdl:service@name. In the format of "ns:SERVICE_NAME" where ns is a namespace prefix valid at this scope.

Name	Description
endpointName	The name of the endpoint to invoke, if this address/WSDL hosts several. It maps to the wsdl:port@name. In the format of "ns:ENDPOINT_NAME" where ns is a namespace prefix valid at this scope.
inInterceptors	The incoming interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
outInterceptors	The outgoing interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of `<bean>` or `<ref>` elements. Each should implement org.apache.cxf.interceptor.Interceptor or org.apache.cxf.phase.PhaseInterceptor
features	The features that hold the interceptors for this endpoint. A list of `<bean>` or `<ref>` elements
handlers	A list of `<bean>` or `<ref>` elements pointing to JAX-WS handler classes to be used for this client. Each should implement javax.xml.ws.handler.Handler or javax.xml.ws.handler.soap.SOAPHandler . These are more portable than CXF interceptors, but may cause the full message to be loaded in as a DOM (slower for large messages).
bindingConfig	
bindingId	The URI, or ID, of the message binding for the endpoint to use. For SOAP the binding URI(ID) is specified by the JAX-WS specification. For other message bindings the URI is the namespace of the WSDL extensions used to specify the binding.
bus	A reference to a CXF bus bean. Must be provided if, for example, handlers are used. May require additional Spring context imports (e.g. to bring in the default CXF bus bean).
conduitSelector	
dataBinding	You can specify the which DataBinding will be use in the endpoint , This can be supplied using the Spring <bean class="MyDataBinding"/> syntax.
properties	A properties map which should be supplied to the JAX-WS endpoint.
serviceFactory	

Using some of the properties will require additional configuration in the Spring context. For instance, using JAX-WS handlers requires that you explicitly import several CXF Spring configurations, and assign the "bus" property of the JaxWsProxyFactory bean like this:

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-http.xml" />

<bean id="clientFactory"
    class="org.apache.cxf.jaxws.JaxWsProxyFactoryBean">
    <property name="serviceClass" value="demo.spring.HelloWorld"/>
    <property name="address" value="http://localhost:9002/HelloWorld"/>
    <property name="bus" ref="cxf" />
</bean>
```

2.2.4.4. Configuring an Endpoint/Client Proxy Using CXF APIs

JAX-WS endpoints and client proxies are implemented on top of CXF's frontend-neutral endpoint API. You can therefore use CXF APIs to enhance the functionality of a JAX-WS endpoint or client proxy, for example by adding interceptors.

To cast a client proxy to a CXF client:

```
GreeterService gs = new GreeterService();
Greeter greeter = gs.getGreeterPort();

org.apache.cxf.endpoint.Client client =
org.apache.cxf.frontend.ClientProxy.getClient(greeter);
org.apache.cxf.endpoint.Endpoint cxfEndpoint = client.getEndpoint();
cxfEndpoint.getOutInterceptors().add(...);
```

To cast a JAX-WS endpoint to a CXF server:

```
javax.xml.ws.Endpoint jaxwsEndpoint =
            javax.xml.ws.Endpoint.publish(
            "http://localhost:9020/SoapContext/GreeterPort",
            new GreeterImpl());
org.apache.cxf.jaxws.EndpointImpl jaxwsEndpointImpl =
(org.apache.cxf.jaxws.EndpointImpl)jaxwsEndpoint;
org.apache.cxf.endpoint.Server server = jaxwsEndpointImpl.getServer();
org.apache.cxf.endpoint.Endpoint cxfEndpoint = server.getEndpoint();
cxfEndpoint.getOutInterceptors().add(...);
org.apache.cxf.service.Service cxfService = cxfEndpoint.getService();
cxfService.getOutInterceptors().add(...);
```

2.2.4.5. Configure the JAXWS Server/Client Using Spring

CXF provides <jaxws:server>, <jaxws:client> to configure the server/client side endpoint. Here are some exmples:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xmlns:soap="http://cxf.apache.org/bindings/soap"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://cxf.apache.org/bindings/soap
      http://cxf.apache.org/schemas/configuration/soap.xsd
      http://cxf.apache.org/jaxws
      http://cxf.apache.org/schemas/jaxws.xsd">
 <jaxws:server id="inlineImplementor"
    address="http://localhost:8080/simpleWithAddress">
    <jaxws:serviceBean>
      <bean class="org.apache.hello_world_soap_http.GreeterImpl"/>
    </jaxws:serviceBean>
  </jaxws:server>

  <jaxws:server id="bookServer"
    serviceClass="org.myorg.mytype.AnonymousComplexTypeImpl"
    address="http://localhost:8080/act"
```

```
    bus="cxf">
    <jaxws:invoker>
      <bean class="org.myorg.service.invoker.BeanInvoker">
        <constructor-arg>
          <bean class="org.myorg.mytype.AnonymousComplexTypeImpl"/>
        </constructor-arg>
      </bean>
    </jaxws:invoker>
     <jaxws:dataBinding>
      <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
        <property name="namespaceMap">
            <map>
            <entry>
                <key>
                   <value>
                      http://cxf.apache.org/anon_complex_type/
                   </value>
                </key>
                <value>BeepBeep</value>
            </entry>
            </map>
        </property>
      </bean>
      </jaxws:dataBinding>
  </jaxws:server>

  <jaxws:client id="bookClient"
    serviceClass="org.myorg.mytype.AnonymousComplexType"
    address="http://localhost:8080/act"/>

</beans>
```

2.2.5. JAX-WS Providers

JAX-WS Providers allow you to create services which work at the message level - as opposed to the operation level as with annotated classes. The have a single operation "invoke" which receives either the message payload (i.e. the SOAP Body) or the whole message itself (i.e. the SOAP Envelope).

Here's a simple example:

```
@WebServiceProvider
public class HelloProvider {
  public Source invoke(Source request) {
    return ....;
  }
}
```

Services are publish via one of two means:

- The JAX-WS standard Endpoint APIs
- CXF's XML configuration format - i.e. <jaxws:endpoint ... />

2.2.5.1. Messaging Modes

Overview

Objects that implement the `Provider` interface have two *messaging modes* :

- Message mode
- Payload mode

The messaging mode you specify determines the level of messaging detail that is passed to your implementation.

Message mode

When using *message mode* , a `Provider` implementation works with complete messages. A complete message includes any binding specific headers and wrappers. For example, a `Provider` implementation that uses a SOAP binding would receive requests as fully specified SOAP message. Any response returned from the implementation would also need to be a fully specified SOAP message.

You specify that a `Provider` implementation uses message mode by providing the value `java.xml.ws.Service.Mode.MESSAGE` as the value to the `javax.xml.ws.ServiceMode` annotation.

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class stockQuoteProvider implements Provider<SOAPMessage>
{
  ...
}
```

Payload mode

In *payload mode* a `Provider` implementation works with only the payload of a message. For example, a `Provider` implementation working in payload mode works only with the body of a SOAP message. The binding layer processes any binding level wrappers and headers.

Tip

When working with a binding that does not use special wrappers, such as the XML binding, payload mode and message mode provide the same results.

You specify that a `Provider` implementation uses payload mode by providing the value `java.xml.ws.Service.Mode.PAYLOAD` as the value to the `javax.xml.ws.ServiceMode` annotation.

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class stockQuoteProvider implements Provider<DOMSource>
{
  ...
}
```

Tip

If you do not provide the `@ServiceMode` annotation, the `Provider` implementation will default to using payload mode.

2.2.5.2. Data Types

Overview

`Provider` implementations, because they are low-level objects, cannot use the same JAXB generated types as the higher level consumer APIs. Provider implementations work with the following types of objects:

- `javax.xml.transform.Source`
- `javax.xml.soap.SOAPMessage`
- `javax.activation.DataSource`

Using Source objects

A `Provider` implementation can accept and return objects that are derived from the `javax.xml.transform.Source` interface. `Source` objects are low level objects that hold XML documents. Each `Source` implementation provides methods that access the stored XML documents and manipulate its contents. The following objects implement the `Source` interface:

- `DOMSource` holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of `Node` objects that can be accessed using the `getNode()` method. Nodes can be updated or added to the DOM tree using the `setNode()` method.
- `SAXSource` holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an `InputSource` object that contains the raw data and an `XMLReader` object that parses the raw data.
- `StreamSource` holds XML messages as a data stream. The data stream can be manipulated as would any other data stream.

Important

When using `Source` objects the developer is responsible for ensuring that all required binding specific wrappers are added to the message. For example, when interacting with a service expecting SOAP messages, the developer must ensure that the required SOAP envelope is added to the outgoing request and that the SOAP envelope's contents are correct.

Using SOAPMessage objects

`Provider` implementations can use `javax.xml.soap.SOAPMessage` objects when the following conditions are true:

- the `Provider` implementation is using the SOAP binding.
- the `Provider` implementation is using message mode.

A `SOAPMessage` object, as the name implies, holds a SOAP message. They contain one `SOAPPart` object and zero or more `AttachmentPart` objects. The `SOAPPart` object contains the SOAP specific portions of the SOAP message including the SOAP envelope, any SOAP headers, and the SOAP message body. The `AttachmentPart` objects contain binary data that was passed as an attachment.

Using DataSource objects

Provider implementations can use objects that implement the `javax.activation.DataSource` interface when the following conditions are true:

- the implementation is using the HTTP binding.
- the implementation is using message mode.

`DataSource` objects provide a mechanism for working with MIME typed data from a variety of sources including URLs, files, and byte arrays.

Implementing a Provider Object

Overview

The `Provider` interface is relatively easy to implement. It only has one method, invoke(), that needs to be implemented. In addition it has three simple requirements:

- An implementation must have the `@WebServiceProvider` annotation.
- An implementation must have a default public constructor.
- An implementation must implement a typed version of the Provider interface. In other words, you cannot implement a `Provider<T>` interface. You must implement a version of the interface that uses a concrete data type. For example, you can implement an instance of a `Provider<SAXSource>` .

The complexity of implementing the Provider interface surrounds handling the request messages and building the proper responses.

Working with messages

Unlike the higher-level SEI based service implementations, `Provider` implementations receive requests as raw XML data and must send responses as raw XML data. This requires that the developer has intimate knowledge of the messages used by the service being implemented. These details can typically be found in the WSDL document describing the service.

WS-I Basic Profile provides guidelines about the messages used by services including:

- The root element of a request is based in the value of the name attribute of the `wsdl:operation` element that corresponds to the operation being invoked.

Warning

If the service uses doc/literal bare messages, the root element of the request will be based on the value of name attribute of the `wsdl:part` element referred to by the `wsdl:operation` element.

- The root element of all messages will be namespace qualified.
- If the service uses rpc/literal messages, the top-level elements in the messages will not be namespace qualified.

Important

> The children of top-level elements may be namespace qualified. To be certain you will need to check their schema definitions.

- If the service uses rpc/literal messages, none of the top-level elements can be null.
- If the service uses doc/literal messages, the schema definition of the message determines if any of the elements are namespace qualified.

Implementing the invoke() method

The `Provider` interface has only one method, `invoke()` , that needs to be implemented. `invoke()` receives the incoming request packaged into the type of object declared by the type of `Provider` interface being implemented and returns the response message packaged into the same type of object. For example, an implementation of a `Provider<SOAPMessage>` interface would receive the request as a `SOAPMessage` object and return the response as a `SOAPMessage` object.

The messaging mode used by the `Provider` implementation determines the amount of binding specific information the request and response messages contain. Implementation using message mode receive all of the binding specific wrappers and headers along with the request. They must also add all of the binding specific wrappers and headers to the response message. Implementations using payload mode only receive the body of the request. The XML document returned by an implementation using payload mode will be placed into the body of the request message.

Examples

The following shows a `Provider` implementation that works with `SOAPMessage` objects in message mode.

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
   serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.MESSAGE")
public class  stockQuoteReporterProvider implements Provider<SOAPMessage>
{
public stockQuoteReporterProvider() {}

public SOAPMessage invoke(SOAPMessage request)
  {
  SOAPBody requestBody = request.getSOAPBody();
  if(requestBody.getElementName.getLocalName.equals("getStockPrice"))
    {
    MessageFactory mf = MessageFactory.newInstance();
    SOAPFactory sf = SOAPFactory.newInstance();

    SOAPMessage response = mf.createMessage();
    SOAPBody respBody = response.getSOAPBody();
```

```
    Name bodyName = sf.createName("getStockPriceResponse");
    respBody.addBodyElement(bodyName);
    SOAPElement respContent = respBody.addChildElement("price");
    respContent.setValue("123.00");
    response.saveChanges();
    return response;
    }
    ...
  }
}
```

The code does the following:

1. Specifies that the following class implements a `Provider` object that implements the service whose `wsdl:service` element is named `stockQuoteReporter` and whose `wsdl:port` element is named `stockQuoteReporterPort` .
2. Specifies that this `Provider` implementation uses message mode.
3. Provides the required default public constructor.
4. Provides an implementation of the `invoke()` method that takes a `SOAPMessage` object and returns a `SOAPMessage` object.
5. Extracts the request message from the body of the incoming SOAP message.
6. Checks the root element of the request message to determine how to process the request.
7. Creates the factories needed for building the response.
8. Builds the SOAP message for the response.
9. Returns the response as a SOAPMessage object.

The following shows an example of a Provider implementation using DOMSource objects in payload mode.

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
   serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.PAYLOAD")
public class  stockQuoteReporterProvider implements Provider<DOMSource>
public stockQuoteReporterProvider()
  {
  }

public DOMSource invoke(DOMSource request)
  {
    DOMSource response = new DOMSource();
    ...
    return response;
  }
}
```

The code does the following:

1. Specifies that the class implements a `Provider` object that implements the service whose `wsdl:service` element is named `stockQuoteReporter` and whose `wsdl:port` element is named `stockQuoteReporterPort` .

2. Specifies that this Provider implementation uses payload mode.
3. Provides the required default public constructor.
4. Provides an implementation of the `invoke()` method that takes a `DOMSource` object and returns a `DOMSource` object.

2.2.6. WebserviceContext

The WebserviceContext interface is part of the JAX-WS specification. It allows you to access several context informations the runtime has associated to your service call.

The following code fragment show how to use some parts of the WebserviceContext.

```
public class CustomerServiceImpl implements CustomerService {
@Resource
WebServiceContext wsContext;

public List<Customer> getCustomersByName(String name)
   throws NoSuchCustomerException {
  Principal pr = wsContext.getUserPrincipal();

  // Only joe may access this service operation
  if (pr == null || !"joe".equals(pr.getName())) {
    throw new RuntimeException("Access denied");
  }

  // Only the sales role may access this operation
  if (!wsContext.isUserInRole("sales")) {
    throw new RuntimeException("Access denied");
  }

  MessageContext mContext = wsContext.getMessageContext();

  // See which contents the message context has
  Set<String> s = mContext.keySet();

  // Using this cxf specific code you can access
  // the CXF Message and Exchange objects
  WrappedMessageContext wmc = (WrappedMessageContext)mContext;
  Message m = wmc.getWrappedMessage();
  Exchange ex = m.getExchange();
}
}
```

2.3. JAX-WS Client Development Options

2.3.1. WSDL2Java generated Client

One of the most common scenarios is that where you have a service which you may or not manage and this service has a WSDL. In this case you'll often want to generate a client from the WSDL. This provides you with a strongly

typed interface by which to interact with the service. Once you've generated a client, typical usage of it will look like so:

```
HelloService service = new HelloService();
Hello client = service.getHelloHttpPort();

String result = client.sayHi("Joe");
```

The WSDL2Java tool will generate JAX-WS clients from your WSDL. You can run WSDL2java one of three ways:

- The command line
- The Maven Plugin
- With the WSDL2Java API

For more in depth information read Developing a JAX-WS consumer or see the Hello World demos inside the distribution.

2.3.2. JAX-WS Proxy

Instead of using a wsdl2java-generated stub client directly, you can use Service.create to create Service instances, the following code illustrates this process:

```
import java.net.URL;
import javax.xml.ws.Service;
...

URL wsdlURL = new URL("http://localhost/hello?wsdl");
QName SERVICE_NAME = new QName("http://apache.org/hello_world_soap_http",
   "SOAPService");
Service service = Service.create(wsdlURL, SERVICE_NAME);
Greeter client = service.getPort(Greeter.class);
String result = client.greetMe("test");
```

2.3.3. JAX-WS Dispatch APIs

JAX-WS provides the "dispatch" mechanism which makes it easy to dynamically invoke services which you have not generated a client for. Using the Dispatch mechanism you can create messages (which can be JAXB objects, Source objects, or a SAAJMessage) and dispatch them to the server. A simple example might look like this:

```
import java.net.URL;
import javax.xml.transform.Source;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
...

URL wsdlURL = new URL("http://localhost/hello?wsdl");
Service service = Service.create(wsdlURL, new QName("HelloService"));
Dispatch<Source> disp = service.createDispatch(new QName("HelloPort"),
   Source.class, Service.Mode.PAYLOAD);

Source request = new StreamSource("<hello/>")
Source response = disp.invoke(request);
```

NOTE: you can also use dispatches without a WSDL.

For more in depth information see the Hello World demos inside the distribution.

2.3.4. Usage Modes

2.3.4.1. Overview

`Dispatch` objects have two *usage modes* :

- Message mode
- Message Payload mode (Payload mode)

The usage mode you specify for a `Dispatch` object determines the amount of detail is passed to the user level code.

2.3.4.2. Message mode

In *message mode* , a `Dispatch` object works with complete messages. A complete message includes any binding specific headers and wrappers. For example, a consumer interacting with a service that requires SOAP messages would need to provide the `Dispatch` object's `invoke()` method a fully specified SOAP message. The `invoke()` method will also return a fully specified SOAP message. The consumer code is responsible for completing and reading the SOAP message's headers and the SOAP message's envelope information.

Tip

Message mode is not ideal when you wish to work with JAXB objects.

You specify that a `Dispatch` object uses message mode by providing the value `java.xml.ws.Service.Mode.MESSAGE` when creating the Dispatch object.

2.3.4.3. Payload mode

In *payload mode* , also called message payload mode, a `Dispatch` object works with only the payload of a message. For example, a `Dispatch` object working in payload mode works only with the body of a SOAP message. The binding layer processes any binding level wrappers and headers. When a result is returned from `invoke()` the binding level wrappers and headers are already striped away and only the body of the message is left.

Tip

When working with a binding that does not use special wrappers, such as the Artix ESB XML binding, payload mode and message mode provide the same results.

You specify that a `Dispatch` object uses payload mode by providing the value `java.xml.ws.Service.Mode.PAYLOAD` when creating the `Dispatch` object.

2.3.5. Data Types

2.3.5.1. Overview

`Dispatch` objects, because they are low-level objects, are not optimized for using the same JAXB generated types as the higher level consumer APIs. `Dispatch` objects work with the following types of objects:

- `javax.xml.transform.Source`
- `javax.xml.soap.SOAPMessage`
- `javax.activation.DataSource`
- JAXB

2.3.5.2. Using Source objects

A `Dispatch` object can accept and return objects that are derived from the `javax.xml.transform.Source` interface. Source objects are low level objects that hold XML documents. Each `Source` implementation provides methods that access the stored XML documents and manipulate its contents. The following objects implement the `Source` interface:

- `DOMSource`
- `SAXSource`
- `StreamSource`

Important

When using `Source` objects the developer is responsible for ensuring that all required binding specific wrappers are added to the message. For example, when interacting with a service expecting SOAP messages, the developer must ensure that the required SOAP envelope is added to the outgoing request and that the SOAP envelope's contents are correct.

2.3.5.3. Using SOAPMessage objects

`Dispatch` objects can use `javax.xml.soap.SOAPMessage` objects when the following conditions are true:

- the `Dispatch` object is using the SOAP binding.
- the `Dispatch` object is using message mode.

2.3.5.4. Using DataSource objects

`Dispatch` objects can use objects that implement the `javax.activation.DataSource` interface when the following conditions are true:

- the `Dispatch` object is using the HTTP binding.

- the `Dispatch` object is using message mode.

`DataSource` objects provide a mechanism for working with MIME typed data from a variety of sources including URLs, files, and byte arrays.

2.3.5.5. Using JAXB objects

While `Dispatch` objects are intended to be low level API that allows you to work with raw messages, they also allow you to work with JAXB objects. To work with JAXB objects a `Dispatch` object must be passed a `JAXBContext` that knows how to marshal and unmarshal the JAXB objects in use. The `JAXBContext` is passed when the `Dispatch` object is created.

You can pass any JAXB object understood by the `JAXBContext` object as the parameter to the `invoke()` method. You can also cast the returned message into any JAXB object understood by the `JAXBContext` object.

2.3.6. Working with Dispatch Objects

2.3.6.1. Procedure

To use a Dispatch object to invoke a remote service you do the following:

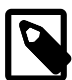
1. Create a `Dispatch` object.
2. Construct a request message.
3. Call the proper `invoke()` method.
4. Parse the response message.

2.3.6.2. Creating a Dispatch object

To create a `Dispatch` object do the following:

1. Create a `Service` object to represent the `wsdl:service` element defining the service on which the `Dispatch` object will make invocations.
2. Create the `Dispatch` object using the `Service` object's `createDispatch()` method.

```
public Dispatch<T> createDispatch(QName portName,
java.lang.Class<T> type, Service.Mode mode) throws WebServiceException;
```



Note

If you are using JAXB objects the method signature for `createDispatch()` is:

```
public Dispatch<T> createDispatch(QName portName,
javax.xml.bind.JAXBContext context, Service.Mode mode)
throws WebServiceException;
```

The following table describes the parameters for `createDispatch()`.

Parameter	Description
portName	Specifies the QName of the `wsdl:port` element that represent the service provider on which the `Dispatch` object will make invocations.
type	Specifies the data type of the objects used by the `Dispatch` object.
mode	Specifies the usage mode for the `Dispatch` object.

The code bellow creates a `Dispatch` object that works with `DOMSource` objects in payload mode.

```
package com.mycompany.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    QName serviceName = new QName("http://org.apache.cxf",
       "stockQuoteReporter");
    Service s = Service.create(serviceName);

    QName portName = new QName("http://org.apache.cxf",
       "stockQuoteReporterPort");
    Dispatch<DOMSource> dispatch = createDispatch(portName,
                                            DOMSource.class,
                                            Service.Mode.PAYLOAD);
    ...
```

2.3.6.3. Constructing request messages

When working with `Dispatch` objects requests must be built from scratch. The developer is responsible for ensuring that the messages passed to a `Dispatch` object match a request that the targeted service provider can process. This requires precise knowledge about the messages used by the service provider and what, if any, header information it requires.

This information can be provided by a WSDL document or an XMLSchema document that defines the messages. While service providers vary greatly there are a few guidelines that can be followed:

- The root element of the request is based in the value of the name attribute of the `wsdl:operation` element that corresponds to the operation being invoked.

Warning

> If the service being invoked uses doc/literal bare messages, the root element of the request will be based on the value of name attribute of the `wsdl:part` element refered to by the `wsdl:operation` element.

- The root element of the request will be namespace qualified.
- If the service being invoked uses rpc/literal messages, the top-level elements in the request will not be namespace qualified.

Important

The children of top-level elements may be namespace qualified. To be certain you will need to check their schema definitions.

- If the service being invoked uses rpc/literal messages, none of the top-level elements can be null.
- If the service being invoked uses doc/literal messages, the schema definition of the message determines if any of the elements are namespace qualified.

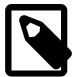
For more information about how services use XML messages see the WS-I Basic Profile.

2.3.6.4. Synchronous invocation

For consumers that make synchronous invocations that generate a response, you use the `Dispatch` object's `invoke()` method shown bellow.

```
T invoke(T msg)
throws WebServiceException;
```

The type of both the response and the request passed to the `invoke()` method are determined when the Dispatch object is created. For example if you created a `Dispatch` object using `createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE)` the response and the request would both be `SOAPMessage` objects.



Note

When using JAXB objects, the response and the request can be of any type the provided `JAXBContext` object can marshal and unmarshal. Also, the response and the request can be different JAXB objects.

The code below makes a synchronous invocation on a remote service using a `DOMSource` object.

```
// Creating a DOMSource Object for the request
DocumentBuilder db = DocumentBuilderFactory.newDocumentBuilder();
Document requestDoc = db.newDocument();
Element root = requestDoc.createElementNS(
   "http://org.apache.cxf/stockExample", "getStockPrice");
root.setNodeValue("DOW");
DOMSource request = new DOMSource(requestDoc);

// Dispatch disp created previously
DOMSource response = disp.invoke(request);
```

Asynchronous invocation

`Dispatch` objects also support asynchronous invocations. As with the higher level asynchronous APIs discussed in Chapter 4, `Dispatch` objects can use both the polling approach and the callback approach.

When using the polling approach the `invokeAsync()` method returns a `Response<t>` object that can be periodically polled to see if the response has arrived.

```
Response <T> invokeAsync(T msg)
throws WebServiceException;
```

When using the callback approach the `invokeAsync()` method takes an `AsyncHandler` implementation that processes the response when it is returned.

```
Future<?> invokeAsync(T msg, AsyncHandler<T> handler)
throws WebServiceException;
```

Note

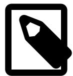
As with the synchronous `invoke()` method, the type of the response and the type of the request are determined when you create the `Dispatch` object.

Oneway invocation

When a request does not generate a response, you make remote invocations using the `Dispatch` object's `invokeOneWay()`.

```
void invokeOneWay(T msg)
throws WebServiceException;
```

The type of object used to package the request is determined when the `Dispatch` object is created. For example if the `Dispatch` object is created using `createDispatch(portName, DOMSource.class, Service.Mode.PAYLOAD)` the request would be packaged into a `DOMSource` object.



Note

When using JAXB objects, the response and the request can be of any type the provided `JAXBContext` object can marshal and unmarshal. Also, the response and the request can be different JAXB objects.

The code below makes a one way invocation on a remote service using a JAXB object.

```
// Creating a JAXBContext and an Unmarshaller for the request
JAXBContext jbc = JAXBContext.newInstance("org.mycompany.StockExample");
Unmarshaller u = jbc.createUnmarshaller();

// Read the request from disk
File rf = new File("request.xml");
GetStockPrice request = (GetStockPrice)u.unmarshal(rf);

// Dispatch disp created previously
disp.invokeOneWay(request);
```

2.3.7. Developing a Consumer

2.3.7.1. Generating the Stub Code

The starting point for developing a service consumer (or client) in CXF is a WSDL contract, complete with port type, binding, and service definitions. You can then use the wsdl2java utility to generate the Java stub code from the WSDL contract. The stub code provides the supporting code that is required to invoke operations on the remote service. For CXF clients, the wsdl2java utility can generate the following kinds of code:

- Stub code - supporting files for implementing a CXF client.

- Client starting point code - sample client code that connects to the remote service and invokes every operation on the remote service.
- Ant build file - a `build.xml` file intended for use with the ant build utility. It has targets for building and for running the sample client application.

Basic HelloWorld WSDL contract

The below shows the HelloWorld WSDL contract. This contract defines a single port type, `Greeter` , with a SOAP binding, `Greeter_SOAPBinding` , and a service, `SOAPService` , which has a single port, `SoapPort` .

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <schema targetNamespace=
            "http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://apache.org/hello_world_soap_http/types"
            elementFormDefault="qualified">
        <simpleType name="MyStringType">
        <restriction base="string">
            <maxLength value="30" />
        </restriction>
        </simpleType>

            <element name="sayHi">
                <complexType/>
            </element>
            <element name="sayHiResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMe">
                <complexType>
                    <sequence>
                        <element name="requestType"
                           type="tns:MyStringType"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
```

```
            </complexType>
        </element>
        <element name="greetMeOneWay">
            <complexType>
                <sequence>
                    <element name="requestType" type="string"/>
                </sequence>
            </complexType>
        </element>
        <element name="pingMe">
            <complexType/>
        </element>
        <element name="pingMeResponse">
            <complexType/>
        </element>
        <element name="faultDetail">
            <complexType>
                <sequence>
                    <element name="minor" type="short"/>
                    <element name="major" type="short"/>
                </sequence>
            </complexType>
        </element>
    </schema>
</wsdl:types>
<wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
</wsdl:message>
<wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeRequest">
    <wsdl:part element="x1:greetMe" name="in"/>
</wsdl:message>
<wsdl:message name="greetMeResponse">
    <wsdl:part element="x1:greetMeResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeOneWayRequest">
    <wsdl:part element="x1:greetMeOneWay" name="in"/>
</wsdl:message>
<wsdl:message name="pingMeRequest">
    <wsdl:part name="in" element="x1:pingMe"/>
</wsdl:message>
<wsdl:message name="pingMeResponse">
    <wsdl:part name="out" element="x1:pingMeResponse"/>
</wsdl:message>
<wsdl:message name="pingMeFault">
    <wsdl:part name="faultDetail" element="x1:faultDetail"/>
</wsdl:message>

<wsdl:portType name="Greeter">
    <wsdl:operation name="sayHi">
        <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
        <wsdl:output message="tns:sayHiResponse"
            name="sayHiResponse"/>
    </wsdl:operation>

    <wsdl:operation name="greetMe">
```

```
            <wsdl:input message="tns:greetMeRequest"
                 name="greetMeRequest"/>
            <wsdl:output message="tns:greetMeResponse"
                 name="greetMeResponse"/>
        </wsdl:operation>

        <wsdl:operation name="greetMeOneWay">
            <wsdl:input message="tns:greetMeOneWayRequest"
                 name="greetMeOneWayRequest"/>
        </wsdl:operation>

        <wsdl:operation name="pingMe">
            <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
            <wsdl:output name="pingMeResponse"
                message="tns:pingMeResponse"/>
            <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="sayHi">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="sayHiRequest">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="sayHiResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="greetMe">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="greetMeRequest">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="greetMeResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="greetMeOneWay">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="greetMeOneWayRequest">
                <soap:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>

        <wsdl:operation name="pingMe">
            <soap:operation style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="pingMeFault">
```

```
                <soap:fault name="pingMeFault" use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

    </wsdl:binding>
    <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
            <soap:address
                location="http://localhost:9000/SoapContext/SoapPort"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

The `Greeter` port type from the example above [42] defines the following WSDL operations:

- `sayHi` - has a single output parameter, of `xsd:string` .
- `greetMe` - has an input parameter, of `xsd:string` , and an output parameter, of `xsd:string` .
- `greetMeOneWay` - has a single input parameter, of `xsd:string` . Because this operation has no output parameters, CXF can optimize this call to be a oneway invocation (that is, the client does not wait for a response from the server).
- `pingMe` - has no input parameters and no output parameters, but it can raise a fault exception.

The above example [42] also defines a binding, `Greeter_SOAPBinding` , for the SOAP protocol. In practice, the binding is normally generated automatically - for example, by running either of the CXF wsdl2soap or wsdl2xml utilities. Likewise, the `SOAPService` service can be generated automatically by running the CXF wsdl2service utility.

Generating the stub code

After defining the WSDL contract, you can generate client code using the CXF wsdl2java utility. Enter the following command at a command-line prompt:

```
wsdl2java -ant -client -d ClientDir hello_world.wsdl
```

Where *ClientDir* is the location of a directory where you would like to put the generated files and `hello_world.wsdl` is a file containing the contract shown in the above WSDL. The `-ant` option generates an ant `build.xml` file, for use with the ant build utility. The `-client` option generates starting point code for a client `main()` method.

The preceding wsdl2java command generates the following Java packages:

- `org.apache.hello_world_soap_http` This package name is generated from the http://apache.org/hello_world_soap_http target namespace. All of the WSDL entities defined in this target namespace (for example, the Greeter port type and the SOAPService service) map to Java classes in the corresponding Java package.
- `org.apache.hello_world_soap_http.types` This package name is generated from the http://apache.org/hello_world_soap_http/types" target namespace. All of the XML types defined in this target namespace (that is, everything defined in the `wsdl:types` element of the HelloWorld contract) map to Java classes in the corresponding Java package.

The stub files generated by the wsdl2java command fall into the following categories:

- Classes representing WSDL entities (in the `org.apache.hello_world_soap_http` package) - the following classes are generated to represent WSDL entities:

  - `Greeter` is a Java interface that represents the Greeter WSDL port type. In JAX-WS terminology, this Java interface is a service endpoint interface.
  - `SOAPService` is a Java class that represents the SOAPService WSDL `service` element.
  - `PingMeFault` is a Java exception class (extending `java.lang.Exception`) that represents the pingMeFault WSDL `fault` element.
- Classes representing XML types (in the `org.apache.hello_world_soap_http.types` package) - in the HelloWorld example, the only generated types are the various wrappers for the request and reply messages. Some of these data types are useful for the asynchronous invocation model.

2.3.7.2. Implementing a CXF Client

This section describes how to write the code for a simple Java client, based on the WSDL contract above. To implement the client, you need to use the following stub classes:

- Service class (that is, `SOAPService`).
- Service endpoint interface (that is, `Greeter`).

Generated service class

The next example shows the typical outline a generated service class, `ServiceName` , which extends the `javax.xml.ws.Service` base class.

Example 2.9. Outline of a Generated Service Class

```
public class ServiceName extends javax.xml.ws.Service
{
  ...
  public ServiceName(URL wsdlLocation, QName serviceName) { }

  public ServiceName() { }

  public Greeter getPortName() { }
  .
  .
  .
}
```

The `ServiceName` class above defines the following methods:

- Constructor methods - the following forms of constructor are defined:
  - `ServiceName(URL wsdlLocation, QName serviceName)` constructs a service object based on the data in the *serviceName* service in the WSDL contract that is obtainable from *wsdlLocation* .
  - `ServiceName()` is the default constructor, which constructs a service object based on the service name and WSDL contract that were provided at the time the stub code was generated (for example, when running the CeltiXfire wsdl2java command). Using this constructor presupposes that the WSDL contract remains available at its original location.
- `get_PortName_()` methods - for every *PortName* port defined on the *ServiceName* service, CXF generates a corresponding `get_PortName_()` method in Java. Therefore, a `wsdl:service` element that defines multiple ports will generate a service class with multiple `get_PortName_()` methods.

Service endpoint interface

For every port type defined in the original WSDL contract, you can generate a corresponding service endpoint interface in Java. A service endpoint interface is the Java mapping of a WSDL port type. Each operation defined in the original WSDL port type maps to a corresponding method in the service endpoint interface. The operation's parameters are mapped as follows:

1. The input parameters are mapped to method arguments.
2. The first output parameter is mapped to a return value.
3. If there is more than one output parameter, the second and subsequent output parameters map to method arguments (moreover, the values of these arguments must be passed using Holder types).

The next example shows the Greeter service endpoint interface, which is generated from the Greeter port type defined in the WSDL above. For simplicity, this example omits the standard JAXB and JAX-WS annotations.

Example 2.10. The Greeter Service Endpoint Interface

```
/* Generated by WSDLToJava Compiler. */

package org.objectweb.hello_world_soap_http;
  ...
public interface Greeter
{
  public java.lang.String sayHi();

  public java.lang.String greetMe(java.lang.String requestType);

  public void greetMeOneWay(java.lang.String requestType);

  public void pingMe() throws PingMeFault;
}
```

Client main function

This example shows the Java code that implements the HelloWorld client. In summary, the client connects to the `SoapPort` port on the `SOAPService` service and then proceeds to invoke each of the operations supported by the `Greeter` port type.

```
package demo.hw.client;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apche.hello_world_soap_http.SOAPService;

public final class Client {

  private static final QName SERVICE_NAME =
    new QName("http://apache.org/hello_world_soap_http",
       "SOAPService");

  private Client()
```

```
  {
  }

  public static void main(String args[]) throws Exception
  {
    if (args.length == 0)
    {
      System.out.println("please specify wsdl");
      System.exit(1);
    }

    URL wsdlURL;
    File wsdlFile = new File(args[0]);
    if (wsdlFile.exists())
    {
      wsdlURL = wsdlFile.toURL();
    }
    else
    {
      wsdlURL = new URL(args[0]);
    }

    System.out.println(wsdlURL);
    SOAPService ss = new SOAPService(wsdlURL, SERVICE_NAME);
    Greeter port = ss.getSoapPort();
    String resp;

    System.out.println("Invoking sayHi...");
    resp = port.sayHi();
    System.out.println("Server responded with: " + resp);
    System.out.println();

    System.out.println("Invoking greetMe...");
    resp = port.greetMe(System.getProperty("user.name"));
    System.out.println("Server responded with: " + resp);
    System.out.println();

    System.out.println("Invoking greetMeOneWay...");
    port.greetMeOneWay(System.getProperty("user.name"));
    System.out.println("No response from server as method is OneWay");
    System.out.println();

    try {
      System.out.println("Invoking pingMe, expecting exception...");
      port.pingMe();
    } catch (PingMeFault ex) {
      System.out.println(
          "Expected exception: PingMeFault has occurred.");
      System.out.println(ex.toString());
    }
    System.exit(0);
  }
}
```

The `Client.main()` function from the above example proceeds as follows:</para>

1. The CXF runtime is implicitly initialized - that is, provided the CXF runtime classes are loaded. Hence, there is no need to call a special function in order to initialize CXF.

2. The client expects a single string argument that gives the location of the WSDL contract for HelloWorld. The WSDL location is stored in `wsdlURL` .

3. A new port object (which enables you to access the remote server endpoint) is created in two steps, as shown in the following code fragment:

   ```
   SOAPService ss = new SOAPService(wsdlURL, SERVICE_NAME);
   Greeter port = ss.getSoapPort();
   ```

   To create a new port object, you first create a service object (passing in the WSDL location and service name) and then call the appropriate `get PortName ()` method to obtain an instance of the particular port you need. In this case, the `SOAPService` service supports only the `SoapPort` port, which is of `Greeter` type.

4. The client proceeds to call each of the methods supported by the `Greeter` service endpoint interface.

5. In the case of the `pingMe()` operation, the example code shows how to catch the `PingMeFault` fault exception.

2.3.7.3. Setting Connection Properties with Contexts

You can use JAX-WS contexts to customize the properties of a client proxy. In particular, contexts can be used to modify connection properties and to send data in protocol headers. For example, you could use contexts to add a SOAP header, either to a request message or to a response message. The following types of context are supported on the client side:

- **Request context** - on the client side, the request context enables you to set properties that affect outbound messages. Request context properties are applied to a specific port instance and, once set, the properties affect every subsequent operation invocation made on the port, until such time as a property is explicitly cleared. For example, you might use a request context property to set a connection timeout or to initialize data for sending in a header.</para>
- **Response context** - on the client side, you can access the response context to read the property values set by the inbound message from the last operation invocation. Response context properties are reset after every operation invocation. For example, you might access a response context property to read header information received from the last inbound message.

Setting a request context

To set a particular request context property, *ContextPropertyName* , to the value, *PropertyValue* , use the code shown in the below example.

Example 2.11. Setting a Request Context Property on the Client Side

```
// Set request context property.
java.util.Map<String, Object> requestContext =
  ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(ContextPropertyName, PropertyValue);

// Invoke an operation.
port.SomeOperation();
```

You have to cast the port object to `javax.xml.ws.BindingProvider` in order to access the request context. The request context itself is of type, `java.util.Map<String, Object>` , which is a hash map that has keys of `String` and values of arbitrary type. Use `java.util.Map.put()` to create a new entry in the hash map.

Reading a response context

To retrieve a particular response context property, *ContextPropertyName* , use the code shown below.

Example 2.12. Reading a Response Context Property on the Client Side

```
// Invoke an operation.
port.SomeOperation();

// Read response context property.
java.util.Map<String, Object> responseContext =
   ((javax.xml.ws.BindingProvider)port).getResponseContext();
PropertyType propValue = (PropertyType)
   responseContext.get(ContextPropertyName);
```

The response context is of type, `java.util.Map<String, Object>` , which is a hash map that has keys of type `String` and values of an arbitrary type. Use `java.util.Map.get()` to access an entry in the hash map of response context properties.

Supported contexts

CXF supports the following context properties:

Context Property Name	Context Property Type
`org.apache.cxf.ws.addressing.JAXWSAConstants.CLIENT_ADDRESSING_PROPERTIES`	`org.apache.cxf.ws.addressing.AddressingProperties`

2.3.7.4. Asynchronous Invocation Model

In addition to the usual synchronous mode of invocation, CXF also supports two forms of asynchronous invocation, as follows:

- **Polling approach** - in this case, to invoke the remote operation, you call a special method that has no output parameters, but returns a `javax.xml.ws.Response` instance. The `Response` object (which inherits from the `javax.util.concurrency.Future` interface) can be polled to check whether or not a response message has arrived.
- **Callback approach** - in this case, to invoke the remote operation, you call another special method that takes a reference to a callback object (of `javax.xml.ws.AsyncHandler` type) as one of its parameters. Whenever the response message arrives at the client, the CXF runtime calls back on the `AsyncHandler` object to give it the contents of the response message.

Both of these asynchronous invocation approaches are described here and illustrated by code examples.

Contract for asynchronous example

The below example shows the WSDL contract that is used for the asynchronous example. The contract defines a single port type, `GreeterAsync` , which contains a single operation, `greetMeSometime` .

```
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_async_soap_http"
    xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://apache.org/hello_world_async_soap_http"
    name="HelloWorld">
    <wsdl:types>
        <schema targetNamespace=
           "http://apache.org/hello_world_async_soap_http/types"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
        elementFormDefault="qualified">
            <element name="greetMeSometime">
                <complexType>
                    <sequence>
                        <element name="requestType" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeSometimeResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
        </schema>
    </wsdl:types>
    <wsdl:message name="greetMeSometimeRequest">
        <wsdl:part name="in" element="x1:greetMeSometime"/>
    </wsdl:message>
    <wsdl:message name="greetMeSometimeResponse">
        <wsdl:part name="out" element="x1:greetMeSometimeResponse"/>
    </wsdl:message>
    <wsdl:portType name="GreeterAsync">
        <wsdl:operation name="greetMeSometime">
            <wsdl:input name="greetMeSometimeRequest"
                message="tns:greetMeSometimeRequest"/>
            <wsdl:output name="greetMeSometimeResponse"
                message="tns:greetMeSometimeResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="GreeterAsync_SOAPBinding"
         type="tns:GreeterAsync">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="greetMeSometime">
            <soap:operation style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
```

```
    </wsdl:binding>
    <wsdl:service name="SOAPService">
        <wsdl:port name="SoapPort"
            binding="tns:GreeterAsync_SOAPBinding">
            <soap:address
                location="http://localhost:9000/SoapContext/SoapPort"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Generating the asynchronous stub code

The asynchronous style of invocation requires extra stub code (for example, dedicated asychronous methods defined on the service endpoint interface). This special stub code is not generated by default, however. To switch on the asynchronous feature and generate the requisite stub code, you must use the mapping customization feature from the WSDL 2.0 specification.

Customization enables you to modify the way the wsdl2java utility generates stub code. In particular, it enables you to modify the WSDL-to-Java mapping and to switch on certain features. Here, customization is used to switch on the asynchronous invocation feature. Customizations are specified using a binding declaration, which you define using a `jaxws:bindings` tag (where the jaxws prefix is tied to the http://java.sun.com/xml/ns/jaxws" namespace). There are two alternative ways of specifying a binding declaration:

- **External binding declaration** - the `jaxws:bindings` element is defined in a file separately from the WSDL contract. You specify the location of the binding declaration file to the wsdl2java utility when you generate the stub code.

- **Embedded binding declaration** - you can also embed the `jaxws:bindings` element directly in a WSDL contract, treating it as a WSDL extension. In this case, the settings in `jaxws:bindings` apply only to the immediate parent element.

This section considers only the first approach, the external binding declaration. The template for a binding declaration file that switches on asynchronous invocations is shown below.

Example 2.13. Template for an Asynchronous Binding Declaration

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          wsdlLocation="@WSDL_LOCATION@/hello_world_async.wsdl"
          xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="wsdl:definitions">
    <enableAsyncMapping>true</enableAsyncMapping>
  </bindings>
</bindings>
```

<para>Where *AffectedWSDLContract* specifies the URL of the WSDL contract that is affected by this binding declaration. The *AffectedNode* is an XPath value that specifies which node (or nodes) from the WSDL contract are affected by this binding declaration. You can set *AffectedNode* to `wsdl:definitions` , if you want the entire WSDL contract to be affected. The {jaxws:enableAsyncMapping}} element is set to `true` to enable the asynchronous invocation feature.

For example, if you want to generate asynchronous methods only for the `GreeterAsync` port type, you could specify `<bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']">` in the preceding binding declaration.

Assuming that the binding declaration is stored in a file, `async_binding.xml` , you can generate the requisite stub files with asynchronous support by entering the following wsdl2java command:

```
wsdl2java -ant -client -d ClientDir -b async_binding.xml hello_world.wsdl
```

When you run the wsdl2java command, you specify the location of the binding declaration file using the -b option. After generating the stub code in this way, the `GreeterAsync` service endpoint interface (in the file `GreeterAsync.java`) is defined as shown below.

Example 2.14. Service Endpoint Interface with Methods for Asynchronous Invocations

```
/* Generated by WSDLToJava Compiler. */
package org.apache.hello_world_async_soap_http;
...
import java.util.concurrent.Future;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
...
public interface GreeterAsync {

  public Future<?> greetMeSometimeAsync(
    java.lang.String requestType,
    AsyncHandler<org.myorg.types.GreetMeSometimeResponse>
      asyncHandler
  );

  public Response<org.myorg.types.GreetMeSometimeResponse>
    greetMeSometimeAsync(
    java.lang.String requestType
  );

  public java.lang.String greetMeSometime(
    java.lang.String requestType
  );
}
```

In addition to the usual synchronous method, `greetMeSometime()` , two asynchronous methods are also generated for the `greetMeSometime` operation, as follows:

- `greetMeSometimeAsync()` method with `Future<?>` return type and an extra `javax.xml.ws.AsyncHandler` parameter - call this method for the callback approach to asynchronous invocation.
- `greetMeSometimeAsync()` method with `Response<GreetMeSometimeResponse>` return type - call this method for the polling approach to asynchronous invocation.

The details of the callback approach and the polling approach are discussed in the following subsections.

Implementing an asynchronous client with the polling approach

The next example illustrates the polling approach to making an asynchronous operation call. Using this approach, the client invokes the operation by calling the special Java method, `_OperationName_Async()` , that returns

a `javax.xml.ws.Response<T>` object, where T is the type of the operation's response message. The `Response<T>` object can be polled at a later stage to check whether the operation's response message has arrived.

Example 2.15. Polling Approach for an Asynchronous Operation Call

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_async_soap_http.GreeterAsync;
import org.apache.hello_async_soap_http.SOAPService;
import org.apache.hello_async_soap_http.types.GreetMeSometimeResponse;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://objectweb.org/hello_async_soap_http",
       "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {
    ...
    // Polling approach:
    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));
      while (!greetMeSomeTimeResp.isDone()) {
        Thread.sleep(100);
      }
      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      ...
      System.exit(0);
  }
}
```

The `greetMeSometimeAsync()` method invokes the `greetMeSometimes` operation, transmitting the input parameters to the remote service and returning a reference to a `javax.xml.ws.Response<GreetMeSometimeResponse>` object. The `Response` class is defined by extending the standard `java.util.concurrency.Future<T>` interface, which is specifically designed for polling the outcome of work performed by a concurrent thread. There are essentially two basic approaches to polling using the `Response` object:

- **Non-blocking polling** - before attempting to get the result, check whether the response has arrived by calling the non-blocking `Response<T>.isDone()` method. For example:

  ```
  Response<GreetMeSometimeResponse> greetMeSomeTimeResp = ...;

  if (greetMeSomeTimeResp.isDone()) {
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
  }
  ```

- **Blocking polling** - call `Response<T>.get()` right away and block until the response arrives (optionally specifying a timeout). For example, to poll for a response, with a 60 second timeout:

  ```
  Response<GreetMeSometimeResponse> greetMeSomeTimeResp = ...;
  ```

```
  GreetMeSometimeResponse reply = greetMeSomeTimeResp.get(
    60L,
    java.util.concurrent.TimeUnit.SECONDS
    );
```

Implementing an asynchronous client with the callback approach

An alternative approach to making an asynchronous operation invocation is to implement a callback class, by deriving from the `javax.xml.ws.AsyncHandler` interface. This callback class must implement a `handleResponse()` method, which is called by the CXF runtime to notify the client that the response has arrived. The below example shows an outline of the `AsyncHandler` interface that you need to implement.

Example 2.16. The javax.xml.ws.AsyncHandler Interface

```
package javax.xml.ws;

public interface AsyncHandler<T>
{
  void handleResponse(Response<T> res);
}
```

In this example, a callback class, `TestAsyncHandler` , is defined as shown in the example below.

Example 2.17. The TestAsyncHandler Callback Class

```
package demo.hw.client;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import org.apache.hello_async_soap.types.GreetMeSometimeResponse;

public class TestAsyncHandler implements
    AsyncHandler<GreetMeSometimeResponse> {
  private GreetMeSometimeResponse reply;

  public void handleResponse(Response<GreetMeSometimeResponse>
    response) {
    try {
      reply = response.get();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public String getResponse() {
    return reply.getResponseType();
  }
}
```

The implementation of `handleResponse()` shown in Example11 [55] simply gets the response data and stores it in a member variable, `reply` . The extra `getResponse()` method is just a convenience method that extracts the sole output parameter (that is, `responseType`) from the response.

Example12 [56] illustrates the callback approach to making an asynchronous operation call. Using this approach, the client invokes the operation by calling the special Java method, `_OperationName_Async()` , that returns a `java.util.concurrency.Future<?>` object and takes an extra parameter of `AsyncHandler<T>`.

Example 2.18. Callback Approach for an Asynchronous Operation Call

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_async_soap_http.GreeterAsync;
import org.apache.hello_async_soap_http.SOAPService;
import org.apache.hello_async_soap_http.types.GreetMeSometimeResponse;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
        "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {
    ...
    // Callback approach
    TestAsyncHandler testAsyncHandler = new TestAsyncHandler();
    System.out.println(
       "Invoking greetMeSometimeAsync using callback object...");
    Future<?> response = port.greetMeSometimeAsync(
       System.getProperty("user.name"), testAsyncHandler);
    while (!response.isDone()) {
      Thread.sleep(100);
    }
    resp = testAsyncHandler.getResponse();
    ...
    System.exit(0);
  }
}
```

The `Future<?>` object returned by `greetMeSometimeAsync()` can be used only to test whether or not a response has arrived yet - for example, by calling `response.isDone()` . The value of the response is only made available to the callback object, `testAsyncHandler` .

2.4. Data Binding Options

CXF uses JAXB 2.x as its default databinding.

CXF also includes other data bindings. There is the Aegis data binding which will turn nearly any Java object into something that can be represented using schema, including Maps, Lists, and unannotated java types. CXF 2.1 added an XMLBeans databinding, and CXF 2.3.0 added an SDO databinding.

2.4.1. Aegis

2.4.1.1. What is Aegis?

Aegis is a databinding. That is, it is a subsystem that can map Java objects to XML documents described by XML schema, and vica-versa. Aegis is designed to give useful mappings with a minimum of programmer effort, while allowing detailed control and customization.

Aegis began as part of XFire, and moved with XFire into Apache CXF.

You can use Aegis independently of CXF as a mechanism for mapping Java objects to and from XML. This page, however, describes Aegis as used inside of CXF.

Aegis has some advantages over JAXB for some applications. Some users find that it produces a more natural XML mapping for less configuration. For example, Aegis has a default setting for 'nillable', allowing you to declare it for your entire service in one place instead of having to annotate every single element. The biggest advantage of Aegis, however, is a convenient way to customize the mapping without adding (@)annotations to your Java code. This allows you to avoid class loading dependencies between your data classes and your web service binding.

2.4.1.2. Getting Started: Basic Use of Aegis

You can configure any web *service* to use the Aegis data binding. A service configured with Aegis will yield a valid WSDL description, and you can use that to configure any *client* that you like. You can talk to an Aegis service with JAXB, or .NET, or a scripted language, or ... Aegis itself.

You can use Aegis as a client to talk to Aegis, by using the very same Java classes and configuration files in the client environment that you use on the server. However, it's not all that practical to use Aegis as a client to talk to some a service using some other data binding, since Aegis lacks a 'wsdl2java' tool.

Using Aegis on the client side also carries severe risks of compatibility problems. Since there is no WSDL to specify the contract, small changes in your code or in CXF can result in a situation where the client and the server are incompatible. If you want to use Aegis on the client side, you should be sure to use exactly the same version of CXF on both sides. If you cannot do that, you should consider generating JAX-WS/JAX-B code for the client using wsdl2java.

Every CXF service and client uses a front end: JAX-WS, Simple, etc. Each of these provides a place to configure the data binding, both in Spring and via Java code.

For example, here is a Simple front-end service using Aegis as a data binding.

```
<simple:server id="pojoservice" serviceClass="demo.hw.server.HelloWorld"
              address="/hello_world">
   <simple:serviceBean>
    <bean class="demo.hw.server.HelloWorldImpl" />
   </simple:serviceBean>
         <simple:dataBinding>
       <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding"/>
    </simple:dataBinding>
  </simple:server>
 </bean>
```

AegisDatabinding is the class that integrates Aegis into CXF as a databinding.

Note that AegisDatabinding beans, like all databinding beans, are *not reusable* . The example above uses an anonymous nested bean for the databinding. If you make a first-class bean for a databinding, be sure to use scope='prototype' if you are inclined to define more than one endpoint.

2.4.1.3. Aegis Operations - The Simple Case

How does Aegis work? Aegis maintains, for each service, a set of mappings from Java types (Class<?> objects) to XML Schema types. It uses that mapping to read and write XML. Let's look at a simple service, where all the Java types involved are either Java built-in types, other types with predefined mappings to XML Schema, or simple bean-pattern classes that have properties that (recursively) are simple.

Let's start with *serializing* : mapping from Java to XML. (JAXB calls this marshalling, and cannot decide how many 'l's to use in spelling it.) Given a Java object, Aegis looks to see if it has a mapping. By default, Aegis has a set of default mappings for the basic types defined in XML Schema, plus a few other special items. These mappings are implemented by Java classes, parts of Aegis, that can turn objects in to XML and visa versa. In particular, note that Aegis will map a DataSource or DataHandler to an MTOM attachment.

What if Aegis finds no mapping for a type? In the default configuration, Aegis invokes the *type creators* to create a mapping. Type creators use several mechanisms to create XML schema from Java objects. This include reflection, annotations, and XML type mappings files. As part of the mapping process, Aegis will assign a namespace URI based on the Java package. (**Note** : Aegis does not support elementForm='unqualified' at this time.) These mappings are implemented by a generic mapping class, and stored away.

How about the reverse process: *deserializing* ? (JAXB calls this unmarshalling.) In this case, by default, Aegis is presented with an XML element and asked to produce a Java object. Recall, however, that the Aegis maintains a mapping from Java types to XML Schema Types. By default, an XML instance document offers no information as to the type of a given element. How can Aegis determine the Java type? Outside of CXF, the application would have to tell Aegis the expected type for the root element of a document.

Or, as an alternative, Aegis can add xsi:type attributes to top-level elements when writing. It will always respect them when reading.

Inside CXF, Aegis gets the benefit of the Message and Part information for the service. The WSDL service configuration for a service gives enough information to associate an XML Schema type with each part. Once the front-end has determined the part, it can call Aegis with the QName for the schema type, and Aegis can look it up in the mapping.

Will it be in the mapping? Yes, inside CXF because Aegis precreates mappings for the types in the service's parts. Aegis *cannot* dynamically create or choose a Java class based on XML schema, so the type creators cannot start from XML. Thus, outside CXF you are responsible for ensuring that your top-level types are mapped.

Schema Validation

As of CXF 2.3, the Aegis databinding can leverage the Schema Validation capabilities built into the Woodstox 4.x Stax parser to validate incoming requests. To enable this, you must do the following:

1. Make sure you are using the Woodstox 4.x Stax parser and not a 3.x or other parser. By default, CXF 2.3 ships with an appropriate version of Woodstox.

2. If not using the CXF bundle jar, (example, if using maven), you'll need to add the cxf-wstx-msv-validation-2.3.0.jar to the classpath

3. If not using maven or similar to obtain the cxf-wstx-msv-validation jar, you'll also need to add the msv validation jars as CXF does not ship them by default. You will need:

   ```
   isorelax-20030108.jar
   msv-core-2009.1.jar
   relaxngDatatype-20020414.jar
   xercesImpl-2.9.1.jar
   xml-resolver-1.2.jar
   xsdlib-2009.1.jar
   ```

4. If not using a default bus (such as configuring your own spring context), you'll need to add:

```
<import resource=
    "classpath:META-INF/cxf/cxf-extension-wstx-msv-validation.xml"/>
```

to load the validator utilities that Aegis will use.

5. Turn on schema validation like you would for JAXB by using the @SchemaValidation annotation or setting the "schema-validation-enabled" property on the endpoint to "true".

2.4.1.4. Using Java Classes That Aren't Visible to the Service Interface

Many web service programmers want to use types that are *not* directly visible by reflection of the service interface. Here are some popular examples of types that programmers want to use for property or parameter types:

- Declare a base type, but transfer any one of a number of classes that extend it.
- Declare a raw Collection class, such as a Set, List, or Map, and send arbitrary objects as keys and values.
- Declare a base exception type for 'throws', and then throw other exception classes that derive from it.
- Declare an interface or an abstract type.

Aegis can handle all of these. For all except interfaces, there are two mechanisms involved: the root class list and xsi:type attributes.

As explained above, Aegis can write 'anything', but it can only read objects of types that are mapped. You must give Aegis a list of all the types that you want to use over and above those visible from the service, and you must instruct Aegis to send xsi:type attributes when sending objects of such types. These type attributes allow Aegis to identify the type of these additional objects and look them up in the mappings.

Interfaces and abstract types require one further step. Obviously, Aegis cannot instantiate (run 'new') on an interface. Thus, knowing that a particular XML Schema type maps to an interface is not enough information. To be able to read an XML element that corresponds to an interface, Aegis must know a 'proxy class' that implements the interface. You must give Aegis a mapping from interface types to proxy class names.

How does this work? The core of Aegis is the AegisContext class. Each AegisDatabinding object has an AegisContext. (It is probably not possible to share an AegisContext amongst databindings.)

By default, AegisDatabinding will create its own AegisContext with default properties. To configure additional types, as well control other options that we will examine later on, you must create the AegisContext for yourself and specify some of its properties. Then you pass your AegisContext object into your AegisDatabinding object.

To use additional classes or interfaces, you need to set two (or three) properties of your AegisContext.

- **rootClasses** is a collection of Java Class<?> objects. These are added to the list of types known to Aegis. Aegis will create a mapping for each. For convenience, there is a rootClassNames property for use from Spring. It is a list of Strings containing class names.
- **writeXsiTypes** is a boolean. Set it to **true** to send xsi:type attributes.
- **beanImplementationMap** is a mapping from Class<?> to class names. Use this to specify proxy classes for interfaces (or abstract classes).

2.4.1.5. Global Type Creation Options

There are a few global options to the default type mapping process. You can control these by creating a org.apache.cxf.aegis.type.TypeCreationOptions and passing it into your AegisContext object.

There are four properties in the class, of which two are much more commonly used.

- **defaultNillable** defines the default value of the **nillable** attribute of xsd:element items in the xsd:sequences built for non-primitive types. By default, it is true, since any Java reference can be null. However, nillable='true' has annoying consequences in some wsdl2java tools (turning scalars into arrays, e.g.), and so many programmers prefer to default to **false** .
- **defaultMinOccurs** defines the default value of the **minOccurs** attribute of xsd:element items in the xsd:sequences built for Java arrays. In combination with nillable, programmers often want to adjust this value from 0 to 1 to get a more useful mapping of an array.
- **defaultExtensibleElements** causes each sequence to end with an xsd:any. The idea here is to allow for schema evolution; a client that has generated Java from one version of the service will tolerate data from a newer version that has additional elements. Use this feature with care; version management of web services is a complex topic, and xsd:any may have unexpected consequences.
- **defaultExtensibleAttributes** causes each element to permit any attribute. By default, Aegis doesn't map **any** properties or parameters to attributes. As with the element case, care is called for.

Note that these are options to the default type creators. If you take the step of creating a customized type creator, it will be up to you to respect or ignore these options.

Here's a quick example of Java code setting these options. In Spring you would do something analogous with properties.

```
TypeCreationOptions tOpts = new TypeCreationOptions();
tOpts.setDefaultMinOccurs(1);
tOpts.setDefaultNillable(false);
AegisDatabinding aDB = new AegisDatabinding();
aDB.getAegisContext().setTypeCreationOptions(tOpts);
sFactory.getServiceFactory().setDataBinding(aDB);
```

2.4.1.6. Detailed Control of Bean Type Mapping

This page has descended, gradually, from depending on Aegis' defaults toward exercising more detailed control over the process. The next level of detail is to customize the default type creators' behavior via XML mapping files and annotations.

XML Mapping Files

XML mapping files are a major distinguishing feature of Aegis. They allow you to specify details of the mapping process without either (a) modifying your Java source for your types or (b) maintaining a central file of some kind containing mapping instructions.

Aegis XML mapping applies to services and to beans. By "beans," we mean "Java classes that follow the bean pattern, used in a web service." "Services," you ask? Aren't they the responsibility of the CXF front end? There is some overlap in the responsibilities of front-ends and databindings.

By and large, front-ends map services to XML schema, filling in XML Schema elements and types for messages and parts. Data bindings then map from those schema items to Java. However, Aegis also provides XML configuration for methods and parameters, which 'poach' in the territory of the front end. This works well for the Simple front end, which has no other way to control these mappings. The present author is not sure what will happen in the event of a conflict between Aegis and any other front-end, like JAX-WS, that has explicit configuration. Thus, Aegis service configuration is best used with the Simple front end.

For both bean and service customization, Aegis looks for customization in files found by the classloader. If your class is my.hovercraft.is.full.of.Eels, Aegis will search the classpath for /my/hovercraft/is/full/of/Eels.aegis.xml. In other words, if Eels.class is sitting in a JAR file or a directory, Eels.aegis.xml can be sitting right next to it.

Or, on the other hand, it can be in a completely different JAR or tree, so long as it ends up in the same logical location. In other words, you can create XML files for classes when you don't even have their source.

This is a copy of the XML Schema for mapping XML files that is annotated with comments.

Bean Mapping

Here is a very simple mapping. It takes a property named 'horse', renames it to 'feathers', and makes it an attribute instead of an element.

```
<mappings>
  <mapping name="">
    <property name="horse" mappedName="Feathers" style="attribute"/>
  </mapping>
</mappings>
```

Names and Namespaces

You can also specify the full QName of the bean itself. The following mapping causes a class to have the QName {urn:north-pole:operations}Employee.

```
<mappings xmlns:np="urn:north-pole:operations">
  <mapping name="np:Employee">
  </mapping>
</mappings>
```

Notice that the namespace was declared on the mappings element and then the prefix was used to specify the element QNames for the name/title properties.

This will result in a mapping like so:

```
<np:Employee xmlns:np="urn:north-pole:operations">
  <np:Name>Santa Claus</np:Name>
  <np:Title>Chief Present Officer (CPO)</np:Title>
</np:Employee>
```

Ignoring properties

If you don't want to serialize a certain property it is easy to ignore it:

```
<mappings>
  <mapping>
    <property name="propertyName" ignore="true"/>
  </mapping>
</mappings>
```

MinOccurs and Nillable

The default Aegis mapping is to assume that, since any Java object can be null, that the corresponding schema elements should have minOccurs of 0 and nillable of true. There are properties on the mappings for to control this.

```
<mappings>
  <mapping>
    <property name='everpresentProperty' minOccurs='1'
       nillable='false'/>
```

```
  </mapping>
<mappings>
```

Alternative Type Binding

Later on, we will explain how to replace the default mappings that Aegis provides for basic types. However, there are some cases where you may want to simply specify one of the provided type mappings for one of your properties. You can do that from the XML mapping file without creating any Java customization.

By default, for example, if Aegis maps a property as a Date, it uses the XML schema type xsd:dateTime. Here is an example that uses xsd:date, instead.

```
<mappings xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <mapping>
  <property name="birthDate"
   type="org.apache.cxf.aegis.type.basic.DateType"
   typeName="xsd:date"
   />
 </mapping>
</mappings>
```

Collections

If you use a 'raw' collection type, Aegis will map it as a collection of xsd:any particles. If you want the WSDL to show it as a collection of some specific type, the easiest thing to do is to use Java generics instead of raw types. If, for some reason, you can't do that, you can use the componentType and keyType attributes of a property to specify the Java classes.

Multiple mappings for Different Services

What if you want to specify different mapping behavior for different services on the same types? The 'mapping' element of the file accepts a 'uri' attribute. Each AegisContext has a 'mappingNamespaceURI' attribute. If a mapping in a .aegis.xml file has a uri attribute, it must match the current service's uri.

Services and Parameters

For a service, mapping files specify attributes of operations and parameters.

This example specifies that getUnannotatedStrings returns an element named UnannotatedStringCollection which is a raw collection of String values. It then specifies the first parameter of getValues is also a raw collection of String values.

```
<mappings>
 <mapping>
  <method name="getUnannotatedStrings">
   <return-type name="UnannotatedStringCollection"
       componentType="java.lang.String"/>
  </method>
        <method name="getValues">
         <parameter index="0" componentType="java.lang.String"/>
        </method>
 </mapping>
</mappings>
```

Annotations

Like JAXB, Aegis supports some Java annotations to control the mapping process. These attributes are modelled after JAXB. Aegis defines them in the package org.apache.cxf.aegis.type.java5. They are:

- XmlAttribute
- XmlType
- XmlElement
- XmlReturnType
- XmlIgnore

In addition, Aegis will respect actual JAXB annotations from the following list:

- javax.jws.WebParam
- javax.jws.WebResult
- javax.xml.bind.annotation.XmlAttribute
- javax.xml.bind.annotation.XmlElement
- javax.xml.bind.annotation.XmlSchema
- javax.xml.bind.annotation.XmlType
- javax.xml.bind.annotation.XmlTransient

Note, however, that Aegis does not handle package-info.java classes, and so XmlSchema must be applied to a class.

2.4.1.7. Creating Your Own Type Mappings

If you want complete control on the mapping between Java and XML, you must create your own type mappings. To do this, you should make a class that extends org.apache.cxf.aegis.type.Type, and then you must register it in a type mapping for your service.

To see how these classes work, read the source code.

To register your type mappings, you have two choices.

If you just want to add a custom type mapping into your service, the easiest thing to do is to retrieve the TypeMapping from the AegisContext, and register your type as a mapping from Class<?> to your custom mapping object.

If you want complete control over the process, you can create your own TypeMapping. The class DefaultTypeMapping is the standard type map. You can use these, or you can create your own implementation of TypeMapping. Set up your type mapping as you like, and install it in your context *before* the service is initialized.

2.4.1.8. Customizing Type Creation

What if you want to change how Aegis builds new type mappings and types from Java classes? You can create your own TypeCreator, and either put it in the front of the list of type creators or replace the entire standard list.

As with type mappings, reading the source is the only way to learn the details. Type creators are associated with type mappings; you can call setTypeCreator on an instance of DefaultTypeMapping to install yours.

2.4.1.9. Aegis Default Mappings

For services declared to operate with Soap 1.1, Aegis sets up two sets of mappings.

Soap 1.1 SOAP mappings

Type	**SOAP Mapping**
boolean	Soap-encoded boolean
Boolean	Soap-encoded boolean
int	Soap-encoded int
Integer	Soap-encoded int
short	Soap-encoded int
Short	Soap-encoded int
double	Soap-encoded double
Double	Soap-encoded double
float	Soap-Encoded float
Float	Soap-Encoded float
long	Soap-encoded long
Long	Soap-encoded long
char	Soap-encoded char
Character	Soap-encoded char
String	Soap-encoded String
java.sql.Date	Soap-encoded date-time
java.util.Calendar	Soap-encoded date-time
byte[]	soap-encoded Base64
BigDecimal	Soap-encoded Decimal
BigInteger	Soap-encoded BigInteger

Soap 1.1 XSD mappings

Type	**XSD Mapping**
boolean	XSD boolean
Boolean	XSD boolean
int	XSD int
Integer	XSD int
short	XSD int
Short	XSD int
double	XSD double
Double	XSD double
float	XSD float
Float	XSD float

Type	XSD Mapping
long	XSD long
Long	XSD long
char	XSD char
Character	XSD char
String	XSD String
java.sql.Date	XSD date-time
java.sql.Time	XSD time
java.util.Calendar	XSD date-time
byte[]	XSD Base64
BigDecimal	XSD Decimal
BigInteger	XSD Integer
org.w3c.Document	XSD Any
org.jdom.Document	XSD Any
org.jdom.Element	XSD Any
javax.xml.transform.source	XSD Any
javax.xml.stream.XMLStreamReader	XSD Any
Object	XSD Any
javax.activation.DataSource	XSD Base64 via MTOM data source type (See org.apache.cxf.argis.type.mtom)
javax.activation.DataHandler	XSD Base64 via MTOM data source type (See org.apache.cxf.argis.type.mtom)

Services that Don't Use Soap 1.1

The type mappings for non-Soap-1.1 services start out with the same XSD types as the Soap-1.1 services

Type	XSD Mapping
boolean	XSD boolean
Boolean	XSD boolean
int	XSD int
Integer	XSD int
short	XSD int
Short	XSD int
double	XSD double
Double	XSD double
float	XSD float
Float	XSD float
long	XSD long
Long	XSD long
char	XSD char
Character	XSD char

Type	XSD Mapping
String	XSD String
java.sql.Date	XSD date-time
java.sql.Time	XSD time
java.util.Calendar	XSD date-time
byte[]	XSD Base64
BigDecimal	XSD Decimal
BigInteger	XSD Integer
org.w3c.Document	XSD Any
org.jdom.Document	XSD Any
org.jdom.Element	XSD Any
javax.xml.transform.source	XSD Any
javax.xml.stream.XMLStreamReader	XSD Any
Object	XSD Any
javax.activation.DataSource	Base64 via MTOM data source type (See org.apache.cxf.argis.type.mtom)
javax.activation.DataHandler	Base54 MTOM data source type (See org.apache.cxf.argis.type.mtom)

These services get some additional mappings, as well:

Type	Mapping
javax.xml.datatype.Duration	XSD Duration
javax.xml.datatype.XMLGregorianCalendar	XSD Date
javax.xml.datatype.XMLGregorianCalendar	XSD Time
javax.xml.datatype.XMLGregorianCalendar	XSD gDay
javax.xml.datatype.XMLGregorianCalendar	XSD gMonth
javax.xml.datatype.XMLGregorianCalendar	XSD gMonthDay
javax.xml.datatype.XMLGregorianCalendar	XSD gYear
javax.xml.datatype.XMLGregorianCalendar	XSD gYearMonth
javax.xml.datatype.XMLGregorianCalendar	XSD Date-Time

2.4.2. JAXB

2.4.2.1. Introduction

JAXB is the default data binding for CXF. If you don't specify one of the other data bindings in your Spring configuration or through the API, you will get JAXB. CXF 2.0.x branch supplies JAXB 2.0, CXF 2.1.x and CXF 2.2.x use JAXB 2.1.

JAXB uses Java annotation combined with files found on the classpath to build the mapping between XML and Java. JAXB supports both code-first and schema-first programming. The schema-first support the ability to create a client proxy, dynamically, at runtime. See the CXF DynamicClientFactory class.

CXF uses the JAXB reference implementation. To learn more about annotating your classes or how to generate beans from a schema, please read the JAXB user's guide .

2.4.2.2. JAXB versus JAX-WS (or other front-ends)

There are some pitfalls in the interaction between the front end and the data binding. If you need detailed control over the XML that travels on the wire, you may want to avoid the 'wrapped' alternative, and stick with 'bare'. When you use the wrapped parameter style or the RPC binding, the front ends construct more or less elaborate XML representations for your operations. You have less control over those constructs than you do over JAXB's mappings. In particular, developers with detailed requirements to control the XML Schema 'elementFormDefault' or the use or non-use of XML namespace prefixes often become frustrated because the JAXB annotations for these options don't effect mappings that are purely the work of the front-end. The safest course is to use Document/Literal/Bare.

2.4.2.3. Configuring JAXB

CXF allows you to configure JAXB in two ways.

JAXB Properties

JAXB allows the application to specify two sets of properties that modify its behavior: *context* properties and *marshaller* properties. CXF allows applications to add to these properties. **Take care.** In some cases, CXF sets these properties for its own use.

You can add items to both of these property sets via the JAXBDataBinding class. The 'contextProperties' and 'marshallerProperties' *properties* (in the Spring sense) of JAXBDataBinding each store a Map<String, Object>. Whatever you put in the map, CXF will pass along to JAXB. See the JAXB documentation for details.

```
<jaxws:server id="bookServer"
   serviceClass="org.apache.cxf.mytype.AnonymousComplexTypeImpl"
    address="http://localhost:8080/act"
    bus="cxf">
    <jaxws:invoker>
      <bean class="org.apache.cxf.service.invoker.BeanInvoker">
        <constructor-arg>
          <bean class="org.apache.cxf.mytype.AnonymousComplexTypeImpl"/>
        </constructor-arg>
      </bean>
    </jaxws:invoker>
     <jaxws:dataBinding>
      <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
        <property name="contextProperties">
      <map>
      <entry>
         <key><value>com.sun.xml.bind.defaultNamespaceRemap</value></key>
         <value>uri:ultima:thule</value>
      </entry>
        </map>
        </property>
      </bean>
      </jaxws:dataBinding>
  </jaxws:server>
```

Activating JAXB Validation of SOAP requests and responses

For the client side

```
<jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort"
     createdFromAPI="true">
     <jaxws:properties>
         <entry key="schema-validation-enabled" value="true" />
     </jaxws:properties>
</jaxws:client>
```

You may also do this programmatically:

```
((BindingProvider)port).getRequestContext().put(
          "schema-validation-enabled", "true");
```

For the server side

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
   wsdlLocation="wsdl/hello_world.wsdl" createdFromAPI="true">
     <jaxws:properties>
         <entry key="schema-validation-enabled" value="true" />
     </jaxws:properties>
   </jaxws:endpoint>
```

You can also use the org.apache.cxf.annotations.SchemaValidation annotation.

Namespace Prefix Management

The JAXB reference implementation allows the application to provide an object that in turn maps namespace URI's to prefixes. You can create such an object and supply it via the marshaller properties. However, CXF provides an easier process. The namespaceMap property of the JAXBDataBinding accepts a Map<String, String>. Think of it as a map from namespace URI to namespace prefix. If you load up this map, CXF will set up the necessary marshaller property for you.

```
<jaxws:server id="bookServer"
   serviceClass="org.apache.cxf.mytype.AnonymousComplexTypeImpl"
    address="http://localhost:8080/act"
    bus="cxf">
    <jaxws:invoker>
      <bean class="org.apache.cxf.service.invoker.BeanInvoker">
        <constructor-arg>
          <bean class="org.apache.cxf.mytype.AnonymousComplexTypeImpl"/>
        </constructor-arg>
      </bean>
    </jaxws:invoker>
     <jaxws:dataBinding>
      <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
        <property name="namespaceMap">
      <map>
      <entry>
         <key>
            <value>
               http://cxf.apache.org/anonymous_complex_type/
            </value>
         </key>
         <value>BeepBeep</value>
```

```
        </entry>
      </map>
      </property>
    </bean>
    </jaxws:dataBinding>
  </jaxws:server>
```

2.4.3. MTOM Attachments with JAXB

MTOM is a standard which allows your services to transfer binary data efficiently and conveniently. Many frameworks have support for MTOM - Axis2, JAX-WS RI, JBoss WS, XFire, Microsoft's WCF, and more.

If the binary is part of the XML document, it needs to be base64 encoded - taking CPU time and increasing the payload size. When MTOM is enabled on a service, it takes binary data which might normally be part of the XML document, and creates an attachment for it.

Enabling MTOM is a rather simple process. First, you must annotate your schema type or POJO to let JAXB know that a particular field could be a candidate for MTOM optimization. Second, you just tell CXF that you wish to enable MTOM.

This page tells you how to activate MTOM for JAXB. MTOM is also supported in Aegis.

2.4.3.1. 1) Annotating the Message

1a) Modifying your schema for MTOM

Lets say we have a Picture schema type like this:

```
<schema targetNamespace="http://pictures.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <element name="Picture">
    <complexType>
      <sequence>
        <element name="Title" type="xsd:string"/>
        <element name="ImageData" type="xsd:base64Binary"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

In this case the ImageData element is something we would like to have transferred as an attachment. To do this we just need to add an xmime:expectedContentTypes annotation:

```
<schema targetNamespace="http://pictures.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
  <element name="Picture">
    <complexType>
      <sequence>
        <element name="Title" type="xsd:string"/>
        <element name="ImageData" type="xsd:base64Binary"
           xmime:expectedContentTypes="application/octet-stream"/>
```

```
      </sequence>
    </complexType>
  </element>
</schema>
```

This tells JAXB (which WSDL2Java uses to generate POJOs for your service) that this field could be of any content type. Instead of creating a byte[] array for the base64Binary element, it will create a DataHandler instead which can be used to stream the data.

1b) Annotation your JAXB beans to enable MTOM

If you're doing code first, you need to add an annotation to your POJO to tell JAXB that the field is a candidate for MTOM optimization. Lets say we have a Picture class with has Title and ImageData fields, then it might look like this:

```
@XmlType
public class Picture {
  private String title;

  @XmlMimeType("application/octet-stream")
  private DataHandler imageData;

  public String getTitle() { return title; }
  public void setTitle(String title) { this.title = title; }

  public DataHandler getImageData() { return imageData; }
  public void setImageData(DataHandler imageData)
     { this.imageData = imageData; }
}
```

Note the use of 'application/octet-stream'. According to the standard, you should be able to use any MIME type you like, in order to specify the actual content of the attachment. However, due to a defect in the JAX-B reference implementation, this won't work.

2.4.3.2. 2) Enable MTOM on your service

If you've used JAX-WS to publish your endpoint you can enable MTOM like so:

```
import javax.xml.ws.Endpoint;
import javax.xml.ws.soap.SOAPBinding;

Endpoint ep = Endpoint.publish("http://localhost/myService",
   new MyService());
SOAPBinding binding = (SOAPBinding) ep.getBinding();
binding.setMTOMEnabled(true);
```

Or, if you used XML to publish your endpoint:

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

```
http://cxf.apache.org/schema/jaxws.xsd">

  <jaxws:endpoint
    id="helloWorld"
    implementor="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>

</beans>
```

If you're using the simple frontend you can set the mtom-enabled property on your ServerFactoryBean or ClientProxyFactoryBean:

```
Map<String,Object> props = new HashMap<String, Object>();
// Boolean.TRUE or "true" will work as the property value here
props.put("mtom-enabled", Boolean.TRUE);

ClientProxyFactoryBean pf = new ClientProxyFactoryBean();
pf.setPropertyies(props);
....
YourClient client = (YourClient) pf.create();

ServerFactoryBean sf = new ServerFactoryBean();
sf.setPropertyies(props);
...
sf.create();
```

Similarly, you can use the XML configuration:

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

  <simple:server
    id="helloWorld"
    serviceClass="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <simple:properties>
      <entry key="mtom-enabled" value="true"/>
    </simple:properties>
  </simple:server>

  <simple:client
    id="helloWorldClient"
    serviceClass="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <simple:properties>
      <entry key="mtom-enabled" value="true"/>
    </simple:properties>
  </simple:client>
```

```
</beans>
```

2.4.3.3. Using DataHandlers

Once you've got the above done, its time to start writing your logic. DataHandlers are easy to use and create. To consume a DataHandler:

```
Picture picture = ...;
DataHandler handler = picture.getImageData();
InputStream is = handler.getInputStream();
```

There are many ways to create DataHandlers. You can use a FileDataSource, ByteArrayDataSource, or write your own DataSource:

```
DataSource source = new ByteArrayDataSource(new byte[] {...},
   "content/type");
DataSource source = new FileDataSource(new File("my/file"));

Picture picture = new Picture();
picture.setImageData(new DataHandler(source));
```

2.4.4. SDO

Apache CXF 2.3 added support for the Tuscany implementation of Service Data Objects as alternative data binding.

2.4.4.1. Setup

By default, CXF does not ship with the Tuscany SDO jars. You will need to acquire them elsewhere and add them to the classpath for the SDO databinding to work. The list of required jars are:

```
backport-util-concurrent-3.0.jar
codegen-2.2.3.jar
codegen-ecore-2.2.3.jar
common-2.2.3.jar
ecore-2.2.3.jar
ecore-change-2.2.3.jar
ecore-xmi-2.2.3.jar
tuscany-sdo-api-r2.1-1.1.1.jar
tuscany-sdo-impl-1.1.1.jar
tuscany-sdo-lib-1.1.1.jar
tuscany-sdo-tools-1.1.1.jar
xsd-2.2.3.jar
```

2.4.4.2. Code Generation

If all the SDO required jars are available (by default, CXF does not ship them, see above), wsld2java tool can be run with the -db sdo flag to have the code generator emit SDO objects instead of the default JAXB objects. The generated SEI interface will have @DataBinding(org.apache.cxf.sdo.SDODataBinding.class) annotation on it which is enough to configure the runtime to know to use SDO.

2.4.5. XMLBeans

Apache XMLBeans is another technology for mapping XML Schema to java objects. CXF added support for XMLBeans in 2.1. There are a two parts to the support for XMLBeans:

2.4.5.1. Code Generation

The wsdl2java tool now allows a "-db xmlbeans" flag to be added that will generate XMLBeans types for all the schema beans instead of the default JAXB beans. With 2.1 and 2.2, the types are generated, but you still need to configure the XMLBeans databinding to be used at runtime. With 2.3, the generated code contains an @Databinding annotation marking it as XMLBeans and the configuration is unnecessary.

2.4.5.2. Runtime

You need to configure the runtime to tell it to use XMLBeans for the databinding instead of JAXB.

Spring config

For the server side, your spring configuration would contain something like:

```
<jaxws:server serviceClass="demo.hw.server.HelloWorld"
    address="/hello_world">
    <jaxws:dataBinding>
       <bean class="org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
    </jaxws:dataBinding>
</jaxws:server>
```

or

```
<jaxws:endpoint
    id="helloWorld"
    implementor="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <jaxws:dataBinding>
       <bean class="org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
    </jaxws:dataBinding>
</jaxws:endpoint>
```

The client side is very similar:

```
<jaxws:client id="helloClient"
                  serviceClass="demo.spring.HelloWorld"
                  address="http://localhost:9002/HelloWorld">
    <jaxws:dataBinding>
       <bean class="org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
    </jaxws:dataBinding>
<jaxws:client>
```

FactoryBeans

If using programmatic factory beans instead of spring configuration, the databinding can be set on the ClientProxyFactoryBean (and subclasses) and the ServerFactoryBean (and subclasses) via:

```
factory.getServiceFactory().setDataBinding(
               new org.apache.cxf.xmlbeans.XmlBeansDataBinding());
```

2.5. CXF Transports

2.5.1. HTTP Transport

HTTP transport support via servlet-based environments is described below (embedded Jetty and OSGi deployment is also available in CXF).

2.5.1.1. Client HTTP Transport (including SSL support)

Configuring SSL Support

To configure your client to use SSL, you'll need to add an <http:conduit> definition to your XML configuration file. If you are already using Spring, this can be added to your existing beans definitions.

A wsdl_first_https sample can be found in the CXF distribution with more detail. Also see this blog entry for another example.

Here is a sample of what your conduit definition might look like:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="
        http://cxf.apache.org/configuration/security
        http://cxf.apache.org/schemas/configuration/security.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <http:conduit
      name="{http://apache.org/hello_world}HelloWorld.http-conduit">

       <http:tlsClientParameters>
          <sec:keyManagers keyPassword="password">
               <sec:keyStore type="JKS" password="password"
                    file="my/file/dir/Morpit.jks"/>
          </sec:keyManagers>
          <sec:trustManagers>
              <sec:keyStore type="JKS" password="password"
                   file="my/file/dir/Truststore.jks"/>
          </sec:trustManagers>
          <sec:cipherSuitesFilter>
            <!-- these filters ensure that a ciphersuite with
              export-suitable or null encryption is used,
              but exclude anonymous Diffie-Hellman key change as
```

```
              this is vulnerable to man-in-the-middle attacks -->
            <sec:include>.*_EXPORT_.*</sec:include>
            <sec:include>.*_EXPORT1024_.*</sec:include>
            <sec:include>.*_WITH_DES_.*</sec:include>
            <sec:include>.*_WITH_NULL_.*</sec:include>
            <sec:exclude>.*_DH_anon_.*</sec:exclude>
          </sec:cipherSuitesFilter>
      </http:tlsClientParameters>
      <http:authorization>
         <sec:UserName>Betty</sec:UserName>
         <sec:Password>password</sec:Password>
      </http:authorization>
      <http:client AutoRedirect="true" Connection="Keep-Alive"/>

   </http:conduit>

</beans>
```

The first thing to notice is the "name" attribute on <http:conduit>. This allows CXF to associate this HTTP Conduit configuration with a particular WSDL Port. The name includes the service's namespace, the WSDL port name (as found in the wsdl:service section of the WSDL), and ".http-conduit". It follows this template: "{WSDL Namespace}portName.http-conduit". Note: it's the PORT name, not the service name. Thus, it's likely something like "MyServicePort", not "MyService". If you are having trouble getting the template to work, another (temporary) option for the name value is simply "*.http-conduit".

Another option for the name attribute is a reg-ex expression for the ORIGINAL URL of the endpoint. The configuration is matched at conduit creation so the address used in the WSDL or used for the JAX-WS Service.create(...) call can be used for the name. For example, you can do:

```
<http:conduit name="http://localhost:8080/.*">
       ......
   </http:conduit>
```

to configure a conduit for all interactions on localhost:8080. If you have multiple clients interacting with different services on the same server, this is probably the easiest way to configure it.

Advanced Configuration

HTTP client endpoints can specify a number of HTTP connection attributes including whether the endpoint automatically accepts redirect responses, whether the endpoint can use chunking, whether the endpoint will request a keep-alive, and how the endpoint interacts with proxies.

A client endpoint can be configured using three mechanisms:

- Configuration
- WSDL
- Java code

Using Configuration

Namespace

The elements used to configure an HTTP client are defined in the namespace http://cxf.apache.org/transports/http/configuration" . It is commonly referred to using the prefix `http-conf` . In order to use the HTTP configuration

elements you will need to add the lines shown below to the beans element of your endpoint's configuration file. In addition, you will need to add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 2.19. HTTP Consumer Configuration Namespace

```
<beans ...
       xmlns:http-conf=
          "http://cxf.apache.org/transports/http/configuration
       ...
       xsi:schemaLocation="...
            http://cxf.apache.org/transports/http/configuration
            http://cxf.apache.org/schemas/configuration/http-conf.xsd
       ...>
```

The conduit element

You configure an HTTP client using the `http-conf:conduit` element and its children. The `http-conf:conduit` element takes a single attribute, `name` , that specifies the WSDL port element that corresponds to the endpoint. The value for the `name` attribute takes the form *portQName* `.http-conduit` . For example, the code below shows the `http-conf:conduit` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment `<port binding="widgetSOAPBinding" name="widgetSOAPPort>` if the endpoint's target namespace was http://widgets.widgetvendor.net" .

Example 2.20. http-conf:conduit Element

```
...
  <http-conf:conduit name=
     "{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
    ...
  </http-conf:conduit>

  <http-conf:conduit name="*.http-conduit">
  <!-- you can also using a wild card specify the http-conduit that
     you want to configure -->
    ...
  </http-conf:conduit>
...
```

The `http-conf:conduit` element has a number of child elements that specify configuration information. They are described below. See also Sun's JSSE Guide for more information on configuring SSL.

Element	Description
`http-conf:client`	Specifies the HTTP connection properties such as timeouts, keep-alive requests, content types, etc.
`http-conf:authorization`	Specifies the the parameters for configuring the basic authentication method that the endpoint uses preemptively.
`http-conf:proxyAuthorization`	Specifies the parameters for configuring basic authentication against outgoing HTTP proxy servers.
`http-conf:tlsClientParameters`	Specifies the parameters used to configure SSL/TLS.
`http-conf:basicAuthSupplier`	Specifies the bean reference or class name of the object that supplies the the basic authentication information used by the endpoint both preemptively or in response to a 401 HTTP challenge.

Element	Description
`http-conf:trustDecider`	Specifies the bean reference or class name of the object that checks the HTTP(S) URLConnection object in order to establish trust for a connection with an HTTPS service provider before any information is transmitted.

The client element

The `http-conf:client` element is used to configure the non-security properties of a client's HTTP connection. Its attributes, described below, specify the connection's properties.

Attribute	Description
`ConnectionTimeout`	Specifies the amount of time, in milliseconds, that the client will attempt to establish a connection before it times out. The default is 30000 (30 seconds). 0 specifies that the client will continue to attempt to open a connection indefinitely.
`ReceiveTimeout`	Specifies the amount of time, in milliseconds, that the client will wait for a response before it times out. The default is 60000. 0 specifies that the client will wait indefinitely.
`AutoRedirect`	Specifies if the client will automatically follow a server issued redirection. The default is false.
`MaxRetransmits`	Specifies the maximum number of times a client will retransmit a request to satisfy a redirect. The default is -1 which specifies that unlimited retransmissions are allowed.
`AllowChunking`	Specifies whether the client will send requests using chunking. The default is true which specifies that the client will use chunking when sending requests. Chunking cannot be used used if either of the following are true: • `http-conf:basicAuthSupplier` is configured to provide credentials preemptively. • `AutoRedirect` is set to true. In both cases the value of `AllowChunking` is ignored and chunking is disallowed. See note about chunking below.
`ChunkingThreshold`	Specifies the threshold at which CXF will switch from non-chunking to chunking. By default, messages less than 4K are buffered and sent non-chunked. Once this threshold is reached, the message is chunked.
`Accept`	Specifies what media types the client is prepared to handle. The value is used as the value of the HTTP `Accept` property. The value of the attribute is specified using as multipurpose internet mail extensions (MIME) types. See note about chunking below.
`AcceptLanguage`	Specifies what language (for example, American English) the client prefers for the purposes of receiving a response. The value is used as the value of the HTTP AcceptLanguage property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.
`AcceptEncoding`	Specifies what content encodings the client is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP `AcceptEncoding` property.

Attribute	Description
`ContentType`	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP `ContentType` property. The default is `text/xml` . **Tip:** For web services, this should be set to `text/xml` . If the client is sending HTML form data to a CGI script, this should be set to application/x-www-form-urlencoded. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to application/octet-stream.
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP `Host` property. **Tip:** This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).
`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • `Keep-Alive` (default) specifies that the client wants to keep its connection open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • `close` specifies that the connection to the server is closed after each request/response sequence.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
`Cookie`	Specifies a static cookie to be sent with all requests.
`BrowserType`	Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent* . Some servers optimize based upon the client that is sending the request.
`Referer`	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property. **Note:** This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. **Important:** If the AutoRedirect attribute is set to true and the request is redirected, any value specified in the Refererattribute is overridden. The value of the HTTP Referer property will be set to the URL of the service who redirected the consumer's original request.
`DecoupledEndpoint`	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate server->client connection. **Warning:** You must configure both the client and server to use WS-Addressing for the decoupled endpoint to work.
`ProxyServer`	Specifies the URL of the proxy server through which requests are routed.
`ProxyServerPort`	Specifies the port number of the proxy server through which requests are routed.
`ProxyServerType`	Specifies the type of proxy server used to route requests. Valid values are: • HTTP(default)

Attribute	Description
	• SOCKS

Example using the Client Element

The example below shows a the configuration for an HTTP client that wants to keep its connection to the server open between requests, will only retransmit requests once per invocation, and cannot use chunking streams.

Example 2.21. HTTP Consumer Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf=
          "http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit name=
     "{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http-conf:client Connection="Keep-Alive"
                      MaxRetransmits="1"
                      AllowChunking="false" />
  </http-conf:conduit>
</beans>
```

Again, see the Configuration page for information on how to get CXF to detect your configuration file.

The tlsClientParameters element

The TLSClientParameters are listed here and here .

Attribute	Description
`certConstraints`	Certificate Constraints specification.
`cipherSuites`	CipherSuites that will be supported. JVM defaults if not specified.
`cipherSuitesFilter`	filters of the supported CipherSuites that will be supported and used if available.
`disableCNcheck`	Indicates whether that the hostname given in the HTTPS URL will be checked against the service's Common Name (CN) given in its certificate during SOAP client requests, and failing if there is a mismatch. If set to `true` (**not recommended for production use**), such checks will be bypassed. That will allow you, for example, to use a URL such as `localhost` during development. Default is false.
`jsseProvider`	JSSE provider name. JVM default if not specified.
`keyManagers`	Key Managers to hold X509 certificates. JVM defaults used if not specified.
`secureRandom-Parameters`	SecureRandom specification. JVM default if not specified.
`secureSocketProtocol`	Protocol Name. Most common example are "SSL", "TLS" (default) or "TLSv1".

Attribute	Description
trustManagers	TrustManagers to validate peer X509 certificates. JVM default used if not specified.
useHttpsURL-Connection-DefaultSslSocket-Factory	specifies if HttpsURL-Connection.getDefault-SSLSocketFactory() should be used to create https connections. If true, jsseProvider, secureSocket-Protocol, trustManagers, keyManagers, secureRandom, cipherSuites, and cipherSuitesFilter configuration parameters are ignored. Default is false.
useHttpsURL-Connection-DefaultHostname-Verifier	This attribute specifies if HttpsURL-Connection.getDefault-HostnameVerifier() should be used to create https connections. If true, the disableCNCheck configuration parameter is ignored. Default is false.

Note : disableCNcheck is a parameterized boolean, you can use a fixed variable true | false as well as a Spring externalized property variable (e.g. ${disable-https-hostname-verification }) or a Spring expression (e.g. #{systemProperties['dev-mode'] }).

Sample :

Example 2.22. HTTP conduit configuration disabling HTTP URL hostname verification (usage of localhost, etc)

```
...
 <http-conf:conduit
    name="{http://example.com/}HelloServicePort.http-conduit">

   <!-- deactivate HTTPS url hostname verification (localhost, etc)    -->
   <!-- WARNING ! disableCNcheck=true should NOT be used in production -->
   <http-conf:tlsClientParameters disableCNcheck="true" />
   ...
 </http-conf:conduit>
 ...
```

Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP client are defined in the namespace http://cxf.apache.org/transports/http/configuration . It is commonly referred to using the prefix http-conf . In order to use the HTTP configuration elements you will need to add the line shown below to the definitions element of your endpoint's WSDL document.

Example 2.23. HTTP Consumer WSDL Element's Namespace

```
<definitions ...
   xmlns:http-conf="http://cxf.apache.org/transports/http/configuration
```

The client element

The http-conf:client element is used to specify the connection properties of an HTTP client in a WSDL document. The http-conf:client element is a child of the WSDL port element. It has the same attributes as the client element used in the configuration file.

Example

The example below shows a WSDL fragment that configures an HTTP clientto specify that it will not interact with caches.

Example 2.24. WSDL to Configure an HTTP Consumer Endpoint

```
<service ...>
  <port ...>
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

Using java code

How to configure the HTTPConduit for the SOAP Client?

First you need get the HTTPConduit from the Proxy object or Client, then you can set the HTTPClientPolicy , AuthorizationPolicy, ProxyAuthorizationPolicy, TLSClientParameters , and/or HttpBasicAuthSupplier .

```
import org.apache.cxf.endpoint.Client;
  import org.apache.cxf.frontend.ClientProxy;
  import org.apache.cxf.transport.http.HTTPConduit;
  import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;
  ...

  URL wsdl = getClass().getResource("wsdl/greeting.wsdl");
  SOAPService service = new SOAPService(wsdl, serviceName);
  Greeter greeter = service.getPort(portName, Greeter.class);

  // Okay, are you sick of configuration files ?
  // This will show you how to configure the http conduit dynamically
  Client client = ClientProxy.getClient(greeter);
  HTTPConduit http = (HTTPConduit) client.getConduit();

  HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();

  httpClientPolicy.setConnectionTimeout(36000);
  httpClientPolicy.setAllowChunking(false);
  httpClientPolicy.setReceiveTimeout(32000);

  http.setClient(httpClientPolicy);

  ...
  greeter.sayHi("Hello");
```

How to override the service address ?

If you are using JAXWS API to create the proxy obejct, here is an example which is complete JAX-WS compliant code

```
URL wsdlURL = MyService.class.getClassLoader
```

```
            .getResource ("myService.wsdl");
   QName serviceName = new QName("urn:myService", "MyService");
   MyService service = new MyService(wsdlURL, serviceName);
   ServicePort client = service.getServicePort();
   BindingProvider provider = (BindingProvider)client;
   // You can set the address per request here
   provider.getRequestContext().put(
        BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        "http://my/new/url/to/the/service");
```

If you are using CXF ProxyFactoryBean to create the proxy object , you can do like this

```
JaxWsProxyFactoryBean proxyFactory = new JaxWsProxyFactoryBean();
   poxyFactory.setServiceClass(ServicePort.class);
   // you could set the service address with this method
   proxyFactory.setAddress("theUrlyouwant");
   ServicePort client = (ServicePort) proxyFactory.create();
```

Here is another way which takes advantage of JAXWS's Service.addPort() API

```
URL wsdlURL = MyService.class.getClassLoader.getResource(
      "service2.wsdl");
   QName serviceName = new QName("urn:service2", "MyService");
   QName portName = new QName("urn:service2", "ServicePort");
   MyService service = new MyService(wsdlURL, serviceName);
   // You can add whatever address as you want
   service.addPort(portName, "http://schemas.xmlsoap.org/soap/",
      "http://the/new/url/myService");
   // Passing the SEI class that is generated by wsdl2java
   ServicePort proxy = service.getPort(portName, SEI.class);
```

Client Cache Control Directives

The following table lists the cache control directives supported by an HTTP client.

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, it means the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that will be still be fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.

Directive	Behavior
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

A Note About Chunking

There are two ways of putting a body into an HTTP stream:

- The "standard" way used by most browsers is to specify a Content-Length header in the HTTP headers. This allows the receiver to know how much data is coming and when to stop reading. The problem with this approach is that the length needs to be pre-determined. The data cannot be streamed as generated as the length needs to be calculated upfront. Thus, if chunking is turned off, we need to buffer the data in a byte buffer (or temp file if too large) so that the Content-Length can be calculated.
- Chunked - with this mode, the data is sent to the receiver in chunks. Each chunk is preceded by a hexidecimal chunk size. When a chunk size is 0, the receiver knows all the data has been received. This mode allows better streaming as we just need to buffer a small amount, up to 8K by default, and when the buffer fills, write out the chunk.

In general, Chunked will perform better as the streaming can take place directly. HOWEVER, there are some problems with chunking:

- Many proxy servers don't understand it, especially older proxy servers. Many proxy servers want the Content-Length up front so they can allocate a buffer to store the request before passing it onto the real server.
- Some of the older WebServices stacks also have problems with Chunking. Specifically, older versions of .NET.

If you are getting strang errors (generally not soap faults, but other HTTP type errors) when trying to interact with a service, try turning off chunking to see if that helps.

Authentication

Basic Authentication sample:

```
<conduit name="{http://example.com/}HelloWorldServicePort.http-conduit"
       xmlns="http://cxf.apache.org/transports/http/configuration">
   <authorization>
      <AuthorizationType>Basic</AuthorizationType>
      <UserName>myuser</UserName>
      <Password>mypasswd</Password>
   </authorization>
</conduit>
```

For Digest Authentication, use the same as above but with AuthorizationType value of `Digest`.

Authorization can also be supplied dynamically, by implementing the `org.apache.cxf.transport.http.auth.HttpAuthSupplier` interface or another interface which extends it. The main method this interface provides is:

```
public String getAuthorization(AuthorizationPolicy authPolicy,
   URL currentURL, Message message, String fullHeader);
```

With this method you'll need to supply the HttpAuthPolicy, the service URL, the CXF message and the full Authorization header (what the server sent in its last response). With the latter value multi-phase authentications can be implemented. For a simple implementation check the the `org.apache.cxf.transport.http.auth.DefaultBasicAuthSupplier` class. On the conduit above, declare your implementation class in an `AuthSupplier` element for CXF to use it.

Spnego Authentication (Kerberos)

Starting with CXF 2.4.0 CXF supports Spnego authentication using the standard AuthPolicy mechanism. Spnego is activated by setting the AuthPolicy.authorizationType to 'Negotiate'. If userName is left blank then single sign on is used with the TGT from e.g. Windows Login. If userName is set then a new LoginContext is established and the ticket is created out of this. By default the SpnegoAuthSupplier uses the OID for Spnego. Some servers require the OID for Kerberos. This can be activated by setting the contextual property auth.spnego.useKerberosOid to 'true'.

Kerberos Config: Make sure that krb5.conf/krb5.ini is configured correctly for the Kerberos realm you want to authenticate against and supply it to your application by setting the java.security.krb5.conf system property

Login Config: Create a file login.conf and supply it to CXF using the System property java.security.auth.login.config. The file should contain:

```
CXFClient {
com.sun.security.auth.module.Krb5LoginModule     //
   required client=TRUE useTicketCache=true;
};
```

Sample config: Make sure the Authorization element contains the same name as the Section in the login.conf (here: CXFClient).

```
<!-- HTTP conduit configuration for spnego with single sign on -->
...
<conduit name="{http://example.com/}HelloWorldServicePort.http-conduit"
    xmlns="http://cxf.apache.org/transports/http/configuration">
<authorization>
<AuthorizationType>Negotiate</AuthorizationType>
<Authorization>CXFClient</Authorization>
</authorization>
</conduit>
...
```

You can use UserName and Password in the above xml config if you want to log in explicitly. If you want to use the cached Ticket Granting Ticket then do not supply them. On Windows you will also have to make sure you allow the TGT to be used in Java. See: http://www.javaactivedirectory.com/?page_id=93 for more information.

```
<!-- Switching to Kerberos OID instead of Spnego -->
...
<jaxws:client>
<jaxws:properties>
<entry key="auth.spnego.useKerberosOid" value="true"/>
</jaxws:properties>
</jaxws:client>
```

```
...
```

NTLM Authentication

On Java 6, NTLM authentication is built into the Java runtime and you don't need to do anything special.

Next, you need to configure jcifs to use the correct domains, wins servers, etc... Notice that the bit which sets the username/password to use for NTLM is commented out. If credentials are missing jcifs will use the underlying NT credentials.

```
//Set the jcifs properties
jcifs.Config.setProperty("jcifs.smb.client.domain", "ben.com");
jcifs.Config.setProperty("jcifs.netbios.wins", "xxx.xxx.xxx.xxx");
jcifs.Config.setProperty("jcifs.smb.client.soTimeout",
"300000"); //5 minutes
jcifs.Config.setProperty("jcifs.netbios.cachePolicy",
   "1200"); //20 minutes
//jcifs.Config.setProperty("jcifs.smb.client.username", "myNTLogin");
//jcifs.Config.setProperty("jcifs.smb.client.password", "secret");

//Register the jcifs URL handler to enable NTLM
jcifs.Config.registerSmbURLHandler();
```

Finally, you need to setup the CXF client to turn off chunking. The reason is that the NTLM authentication requires a 3 part handshake which breaks the streaming.

```
//Turn off chunking so that NTLM can occur
Client client = ClientProxy.getClient(port);
HTTPConduit http = (HTTPConduit) client.getConduit();
HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();
httpClientPolicy.setConnectionTimeout(36000);
httpClientPolicy.setAllowChunking(false);
http.setClient(httpClientPolicy);
```

2.5.1.2. Server HTTP Transport

HTTP server endpoints can specify a number of HTTP connection attributes including if it will honor keep alive requests, how it interacts with caches, and how tolerant it is of errors in communicating with a consumer.

A server endpoint can be configured using two mechanisms:

- Configuration
- WSDL

Using Configuration

Namespace

The elements used to configure an HTTP provider endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix `http-conf` . In order to use the HTTP configuration elements you will need to add the lines shown below to the beans element of your

endpoint's configuration file. In addition, you will need to add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 2.25. Adding the Configuration Namespace

```
<beans ...
   xmlns:http-conf="http://cxf.apache.org/transports/http/configuration
   ...
       xsi:schemaLocation="...
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          ...>
```

The destination element

You configure an HTTP server endpoint using the `http-conf:destination` element and its children. The `http-conf:destination` element takes a single attribute, `name` , the specifies the WSDL port element that corresponds to the endpoint. The value for the `name` attribute takes the form *portQName* `.http-destination` . The example below shows the `http-conf:destination` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment `<port binding="widgetSOAPBinding" name="widgetSOAPPort>` if the endpoint's target namespace was http://widgets.widgetvendor.net .

Example 2.26. http-conf:destination Element

```
...
  <http-conf:destination name=
     "{http://widgets/widgetvendor.net}widgetSOAPPort.http-destination">
     ...
  </http-conf:destination>
...
```

The `http-conf:destination` element has a number of child elements that specify configuration information. They are described below.

Element	Description
`http-conf:server`	Specifies the HTTP connection properties.
`http-conf:contextMatchStrategy`	Specifies the parameters that configure the context match strategy for processing HTTP requests.
`http-conf:fixedParameterOrder`	Specifies whether the parameter order of an HTTP request handled by this destination is fixed.

The server element

The `http-conf:server` } element is used to configure the properties of a server's HTTP connection. Its attributes, described below, specify the connection's properties.

Attribute	Description
`ReceiveTimeout`	Sets the length of time, in milliseconds, the server tries to receive a request before the connection times out. The default is 30000. The specify that the server will not timeout use 0.
`SuppressClient-SendErrors`	Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. The default is `false` ; exceptions are thrown on encountering errors.

Attribute	Description
SuppressClient-ReceiveErrors	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a client. The default is `false` ; exceptions are thrown on encountering errors.
HonorKeepAlive	Specifies whether the server honors requests for a connection to remain open after a response has been sent. The default is `true` ; keep-alive requests are honored.
RedirectURL	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to 302 and the status description is set to Object Moved. The value is used as the value of the HTTP `RedirectURL` property.
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a server to a client.
ContentLocation	Sets the URL where the resource being sent in a response is located.
ContentType	Specifies the media type of the information being sent in a response. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP `ContentType` location.
ContentEncoding	Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include zip, gzip, compress, deflate, and identity. This value is used as the value of the HTTP `ContentEncoding` property.
ServerType	Specifies what type of server is sending the response. Values take the form program-name/version. For example, Apache/1.2.5.

Example

The example below shows a the configuration for an HTTP service provider endpoint that honors keep alive requests and suppresses all communication errors.

Example 2.27. HTTP Service Provider Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf=
           "http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="
           http://cxf.apache.org/transports/http/configuration
           http://cxf.apache.org/schemas/configuration/http-conf.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:destination name=
      "{http://apache.org/hello_soap_http}SoapPort.http-destination">
    <http-conf:server SuppressClientSendErrors="true"
                      SuppressClientReceiveErrors="true"
                      HonorKeepAlive="true" />
  </http-conf:destination>
</beans>
```

Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP server endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration . It is commonly refered to using the prefix `http-conf` . In order to use the HTTP configuration elements you will need to add the line shown below to the `definitions` element of your endpoint's WSDL document.

Example 2.28. HTTP Provider WSDL Element's Namespace

```
<definitions ...
   xmlns:http-conf="http://cxf.apache.org/transports/http/configuration
```

The server element

The `http-conf:server` element is used to specify the connection properties of an HTTP server in a WSDL document. The `http-conf:server` element is a child of the WSDL port element. It has the same attributes as the `server` element used in the configuration file.

Example

The example below shows a WSDL fragment that configures an HTTP server endpoint to specify that it will not interact with caches.

Example 2.29. WSDL to Configure an HTTP Service Provider Endpoint

```
<service ...>
  <port ...>
    <soap:address ... />
    <http-conf:server CacheControl="no-cache" />
  </port>
</service>
```

Server Cache Control Directives

The table below lists the cache control directives supported by an HTTP server.

Directive	Behavior
`no-cache`	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`public`	Any cache can store the response.
`private`	Public (shared) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with

Directive	Behavior
	this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`no-store`	Caches must not store any part of response or any part of the request that invoked it.
`no-transform`	Caches must not modify the media type or location of the content in a response between a server and a client.
`must-revalidate`	Caches must revaildate expired entries that relate to a response before that entry can be used in a subsequent response.
`proxy-revalidate`	Means the same as must-revalidate, except that it can only be enforced on shared caches and is ignored by private unshared caches. If using this directive, the public cache directive must also be used.
`max-age`	Clients can accept a response whose age is no greater that the specified number of seconds.
`s-max-age`	Means the same as max-age, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by s-max-age overrides the age specified by max-age. If using this directive, the proxy-revalidate directive must also be used.
`cache-extension`	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

2.5.1.3. Servlet Transport

To create services that use this transport you can either use the CXF APIs (for example, see JAX-WS) or create an XML file which registers services for you.

Publishing an endpoint from XML

CXF uses Spring to provide XML configuration of services. This means that first we'll want to load Spring via a Servlet listener and tell it where our XML configuration file is:

Next, you'll need to add CXFServlet to your web.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:com/acme/ws/services.xml
    </param-value>
 </context-param>

  <listener>
     <listener-class>
```

```
        org.springframework.web.context.ContextLoaderListener
     </listener-class>
  </listener>


  <servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
       org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Alternatively, you can point to the configuration file using a CXFServlet init parameter :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

  <servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
        org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <init-param>
      <param-name>config-location</param-name>
      <param-value>/WEB-INF/beans.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>


</web-app>
```

The next step is to actually write the configuration file:

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

```
   http://cxf.apache.org/schemas/jaxws.xsd
   http://cxf.apache.org/jaxrs
   http://cxf.apache.org/schemas/jaxrs.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
  <import resource=
     "classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

  <jaxws:endpoint id="greeter"
                  implementor="org.apache.hello_soap_http.GreeterImpl"
                  address="/Greeter1"/>

  <jaxrs:server id="greeterRest"
                  serviceClass="org.apache.hello_soap_http.GreeterImpl"
                  address="/GreeterRest"/>

</beans>
```

Here we're creating a JAX-WS endpoint based on our implementation class, GreeterImpl.

NOTE: We're publishing endpoints "http://localhost/mycontext/services/Greeter1" and "http://localhost/mycontext/services/GreeterRest", but we set jaxws:endpoint/@address and jaxrs:server/@address to relative values such as "/Greeter1" "/GreeterRest".

Redirecting requests and serving the static content

Starting from CXF 2.2.5 it is possible to configure CXFServlet to redirect current requests to other servlets or serve the static resources.

"redirects-list" init parameter can be used to provide a space separated list of URI patterns; if a given request URI matches one of the patterns then CXFServlet will try to find a RequestDispatcher using the pathInfo of the current HTTP request and will redirect the request to it.

"redirect-servlet-path" can be used to affect a RequestDispatcher lookup, if specified then it will concatenated with the pathInfo of the current request.

"redirect-servlet-name" init parameter can be used to enable a named RequestDispatcher look-up, after one of the URI patterns in the "redirects-list" has matched the current request URI.

"static-resources-list" init parameter can be used to provide a space separated list of static resource such as html, css, or pdf files which CXFServlet will serve directly.

One can have requests redirected to other servlets or JSP pages.

CXFServlets serving both JAXWS and JAXRS based endpoints can avail of this feature.

For example, please see this web.xml .

The "http://localhost:9080/the/bookstore1/books/html/123" request URI will initially be matched by the CXFServlet given that it has a more specific URI pattern than the RedirectCXFServlet. After a current URI has reached a jaxrs:server endpoint, the response will be redirected by the JAXRS RequestDispatcherProvider to a "/book.html" address, see "dispatchProvider1" bean here .

Next, the request URI "/book.html" will be handled by RedirectCXFServlet. Note that a uri pattern can be a regular expression. This servlet redirects the request further to a RequestDispatcher capable of handling a "/static/book.html".

Finally, DefaultCXFServlet serves a requested book.html.

Publishing an endpoint with the API

Once your Servlet is registered in your web.xml, you should set the default bus with CXFServlet's bus to make sure that CXF uses it as it's HTTP Transport. Simply publish with the related path "Greeter" and your service should appear at the address you specify:

```
import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.servlet.CXFServlet;
.....
// cxf is the instance of the CXFServlet, you could also get
// this instance by extending the CXFServlet
Bus bus = cxf.getBus();
BusFactory.setDefaultBus(bus);
Endpoint.publish("/Greeter", new GreeterImpl());
```

The one thing you must ensure is that your CXFServlet is set up to listen on that path. Otherwise the CXFServlet will never receive the requests.

NOTE:

Endpoint.publish(...) is a JAX-WS API for publishing JAX-WS endpoints. Thus, it would require the JAX-WS module and API's to be present. If you are not using JAX-WS or want more control over the published endpoint properties, you should replace that call with the proper calls to the appropriate ServerFactory.

Since CXFServlet know nothing about the web container listen port and the application context path, you need to specify the relate path instead of full http address.

Using the servlet transport without Spring

Some user who doesn't want to touch any Spring stuff could also publish the endpoint with CXF servlet transport. First you should extends the CXFNonSpringServlet and then override the method loadBus which below codes:

```
import javax.xml.ws.Endpoint;
...

    @Override
    public void loadBus(ServletConfig servletConfig)
       throws ServletException {
        super.loadBus(servletConfig);

        // You could add the endpoint publish codes here
        Bus bus = cxf.getBus();
        BusFactory.setDefaultBus(bus);
        Endpoint.publish("/Greeter", new GreeterImpl());

        // You can als use the simple frontend API to do this
        ServerFactoryBean factroy = new ServerFactoryBean();
        factory.setBus(bus);
        factory.setServiceClass(GreeterImpl.class);
        factory.setAddress("/Greeter");
        factory.create();
```

```
    }
```

If you are using the Jetty as the embedded servlet engine, you could publish endpoint like this:

```
import javax.xml.ws.Endpoint;
...

        // Setup the system properties to use
        // the CXFBusFactory not the SpringBusFactory
        String busFactory =
           System.getProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME);
        System.setProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME,
           "org.apache.cxf.bus.CXFBusFactory");
        try {
            // Start up the jetty embedded server
            httpServer = new Server(9000);
            ContextHandlerCollection contexts
               = new ContextHandlerCollection();
            httpServer.setHandler(contexts);

            Context root = new Context(contexts, "/", Context.SESSIONS);

            CXFNonSpringServlet cxf = new CXFNonSpringServlet();
            ServletHolder servlet = new ServletHolder(cxf);
            servlet.setName("soap");
            servlet.setForcedPath("soap");
            root.addServlet(servlet, "/soap/*");

            httpServer.start();

            Bus bus = cxf.getBus();
            setBus(bus);
            BusFactory.setDefaultBus(bus);
            GreeterImpl impl = new GreeterImpl();
            Endpoint.publish("/Greeter", impl);
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            // clean up the system properties
            if (busFactory != null) {
                System.setProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME,
                   busFactory);
            } else {
                System.clearProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME);
            }
        }
```

Accessing the MessageContext and/or HTTP Request and Response

Sometimes you'll want to access more specific message details in your service implementation. One example might be accessing the actual request or response object itself. This can be done using the WebServiceContext object.

First, declare a private field for the WebServiceContext in your service implementation, and annotate it as a resource:

```
@Resource
private WebServiceContext context;
```

Then, within your implementing methods, you can access the MessageContext, HttpServletRequest, and HttpServletResponse as follows:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.transport.http.AbstractHTTPDestination;
...

MessageContext ctx = context.getMessageContext();
HttpServletRequest request = (HttpServletRequest)
   ctx.get(AbstractHTTPDestination.HTTP_REQUEST);
HttpServletResponse response = (HttpServletResponse)
   ctx.get(AbstractHTTPDestination.HTTP_RESPONSE);
```

Of course, it is always a good idea to program defensively if using transport-specific entities like the HttpServletRequest and HttpServletResponse. If the transport were changed (for instance to the JMS transport), then these values would likely be null.

2.5.2. JMS Transport

CXF provides a transport plug-in that enables endpoints to use Java Message Service (JMS) queues and topics. CXF's JMS transport plug-in uses the Java Naming and Directory Interface (JNDI) to locate and obtain references to the JMS provider that brokers for the JMS destinations. Once CXF has established a connection to a JMS provider, CXF supports the passing of messages packaged as either a JMS `ObjectMessage` or a JMS `TextMessage` .

2.5.2.1. Easier configuration using the new JMSConfigFeature

Starting with CXF 2.0.9 and 2.1.3 there is a new easier and more flexible configuration style available. See Section 2.5.2.8, “Using the JMSConfigFeature”

2.5.2.2. JMS Transport wiht SOAP over Java Message Service 1.0-Supported

Starting with the CXF 2.3 , we make some improvement on the JMS Transport to support SOAP over JMS specification . See Section 2.5.2.7, “SOAP over JMS 1.0 support” for more information.

2.5.2.3. JMS Namespaces

WSDL Namespace

The WSDL extensions for defining a JMS endpoint are defined in the namespace http://cxf.apache.org/transports/jms . In order to use the JMS extensions you will need to add the namespace definition shown below to the definitions element of your contract.

Example 2.30. JMS Extension Namespace

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

Configuration Namespaces

In order to use the JMS configuration properties you will need to add the line shown below to the `beans` element of your configuration.

Example 2.31. JMS Configuration Namespaces

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

2.5.2.4. Basic Endpoint Configuration

JMS endpoints need to know certain basic information about how to establish a connection to the proper destination. This information can be provided in one of two places: WSDL or XML configuration. The following configuration elements which are described can be used in both the client side Conduits and the server side Destinations.

Using WSDL

The JMS destination information is provided using the `jms:address` element and its child, the `jms:JMSNamingProperties` element. The `jms:address` element's attributes specify the information needed to identify the JMS broker and the destination. The `jms:JMSNamingProperties` element specifies the Java properties used to connect to the JNDI service.

The address element

The basic configuration for a JMS endpoint is done by using a `jms:address` element as the child of your service's `port` element. The `jms:address` element uses the attributes described below to configure the connection to the JMS broker.

Attribute	Description
`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which requests are sent.
`jndiReplyDestinationName`	Specifies the JNDI name bound to the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies.
`connectionUserName`	Specifies the username to use when connecting to a JMS broker.

Attribute	Description
connectionPassword	Specifies the password to use when connecting to a JMS broker.

The JMSNamingProperties element

To increase interoperability with JMS and JNDI providers, the `jms:address` element has a child element, `jms:JMSNamingProperties`, that allows you to specify the values used to populate the properties used when connecting to the JNDI provider. The `jms:JMSNamingProperties` element has two attributes: `name` and `value`. The `name` attribute specifies the name of the property to set. The `value` attribute specifies the value for the specified property. The `jms:JMSNamingProperties` element can also be used for specification of provider specific properties. The following is a list of common JNDI properties that can be set:

- `java.naming.factory.initial`
- `java.naming.provider.url`
- `java.naming.factory.object`
- `java.naming.factory.state`
- `java.naming.factory.url.pkgs`
- `java.naming.dns.url`
- `java.naming.authoritative`
- `java.naming.batchsize`
- `java.naming.referral`
- `java.naming.security.protocol`
- `java.naming.security.authentication`
- `java.naming.security.principal`
- `java.naming.security.credentials`
- `java.naming.language`
- `java.naming.applet`

For more details on what information to use in these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Using a named reply destination

By default, CXF endpoints using JMS create a temporary queue for sending replies back and forth. You can change this behavior by setting the `jndiReplyDestinationName` attribute in the endpoint's contract. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the `ReplyTo` field of all outgoing requests. A service endpoint will use the value of the `jndiReplyDestinationName` attribute as the location for placing replies if there is no destination specified in the request's `ReplyTo` field.

As of CXF 2.1.3 and 2.0.9 a static reply queue can not be shared by several instances of the service client. Please use a dynamic reply queue or different queue names per instance instead. (See discussion on the mailing list)

The following example shows an example of a JMS WSDL port specification.

Example 2.32. JMS WSDL Port Specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
         jndiDestinationName="dynamicQueues/test.cxf.jmstransport">
      <jms:JMSNamingProperty name="java.naming.factory.initial"
         value="org.apache.activemq.jndi.ActiveMQInitialContextFactory"/>
      <jms:JMSNamingProperty name="java.naming.provider.url"
         value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

Using Configuration

In addition to using the WSDL file to specify the connection information for a JMS endpoint, you can also supply it in the endpoint's XML configuration. The information in the configuration file will override the information in the endpoint's WSDL file.

Configuration elements

You configure a JMS endpoint using one of the following configuration elements:

- **jms:conduit** : The `jms:conduit` element contains the configuration for a consumer endpoint. It has one attribute, `name` , whose value takes the form

  `{WSDLNamespace}WSDLPortName.jms-conduit`

  .

- **jms:destination** : The `jms:destination` element contains the configuration for a provider endpoint. It has one attribute, `name` , whose value takes the form

  `{WSDLNamespace}WSDLPortName.jms-destination`

  .

The address element

JMS connection information is specified by adding a `jms:address` child to the base configuration element. The `jms:address` element used in the configuration file is identical to the one used in the WSDL file. Its attributes are listed in the address element's attribute table[95] . Like the jms:address element in the WSDL file, the jms:address configuration element also has a `jms:JMSNamingProperties` child element that is used to specify additional information used to connect to a JNDI provider.

Example 2.33. Addressing Information in a Configuration File

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ct="http://cxf.apache.org/configuration/types"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://cxf.apache.org/jaxws
           http://cxf.apache.org/schemas/jaxws.xsd
           http://cxf.apache.org/transports/jms
           http://cxf.apache.org/schemas/configuration/jms.xsd">
<jms:conduit
    name="{http://cxf.apache.org/jms_endpt}HelloJMSPort.jms-conduit">
  <jms:address destinationStyle="queue"
               jndiConnectionFactoryName="myConnectionFactory"
               jndiDestinationName="myDestination"
               jndiReplyDestinationName="myReplyDestination"
               connectionUserName="testUser"
               connectionPassword="testPassword">
    <jms:JMSNamingProperty name="java.naming.factory.initial"
        value="org.apache.cxf.transport.jms.MyInitialContextFactory"/>
    <jms:JMSNamingProperty name="java.naming.provider.url"
        value="tcp://localhost:61616"/>
  </jms:address>
</jms:conduit>
</beans>
```

Consumer Endpoint Configuration

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS `ObjectMessage` or a JMS `TextMessage` . When using an `ObjectMessage` the consumer endpoint uses a `byte[]` as the method for storing data into and retrieving data from the JMS message body. When messages are sent, the message data, including any formating information, is packaged into a `byte[]` and placed into the JMS message body before it is placed on the wire. When messages are received, the consumer endpoint will attempt to unmarshall the data stored in the JMS body as if it were packed in a `byte[]` .

When using a `TextMessage` , the consumer endpoint uses a string as the method for storing and retrieving data from the JMS message body. When messages are sent, the message information, including any format-specific information, is converted into a string and placed into the JMS message body. When messages are received the consumer endpoint will attempt to unmashall the data stored in the JMS message body as if it were packed into a string.

When native JMS applications interact with CXF consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the CXF contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as `TextMessage` , the receiving JMS application will get a text message containing all of the SOAP envelope information.

Consumer endpoint can be configured by both XML configuration and via WSDL.

Using Configuration

Specifying the message type

You can specify the message type supported by the consumer endpoint using a `jms:runtimePolicy` element that has a single attribute:

- `messageType` - Specifies how the message data will be packaged as a JMS message. `text` specifies that the data will be packaged as a `TextMessage` . `binary` specifies that the data will be packaged as an `ObjectMessage` .

The following example shows a configuration entry for configuring a JMS consumer endpoint.

Example 2.34. Configuration for a JMS Consumer Endpoint

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ct="http://cxf.apache.org/configuration/types"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd"
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd
          http://cxf.apache.org/transports/jms
          http://cxf.apache.org/schemas/configuration/jms.xsd">
...
<jms:conduit
   name="{http://cxf.apache.org/jms_endpt}HelloJMSPort.jms-conduit">
  <jms:address ... >
    ...
  </jms:address>
  <jms:runtimePolicy messageType="binary"/>
...
</jms:conduit>
...
</beans>
```

The id on the `jms:conduit` is in the form of { `WSDLNamespace}WSDLPortName.jms-conduit` . This provides CXF with the information so that it can associate the configuration with your service's endpoint.

Using WSDL

The type of messages accepted by a JMS consumer endpoint is configured using the optional `jms:client` element. The `jms:client` element is a child of the WSDL port element and has one attribute:

- `messageType` - Specifies how the message data will be packaged as a JMS message. `text` specifies that the data will be packaged as a `TextMessage` . `binary` specifies that the data will be packaged as an ObjectMessage.

2.5.2.5. Service Endpoint Configuration

JMS service endpoints have a number of behaviors that are configurable in the contract. These include:

- how messages are correlated
- the use of durable subscriptions
- if the service uses local JMS transactions
- the message selectors used by the endpoint

Service endpoints can be configure in one of two ways:

- Configuration
- WSDL

Using Configuration

Specifying configuration data

Using the `jms:destination` elements you can configure your service's endpoint. You can specify the service endpoint's behaviors using the `jms:runtimePolicy` element that has a the following attributes:

Attribute	Description
`useMessageIDAsCorrelationID`	Specifies whether the JMS broker will use the message ID to correlate messages. The default is `false` .
`durableSubscriberName`	Specifies the name used to register a durable subscription.
`messageSelector`	Specifies the string value of a message selector to use. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false` .

The following example shows a CXF configuration entry for configuring a JMS service endpoint.

Example 2.35. Configuration for a JMS Service Endpoint

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ct="http://cxf.apache.org/configuration/types"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd"
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd
          http://cxf.apache.org/transports/jms
          http://cxf.apache.org/schemas/configuration/jms.xsd">
...
<jms:destination
   name="{http://cxf.apache.org/jms_endpt}HelloJMSPort.jms-destination">
  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:runtimePolicy messageSelector="cxf_message_selector"
                     useMessageIDAsCorrelationID="true"
                     transactional="true"
                     durableSubscriberName="cxf_subscriber" />
  ...
</jms:destination>
...
</beans>
```

Using WSDL

Service endpoint behaviors are configured using the optional `jms:server` element. The `jms:server` element is a child of the WSDL `port` element and has the following attributes:

Attribute	Description
`useMessageIDAsCorrelationID`	Specifies whether JMS will use the message ID to correlate messages. The default is `false` .
`durableSubscriberName`	Specifies the name used to register a durable subscription.
`messageSelector`	Specifies the string value of a message selector to use. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false` . Currently, this is not supported by the runtime.

2.5.2.6. JMS Runtime Configuration

In addition to configuring the externally visible aspects of your JMS endpoint, you can also configure aspects of its internal runtime behavior. There are three types of runtime configuration:

- Session pool configuration (common to both services and consumers)

- Consumer specific configuration
- Service specific configuration

Session Pool Configuration

You configure an endpoint's JMS session pool using the `jms:sessionPoolConfig` element. This property allows you to set a high and low water mark for the number of JMS sessions an endpoint will keep pooled. The endpoint is guaranteed to maintain a pool of sessions equal to the low water mark and to never pool more sessions than specified by the high water mark. The `jms:sessionPool` element's attributes, listed below, specify the high and low water marks for the endpoint's JMS session pool.

Attribute	Description
`lowWaterMark`	Specifies the minimum number of JMS sessions pooled by the endpoint. The default is 20.
`highWaterMark`	Specifies the maximum number of JMS sessions pooled by the endpoint. The default is 500.

The following example shows an example of configuring the session pool for a CXF JMS service endpoint.

Example 2.36. JMS Session Pool Configuration

```
<jms:destination
name="{http://cxf.apache.org/jms_endpit}HelloJMSPort.jms-destination">
...
  <jms:sessionPool lowWaterMark="10" highWaterMark="5000" />
</jms:destination>
```

The `jms:sessionPool` element can also be used within a `jms:conduit` .

Consumer Specific Runtime Configuration

The JMS consumer configuration allows you to specify two runtime behaviors:

- the number of milliseconds the consumer will wait for a response.
- the number of milliseconds a request will exist before the JMS broker can remove it.

You use the `jms:clientConfig` element to set JMS consumer runtime behavior. This element's attributes, listed in the following table, specify the configuration values for consumer runtime behavior.

Attribute	Description
`clientReceiveTimeout`	Specifies the amount of time, in milliseconds, that the endpoint will wait for a response before it times out and issues an exception. The default value is 2000.
`messageTimeToLive`	Specifies the amount of time, in milliseconds, that a request can remain unrecieved before the JMS broker can delete it. The default value is 0 which specifies that the message can never be deleted.

The following example shows a configuration fragment that sets the consumer endpoint's request lifetime to 500 milliseconds and its timeout value to 500 milliseconds.

Example 2.37. JMS Consumer Endpoint Runtime Configuration

```
<jms:conduit
name="{http://cxf.apache.org/jms_endpt}HelloJMSPort.jms-conduit">
...
  <jms:clientConfig clientReceiveTimeout="500"
                    messageTimeToLive="500" />
</jms:conduit>
```

Service Specific Runtime Configuration

The JMS service configuration allows you to specify to runtime behaviors:

- the amount of time a response message can remain unreceived before the JMS broker can delete it.
- the client identifier used when creating and accessing durable subscriptions.

The `jms:serverConfig` element is used to specify the service runtime configuration. This element's attributes, listed below, specify the configuration values that control the service's runtime behavior.

Attribute	Description
messageTimeToLive	Specifies the amount of time, in milliseconds, that a response can remain unread before the JMS broker is allowed to delete it. The default is 0 which specifies that the message can live forever.
durableSubscriptionClientId	Specifies the client identifier the endpoint uses to create and access durable subscriptions.

The following example shows a configuration fragment that sets the service endpoint's response lifetime to 500 milliseconds and its durable subscription client identifier to `jms-test-id` .

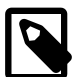
Example 2.38. JMS Service Endpoint Runtime Configuration

```
<jms:destination
  id="{http://cxf.apache.org/jms_endpt}HelloJMSPort.jms-destination">
  <jms:address ... >
    ...
  </jms:address>
  <jms:serverConfig messageTimeToLive="500"
                    durableSubscriptionClientId="jms-test-id" />
</jms:destination>
```

2.5.2.7. SOAP over JMS 1.0 support

SOAP over JMS offers an alternative messaging mechanism to SOAP over HTTP. SOAP over JMS offers more reliable and scalable messaging support than SOAP over HTTP.

SOAP over JMS specification is aimed at a set of standards for the transport of SOAP messages over JMS. The main purpose is to ensure interoperability between the implementations of different Web services vendors. CXF supports and is compliant with this specification.



"New Feature!"

The SOAP over JMS specification feature described here is a new feature for CXF 2.3.

What's new compared to the old CXF JMS Transport

SOAP over JMS transport supports most configurations of JMS Transport and provides some extensions to support the SOAP over JMS specification. SOAP over JMS Transport uses the JMS URI (jms:address, for example) to describe JMS addressing information and provides new WSDL extensions for JMS configuration.

SOAP over JMS Namespace

WSDL Namespace

The WSDL extensions for defining a JMS endpoint use a special namespace. In order to use the JMS WSDL extensions you will need to add the namespace definition shown below to the definitions element of your contract.

JMS Extension Namespace

```
xmlns:soapjms="http://www.w3.org/2008/07/soap/bindings/JMS/"
```

JMS URI

JMS endpoints need to know the address information for establishing connections to the proper destination. SOAP over JMS implements the URI Scheme for Java Message Service 1.0 .

This URI scheme starts with "jms:jndi:" plus a JNDI name for a Destination. Since interaction with some resources may require JNDI contextual information or JMS header fields and properties to be specified as well, the "jndi" variant of the "jms" URI scheme includes support for supplying this additional JNDI information as query parameters.

CXF supports three variants, " **jndi** ", " **queue** ", and " **topic** ". For example:

```
jms:jndi:SomeJndiNameForDestination?jndiInitialContextFactory=
               com.example.jndi.JndiFactory&priority=3
jms:queue:ExampleQueueName?timeToLive=1000
```

Properties are as follows:

Property	DefaultValue	Description
deliveryMode	PERSISTENT	NON_PERSISTENT messages will kept only in memory PERSISTENT messages will be saved to disk
jndiConnectionFactoryName		Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
jndiInitialContextFactory		Specifies the fully qualified Java class name of the "InitialContextFactory" implementation class to use.
jndiURL		Specifies the JNDI provider URL
replyToName		Specifies the JNDI name bound to the JMS destinations where replies are sent.
priority	4	Priority for the messages. See your JMS provider documentation for details

Property	DefaultValue	Description
timeToLive	0	Time (in ms) after which the message will be discarded by the jms provider
Additional JNDI Parameters		Additional parameters for a JNDI provider. A custom parameter name must start with the prefix "jndi-".

For more details about these attributes, please check out the JMS URI specification .

WSDL Extension

Various JMS properties may be set in three places in the WSDL — the binding, the service, and the port. Values specified at the service will propagate to all ports. Values specified at the binding will propagate to all ports using that binding. For example, if the **jndiInitialContextFactory** is indicated for a service, it will be used for all of the port elements it contains.

Field	DefaultValue	Description
deliveryMode	PERSISTENT	NON_PERSISTENT messages will only be kept in memory PERSISTENT messages will be saved to disk
jndiConnectionFactoryName		Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
jndiInitialContextFactory		Specifies the fully qualified Java class name of the "InitialContextFactory" implementation class to use.
jndiURL		Specifies the JNDI provider URL
replyToName		Specifies the JNDI name bound to the JMS destinations where replies are sent.
priority	4	Priority for the messages. See your JMS provider doc for details
timeToLive	0	Time (in ms) after which the message will be discarded by the jms provider
jndiContextParameter		Additional parameters for a JNDI provider.

Here is an example:

```
<wsdl11:binding name="exampleBinding">
  <soapjms:jndiContextParameter name="name" value="value" />
<soapjms:jndiConnectionFactoryName>ConnectionFactory
</soapjms:jndiConnectionFactoryName>
<soapjms:jndiInitialContextFactory>
org.apache.activemq.jndi.ActiveMQInitialContextFactory
</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>tcp://localhost:61616
</soapjms:jndiURL>
<soapjms:deliveryMode>PERSISTENT</soapjms:deliveryMode>
<soapjms:priority>5</soapjms:priority>
  <soapjms:timeToLive>200</soapjms:timeToLive>
</wsdl11:binding>

<wsdl11:service name="exampleService">
  <soapjms:jndiInitialContextFactory>
```

```
    com.example.jndi.InitialContextFactory
  </soapjms:jndiInitialContextFactory>
  <soapjms:timeTolive>100</soapjms:timeToLive>
  ...
  <wsdl11:port name="quickPort" binding="tns:exampleBinding">
    ...
    <soapjms:timeToLive>10</soapjms:timeToLive>
  </wsdl11:port>
  <wsdl11:port name="slowPort" binding="tns:exampleBinding">
    ...
  </wsdl11:port>
</wsdl11:service>
```

If a property is specified at multiple levels, the setting at the most granular level takes precedence (port first, then service, then binding). In the above example, notice the timeToLive property — for the quickPort port, the value will be 10ms (specified at the port level). For the slowPort port, the value will be 100ms (specified at the service level). In this example, the setting in the binding will always be overridden.

WSDL Usage

For this example:

```
<wsdl:definitions name="JMSGreeterService"
    <wsdl:binding name="JMSGreeterPortBinding"
        type="tns:JMSGreeterPortType">
        <soap:binding style="document"
            transport="http://www.w3.org/2010/soapjms/" />
        <soapjms:jndiContextParameter name="name"
            value="value" />
        <soapjms:jndiConnectionFactoryName>ConnectionFactory
        </soapjms:jndiConnectionFactoryName>
        <soapjms:jndiInitialContextFactory>
            org.apache.activemq.jndi.ActiveMQInitialContextFactory
        </soapjms:jndiInitialContextFactory>
        <soapjms:jndiURL>tcp://localhost:61616
        </soapjms:jndiURL>
        <soapjms:deliveryMode>PERSISTENT</soapjms:deliveryMode>
        <soapjms:priority>5</soapjms:priority>
        <soapjms:timeToLive>1000</soapjms:timeToLive>
        <wsdl:operation name="greetMe">
            <soap:operation soapAction="test" style="document" />
            <wsdl:input name="greetMeRequest">
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="greetMeResponse">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
        <wsdl:service name="JMSGreeterService">
        <soapjms:jndiConnectionFactoryName>ConnectionFactory
        </soapjms:jndiConnectionFactoryName>
        <soapjms:jndiInitialContextFactory>
            org.apache.activemq.jndi.ActiveMQInitialContextFactory
        </soapjms:jndiInitialContextFactory>
        <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
            <soap:address location=
```

```
                "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

- The transport URI (http://www.w3.org/2010/soapjms/) is defined in the <soap:binding>.
- The jms: URI is defined in the <soap:address>
- The extension properties are in the <soap:binding>

Publishing a service with the JAVA API

Developers who don't wish to modify the WSDL file can also publish the endpoint information using Java code. For CXF's SOAP over JMS implementation you can write the following:

```
// You just need to set the address with JMS URI
String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
    + "?jndiInitialContextFactory"
    + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
    + "&jndiConnectionFactoryName=ConnectionFactory&jndiURL"
    + "=tcp://localhost:61500";
Hello implementor = new HelloImpl();
JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();
svrFactory.setServiceClass(Hello.class);
svrFactory.setAddress(address);
// And specify the transport ID with SOAP over JMS specification
svrFactory.setTransportId(
   JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPORTID);
svrFactory.setServiceBean(implementor);
svrFactory.create();
```

NOTE: Before you start the server, you need to make sure the JMS broker is stared, you can find some useful code of starting the JMS broker here.

Error formatting macro: snippet: java.lang.IllegalArgumentException: Invalid url: must begin with a configured prefix.

Consume the service with the API

Sample code to consume a SOAP-over-JMS service is as follows:

```
public void invoke() throws Exception {
        // You just need to set the address with JMS URI
        String address =
            "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
            + "?jndiInitialContextFactory"
            + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
            + "&jndiConnectionFactoryName=ConnectionFactory&jndiURL="
            + "tcp://localhost:61500";
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
        // And specify the transport ID with SOAP over JMS specification
        factory.setTransportId(
           JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPORTID);
        factory.setServiceClass(Hello.class);
        factory.setAddress(address);
        Hello client = (Hello)factory.create();
```

```
        String reply = client.sayHi(" HI");
        System.out.println(reply);
    }
```

Even if you want to use the 'queue' or 'topic' variants and avoid dealing with JNDI directly, you still have to specify the two factory parameters in the address:

```
svrFactory.setAddress(
    "jms:queue:test.cxf.jmstransport.queue?timeToLive=1000"
    + "&jndiConnectionFactoryName=ConnectionFactory"
    + "&jndiInitialContextFactory"
    + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory");
```

Differences between the SOAP over JMS and the CXF old JMS transport implementation

There are some differences between the SOAP over JMS and the previous CXF over JMS transport implementation.

1. The JMS Messages sent by SOAP over JMS transport implementation are in accordance with the SOAP over JMS specification, allowing CXF to interoperate with other SOAP over JMS implementations.
2. Additional techniques are provided for configuring SOAP over JMS.
3. The new implementation provides more sophisticated error handling for the SOAP over JMS messages.

2.5.2.8. Using the JMSConfigFeature

In older CXF version the JMS transport is configured by defining a JMSConduit or JMSDestination. Starting with CXF 2.0.9 and 2.1.3 the JMS transport includes an easier configuration option that is more conformant to the spring dependency injection. Additionally the new configuration has much more options. For example it is not necessary anymore to use JNDI to resolve the connection factory. Instead it can be defined in the spring config.

The following example configs use the p-namespace from spring 2.5 but the old spring bean style is also possible.

Inside a features element the JMSConfigFeature can be defined.

```
<jaxws:client id="CustomerService"
    xmlns:customer="http://customerservice.example.com/"
    serviceName="customer:CustomerServiceService"
    endpointName="customer:CustomerServiceEndpoint" address="jms://"
    serviceClass="com.example.customerservice.CustomerService">
    <jaxws:features>
        <bean xmlns="http://www.springframework.org/schema/beans"
            class="org.apache.cxf.transport.jms.JMSConfigFeature"
            p:jmsConfig-ref="jmsConfig"/>
    </jaxws:features>
</jaxws:client>
```

In the above example it references a bean "jmsConfig" where the whole configuration for the JMS transport can be done.

A jaxws Endpoint can be defined in the same way:

```
<jaxws:endpoint
    xmlns:customer="http://customerservice.example.com/"
    id="CustomerService"
    address="jms://"
```

```
    serviceName="customer:CustomerServiceService"
    endpointName="customer:CustomerServiceEndpoint"
    implementor="com.example.customerservice.impl.CustomerServiceImpl">
    <jaxws:features>
        <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
            p:jmsConfig-ref="jmsConfig" />
    </jaxws:features>
</jaxws:endpoint>
```

The JMSConfiguration bean needs at least a reference to a connection factory and a target destination.

```
<bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
    p:connectionFactory-ref="jmsConnectionFactory"
    p:targetDestination="test.cxf.jmstransport.queue"
/>
```

If your ConnectionFactory does not cache connections you should wrap it in a spring SingleConnectionFactory. This is necessary because the JMS Transport creates a new connection for each message and the SingleConnectionFactory is needed to cache this connection.

```
<bean id="jmsConnectionFactory"
    class="org.springframework.jms.connection.SingleConnectionFactory">
    <property name="targetConnectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="tcp://localhost:61616"/>
        </bean>
    </property>
</bean>
```

JMSConfiguration options:

Name	Description
connectionFactory	Mandatory field. Reference to a bean that defines a jms ConnectionFactory. Remember to wrap the connectionFactory like described above when not using a pooling ConnectionFactory
wrapInSingleConnectionFactory	Will wrap the connectionFactory with a Spring SingleConnectionFactory, which can improve the performance of the jms transport. Default is true.
reconnectOnException	If wrapping the connectionFactory with a Spring SingleConnectionFactory and reconnectOnException is true, will create a new connection if there is an exception thrown, otherwise will not try to reconnect if the there is an exception thrown. (Default is false.)
targetDestination	JNDI name or provider specific name of a destination. Example for ActiveMQ: test.cxf.jmstransport.queue
replyDestination	
destinationResolver	Reference to a Spring DestinationResolver. This allows to define how destination names are resolved to jms Destinations. By default a DynamicDestinationResolver is used. It resolves destinations using the jms providers features. If you reference a JndiDestinationResolver you can resolve the destination names using JNDI.
transactionManager	Optional reference to a spring transaction manager. This allows to take part in JTA Transactions with your webservice.
taskExecutor	Reference to a spring TaskExecutor. This is used in listeners to decide how to handle incoming messages. Default is a spring SimpleAsyncTaskExecutor.
useJms11	true means JMS 1.1 features are used false means only JMS 1.0.2 features are used. Default is false.

Name	Description
messageIdEnabled	Default is true.
messageTimestampEnabled	Default is true.
cacheLevel	Specify the level of caching that the JMS listener container is allowed to apply. (Default is -1) Please check out the java doc of the org.springframework. jms.listener. DefaultMessageListenerContainer for more information
pubSubNoLocal	If true do not receive your own messages when using topics. Default is false.
receiveTimeout	How many milliseconds to wait for response messages. 0 (default) means wait indefinitely.
explicitQosEnabled	If true, means that QoS parameters are set for each message. (Default is false.)
deliveryMode	NON_PERSISTENT = 1 messages will only be kept in memory (default) PERSISTENT = 2 messages will be persisted to disk
priority	Priority for the messages. Default is 4. See your JMS provider doc for details.
timeToLive	After this time the message will be discarded by the jms provider. Default is 0.
sessionTransacted	If true, means JMS transactions are used. Default is false.
concurrentConsumers	Minimum number of concurrent consumers for listener (default is 1).
maxConcurrentConsumers	Maximum number of concurrent consumers for listener (default 1).
maxConcurrentTasks	Maximum number of threads that handle the received requests. Default 10.
messageSelector	jms selector to filter incoming messages (allows to share a queue)
subscriptionDurable	Default is false.
durableSubscriptionName	
messageType	text (default) binary byte
pubSubDomain	false (default) means use queues true means use topics
jmsProviderTibcoEms	True means that the jms provider is Tibco EMS. Default is false. Currently this activates that the principal in the SecurityContext is populated from the header JMS_TIBCO_SENDER. (available from cxf version 2.2.6)
useMessageIDAsCorrelationID	Specifies whether the JMS broker will use the message ID to correlate messages. By default (false) a CXF client will set a generated correlation id instead

2.6. WS-* Support

2.6.1. WS-Addressing

2.6.1.1. WS-Addressing via XML Configuration / Java API

CXF provides support for the 2004-08 and 1.0 versions of WS-Addressing.

To enable WS-Addressing you may enable the WSAddressingFeature on your service. If you wish to use XML to configure this, you may use the following syntax:

```
<jaxws:endpoint id="{your.service.namespace}YourPortName">
  <jaxws:features>
    <wsa:addressing xmlns:wsa="http://cxf.apache.org/ws/addressing"/>
  </jaxws:features>
</jaxws:endpoint>
```

You can also use the same exact syntax with a <jaxws:client>

```
<jaxws: client id="{your.service.namespace}YourPortName">
  <jaxws:features>
    <wsa:addressing xmlns:wsa="http://cxf.apache.org/ws/addressing"/>
  </jaxws:features>
</jaxws:client>
```

From an API point of view this looks very similar:

```
import org.apache.cxf.jaxws.EndpointImpl;
import org.apache.cxf.ws.addressing.WSAddressingFeature;

MyServiceImpl implementor =  new MyServiceImpl()
EndpointImpl ep = (EndpointImpl) Endpoint.create(implementor);
ep.getFeatures().add(new WSAddressingFeature());
ep.publish("http://some/address");
```

You can also use it with the ClientProxyFactoryBeans and ServerFactoryBeans (and their JAX-WS versions, namely JaxWsProxyFactoryBean and JaxWsServerFactoryBean):

```
import org.apache.cxf.frontend.simple.ClientProxyFactoryBean;
import org.apache.cxf.ws.addressing.WSAddressingFeature;

ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
factory.setServiceClass(MyService.class);
factory.setAddress("http://acme.come/some-service");
factory.getFeatures().add(new WSAddressingFeature());
MyService client = (MyService) factory.create();
```

2.6.1.2. Enabling WS-Addressing with WS-Policy

If you're using Section 2.6.2, “WS-Policy” , CXF can automatically set up WS-Addressing for you if you use the <Addressing> policy expression.

2.6.2. WS-Policy

2.6.2.1. Developing Assertions

There are two steps involved in developing your domain specific assertions, these are:

1. Implementing the Assertion and AssertionBuilder interfaces, and registering the AssertionBuilder with the AssertionBuilderRegistry

2. Providing runtime support for the Assertion, either in form of an interceptor or inside a conduit or a destination, and registering that support if necessary.

Th steps are outlined in some more detail below:

Implementing the Assertion Interface

You can chose to implement the Assertion interface from scratch, or decide to use one of the existing Assertion implementations in the cxf-api module, extending them as required:

PrimitiveAssertion

This class represents an assertion without any attributes or child elements (in particular without a nested Policy element). The AnonymousResponses or NonAnonymousResponses assertions in the addressing metadata namespace http://www.w3.org/2007/01/addressing/metadata are examples of this type of assertion. The implementation of the equal and normalize methods in the class are trivial, and there should be no need to extend this class.

NestedPrimitiveAssertion

This class represents an assertion without any attributes, but with one mandatory nested Policy child element. The Addressing assertions in the addressing metadata namespace is an example of this type of assertion. The implementation of the equal and normalize methods are generic, and there should be no need to extend this class.

JaxbAssertion

This class represents an assertion described by an xml schema type that has been mapped to a Java class. The RM assertion as well as the assertions used in the HTTP module are extensions of this class. Although the equal and normalize methods are not abstract, you probably want to overwrite these methods.

Implementing and Registering the AssertionBuilder Interface

Implementing the build method of the AssertionBuilder interface is straightforward (in the case of JaxbAssertions you can extend the JaxbAssertionBuilder class, which provides an appropriate JAXB context and some other useful methods).

The implementation of buildCompatible may need some more consideration if your assertion represents an element with attributes and/or child elements.

Registration of your AssertionBuilder with the AssertionBuilderRegistry is easy enough: simply add a bean for your AssertionBuilder to the cxf-* file of your module, or to the application's custom cfg file.

Implementing a Policy-Aware Interceptor

This is the easiest way of providing runtime support for an Assertion. Steps 1. and 2. listed in Interaction with the Framework can usually be coded as follows:

```
package mycompany.com.interceptors;
import org.apache.cxf.ws.policy.AssertionInfoMap;

class MyPolicyAwareInterceptor {
   static final QName assertionType = new QName("http://mycompany.com}",
        "MyType"});
   public void handleMessage(Message message) {

      // get AssertionInfoMap
       org.apache.cxf.ws.policy.AssertionInfoMap aim =
           message.get(org.apache.cxf.ws.policy.AssertionInfoMap.class);
       Collection<AssertionInfo ais> = aim.get(assertionType );

       // extract Assertion information
       for (AssertionInfo ai : ais) {
           org.apache.neethi.Assertion a = ai.getAssertion();
           MyAssertionType ma = (MyAssertionType)a;
          // digest ....
       }

       // process message ...
            // express support

       for (AssertionInfo ai : ais) {
           ai.setAsserted(...);
       }          }
}
```

Sometimes, it may be more convenient to spead the above functionality accross several interceptors, possibly according to chain (in, in fault, out, outfault). In any case, you need to also provide a PolicyInterceptorProvider, and declare a corresponding bean. Either implement one from scratch or use the PolicyInterceptorProviderImpl in the api package and customise it as follows (assuming that one and the same interceptor is used for all paths):

```
<bean name="MyPolicyAwareInterceptor"
class="mycompany.com.interceptors.MyPolicyAwareInterceptor"/>
<bean class="org.apache.cxf.ws.policy.PolicyInterceptorProviderImpl">
        <constructor-arg>
            <!-- the list of assertion types supported
                    by this PolicyInterceptorProvider -->
            <list>
                <bean class="javax.xml.namespace.QName">
                    <constructor-arg value="http://mycompany.com}"/>
                    <constructor-arg value="MyType"/>
                </bean>
            </list>
        </constructor-arg>
        <property name="inInterceptors">
            <list>
                <ref bean="MyPolicyAwareInterceptor"/>
            </list>
        </property>
        <property name="inFaultInterceptors">
            <list>
                <ref bean="MyPolicyAwareInterceptor"/>
            </list>
        </property>
        <property name="outInterceptors">
            <list>
```

```
                <ref bean="MyPolicyAwareInterceptor"/>
            </list>
        </property>
        <property name="outFaultInterceptors">
            <list>
                <ref bean="MyPolicyAwareInterceptor"/>
            </list>
        </property>
    </bean>
```

All beans of type PolicyInterceptorProvider are automatically registered with the framework's PolicyInterceptorProviderRegistry.

Implementing a Policy-Aware Conduit/Destination

Initialisation

Conduits/Destinations have access to the EndpointInfo object in their their constructors,. Assuming they also have access to the bus, they can at any time in their lifecycle obtain the effective policy for the endpoint as follows:

```
class MyPolicyAwareConduit {
    static final QName assertionType = new QName("http://mycompany.com}",
        "MyType"});
    ...

    void init() {
        PolicyEngine engine = bus.getExtenation(PolicyEngine.class);
        if (null != engine && engine.isEnabled()) {
        EffectiveEndpointPolicy ep = engine.getEndpointPolicy(endpoint,
            this);
        Collection<Assertion> as = ep.getChosenAlternative();
        for (Assertion a : as) {
            if (assertType.equals(a.getName()) {
                // do something with it ...
            }
       }
       ...
    }
}
```

and similarly for a Destination.

Policy-Aware Message Sending

Given access to the Message object, a conduit can, in its send method, proceed the same way as an interceptor in handleMessage. It can defer the updating of the assertion status in the AssertionInfo objects until called upon by the PolicyVerificationOutInterceptor, i.e. implement the status update in the assertMessage method. If the status update takes place inside of the send method itself, assertMessage, at least for outbound messages, can be implemented as a no-op.

Implementing the Assertor Interface

With canAssert, the conduit/destination simply informs the framework if it understands a given type of assertions. In assertMessage on the other hand, the conduit/destination expresses support (or the lack thereof) for specific assertion instances. See Verification for a description of how this API is used by the verifying policy interceptors in the POST_STREAM or PRE_INVOKE phases. HTTPConduit is an exmaple of a policy aware Conduit. It supports assertions of type HTTPClientPolicy, which are represented in the runtime as JaxbAssertion<HTTPClientPolicy> objects. HTTPConduit also has a data member of type HTTPClientPolicy. It implements assertMessage as follows: for outbound messages, it asserts all JaxbAssertion<HTTPClientPolicy> that are compatible with this data member. For inboun d messages, all HTTPClientPolicy assertions are asserted regardless their attributes. The rationale for this is that the sematics of the HTTPClientPolicy assertion effectively does not mandate any specific action on the inbound message. Similary, on its inbound path, the HTTPDestination asserts all HTTPServerPolicy assertions that are equal to the HTTPServerPolicy assertion configured for the destination, and all assertions of that type on the outbound path.

```
class MyPolicyAwareConduit implements Assertor {
    static final QName MYTYPE = new QName("http://mycompany.com}",
        "MyType"});

    public boolean canAssert(QName name) {
        return MTYPE.equals(name);
    }

    public void assertMessage(Mesage message) {
        AssertionInfoMap = message.get(AssertioninfoMap.class);
        ...
    }
}
```

2.6.2.2. How It Works

Retrieval of Policies

Policies are associated with policy subjects. In the web services context, there are four different subjects:

- Service
- Endpoint
- Operation
- Message

Using WSDL 1.1, the policy-subject association usually takes the form of **xml element attachment** : A wsp:Policy element (the wsp prefix denotes the ??? namespace) is attached to a WSDL element such as wsdl:port. Alternatively, a wsp:PolicyReference elements is attached to a wsdl element. In that case, the actual wsp:Policy element can reside outside of the wsdl. Note that subjects do not correspond to wsdl elements directly. Rather, they map to a set of wsdl elements (see below). For example wsdl:port, wsdl:portType and wsdl:binding elements together describe the endpoint as a subject.

Another form of associating policies with policy subjects is **external attachment** : wsp:PolicyAttachment elements, which can reside in arbitrary locations, explicitly specify the subject(s) they apply to in their AppliesTo child element.

In CXF, elements attached to a wsdl element are available as extensors in the service model representation of that wsdl element. wsp:Policy or wsp:PolicyReference elements can be obtained as extensors of type

UnknownExtensibilityElement in which the element name matches that of the wsp:Policy or wsp:PolicyReference element. Note that these attached elements are not parsed when the service model is built. With xml element attachment in WSDL 1.1, given a Message object, wsp:Policy elements attached to the endpoint or message subject can therefore be obtained by navigating the service model starting with the OperationInfo and/or EndpointInfo object stored in the message (or in the exchange).

The location of documents containing PolicyAttachment documents on the other hand needs to be made known to the framework. This can easily be achieved through configuration, see Specifying the Location of External Attachments .

PolicyAttachments are flexible w.r.t. the type of domain expressions. Domain expressions are used to identify entities such as endpoints, operations or messages with which a policy can be associated:

```
<wsp:PolicyAttachment>
    <wsp:AppliesTo>
        <x:DomainExpression/> +
    </wsp:AppliesTo>
    (<wsp:Policy>...</wsp:Policy> |
           <wsp:PolicyReference>...</wsp:PolicyReference>)
</wsp:PolicyAttachment>
```

Currently, CXF supports only domain expressions of type wsa:EndpointReferenceType: They allow to associate the policies or policy references in an attachment with an endpoint (by means of matching the endpoint's address with that in the EndpointReferenceType element). It is not possible however to associate a Policy with an operation or a message this way. Support for other types of domain expressions can be plugged in by implementing the DomainExpressionBuilder interface and adding a corresponding bean to your configuration file (all DomainExpressionBuilder instances loaded that way will automatically register with the DomainExpressionBuilder and thus be considered in the process of parsing PolicyAttachment elements).

Once that the framework knows where to look for wsp:Policy elements, it can parses these elements and creates runtime presentations for them. This is where AssertionBuilders come into play: All child elements of a wsp:Policy element that are not in the wsp namespace are considered to be assertions. The framework will use its AssertionBuilderRegistry to find an AssertionBuilder registered for the element type in question and, if it finds one, proceed to build an Assertion object from that element (or else throw a PolicyException).

Computation of Effective Policies

As mentioned above, policies are associated with policy subjects. With WSDL 1.1, the effective policy for a subject is the aggregation, or the **merge** , of the policies attached to the wsdl elments representing that subject: The effective policy for a service subject is the merge of all policies applying to the wsdl:service element. The effective policy for an endpoint subject is the merge of all policies applying to the wsdl:port, wsdl:portType and wsdl:binding elements. The effective policy for an operation subject is the merge of all policies applying to the wsdl:portType/wsdl:operation and wsdl:binding/wsdl:operation elements. The effective policy for a (input | output | fault) message subject is the merge of all policies applying to the wsdl:message, (wsdl:portType/wsdl:operation/wsdl:input | wsdl:portType/wsdl:operation/wsdl:output | wsdl:portType/wsdl:operation/wsdl:fault) and (wsdl:binding/wsdl:operation/wsdl:input | wsdl:binding/wsdl:operation/wsdl:output | wsdl:binding/wsdl:operation/wsdl:fault).

Additional aggregation takes place to determine the effective policy of an endpoint: The effective policy for a service is the effective policy for the service subject. The effective policy for an endpoint is the merge of the effective policies for the service subject and the endpoint subject. The effective policy for an operation is the merge of the effective policies for the service subject, the endpoint subject and the operation subject. The effective policy for a (input | output | fault) message is the merge of the effective policies for the service subject, the endpoint subject, the operation subject and the message subject.

Multiple sources can be used to apply policies to the same subject. In the case of an endpoint subject for example, its associated wsdl:port element can have multiple wsp:Policy child elements. Also, a separate document can contain wsp:PolicyAttachment elements in which the AppliesTo children identify the endpoint in question as the

target subject. Both the Policies attached to the port element as well as those in the matching PolicyAttachment elements will then contribute to the effective policy of the endpoint subject.

It is also important to keep in mind that the aggregation process described above makes it possible for an effective policy to have **multiple assertion elements of the same type in one alternative** (although this would not be considered the normal case). Different assertions of the same type within the same alternative do **not** overwrite each other. In fact, if used inappropriately, they may contradict each other. But it is also possible that they complement each other. Either way, the framework does not remove such duplicates and instead leaves it to the interceptors (or other Assertors) involved in the assertion process to decide if they can meaningfully deal with multiple assertions of the same type.

It is obvious that the above aggregation process can be quite resource intense. Effective policies for messages and endpoints are therefore cached by the framework for future reference. The entity that manages the cache of effective policies is the PolicyEngine.

When computing the effective policy for an endpoint or a message, the framework also chooses one of the effective policy's alternatives. Currently, it choses the first alternative in which all assertions **may** be supported, either by interceptors (i.e. there is a PolicyInterceptorProvider for the assertion type) or by the conduit/destination (if this implements the Assertor interface and through its canAssert method confirms that it can support the assertion type). However, even if such an alternative can be found, the chosen alternative is not necessarily supported: An interceptor may in principle be able to support a specific type of assertions, but it may not actually be able to support an individual instance of that assertion type.

The choice of alternative, along with the set of interceptors (obtained from the PolicyInterceptorProviders in the PolicyInterceptorProviderRegistry), is cached along with the actual effective message or endpoint policy in the form of an EffectivePolicy or EffectiveEndpointPolicy object. In the case of an effective endpoint policy, interceptors are chosen in such a way that the assertions in the chosen alternative of the effective endpoint policy can be supported, but also any assertion in any alternative of any of the operation and message specific policies. This is necessary in situations where the underlying message is not known, for example on the server inbound path: Once an alternative has been chosen for the effective policy of the server's endpoint we know which assertions must definitely be supported, regardless the underlying message/operation. Additional interceptors that are necessary to support the assertions that only appear in specific operation or input message policies are added pre-emptively. Note that this generally requires interceptors to be coded defensively - good practice anyway but especially so for interceptors returned by PolicyInterceptorProviders!

On-the-fly Provision of Interceptors

The policy framework, when activated (by loading the PolicyEngine and setting its "enabled" attribute to true), installs a couple of interceptors at bus level which execute early on in their respective interceptor chains:

Role	**Chain**	**Phase**	**Interceptor**	**Effective Subject Policies Known**
Client	Out	SETUP	ClientPolicy-OutInterceptor	Service, Endpoint, Operation, (Input) Message
Client	In	RECEIVE	ClientPolicy-InInterceptor	Service, Endpoint
Client	InFault	RECEIVE	ClientPolicy-InFault-Interceptor	Service, Endpoint
Server	In	RECEIVE	ServerPolicy-InInterceptor	Service, Endpoint
Server	OutFault	SETUP	ServerPolicy-OutFault-Interceptor	Service, Endpoint, Operation, (Fault) Message

Role	Chain	Phase	Interceptor	Effective Subject Policies Known
Server	Out	SETUP	ServerPolicy-OutInterceptor	Service, Endpoint, Operation, (Out) Message

The main purpose of these policy interceptors is to add further interceptors that are required to support the effective policy of the underlying message - even if that policy is not yet known at the time the policy interceptor executes (because the operation is not yet known at that time). If the effective message policy is known, the assertions of its selected alternative are inserted into the message in the form of an AssertionInfoMap. This is a map, keyed by assertion type name, of collections of AssertionInfo objects, the latter being stateful (asserted/not asserted) representations of Assertion objects. When the effective message policy is not known, not only the assertions for the selected alternative in the effective endpoint policy are included in the AssertionInfoMap, but also all assertions in all alternatives of all of the operation and message specific policies. Not all of these will be asserted at the end of the chain, but that is fine if it turns out the unasserted assertions apply to operation sayHi when in fact the chain has been processing the message for a greetMe request!

Policy Aware Interceptors

Policy-aware interceptors extract the collection of AssertionInfo objects for the assertion types they understand from the AssertionInfoMap in the message. They can then use the wrapped Assertion objects to fine tune their behaviour, possibly exhibiting message specific behaviour. They can also express whether or not they could support these assertions. Given an assertion type that has attributes, and assuming there are two instances of assertions of this type, it is possible that the interceptor can assert one, but not the other. In any case, inability to support all assertions understood by the interceptor does not necessarily indicate a failure. As mentioned above in relation to pre-emptive interceptor installation, it is possible that the ones that cannot be supported do not in fact apply to the underlying message at all. Typically the interceptor would strive at supporting as many of these assertions as possible however, and to do so it may avail of the AssertionBuilder's capability to compute a compatible assertion. For example, by scheduling an acknowledgement to be sent in 3 seconds, an RM interceptor would support both of the following RMAssertions:

```
<wsrmp:RMAssertion
    xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
    <wsrmp:AcknowledgementInterval Milliseconds="30000"/>
</wsrmp:RMAssertion>
<wsrmp:RMAssertion
    xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
    <wsrmp:AcknowledgementInterval Milliseconds="50000"/>
</wsrmp:RMAssertion>
```

Verification

Another set of interceptors installed by the policy framework is responsible for verifying that one of the alternatives in the effective policy of the underlying message is indeed supported. These interceptors are:

Chain	Phase	Interceptor
Out, OutFault	POST_STREAM	PolicyVerificationOutInterceptor
In	PRE_INVOKE	PolicyVerificationInInterceptor
InFault	PRE_INVOKE	PolicyVerificationInFaultInterceptor

Their behaviour is symmetric on client and server side. On the outbound chain the effective message policy was known by the time the policy interceptor executing in the SETUP phase had inserted the AssertionInfoMap into

the message. As the map was built exclusively from the Assertion objects that are part of the chosen alternative of the effective message policy, all of them must be supported. In other words, all of the AssertionInfo objects need to be in the asserted state. If one of them is not, the interceptor throws a Fault (wrapping a PolicyException).

On the inbound paths a little bit more work is necessary: If the message is a fault, we know by now what type of fault it is and what operation it applies to. If the message is not a fault message, knowing the underlying operation we can, from the location of the interceptor (client or server side), infer the message subject (input or output message). Either way, all information is now available to obtain the effective message policy. To check if any of is alternatives is supported, the policy verification interceptors then simply check if for each of its assertions the associated AssertionInfo object in the map is in the asserted state. If no alternative is supported, the interceptor throws a Fault (wrapping a PolicyException).

One thing worth noting is that - both on outbound and inbound chains - there may be assertions that only the conduit or destination can support. Although conduit or destination could access Assertion objects and tailor their behaviour when sending or receiving the current message, it is not knoan at this point whether this "tailoring" actually succeeded for the underlying message, i.e. whether the assertions in questions could actually be supported. For this reason, the policy verification interceptors check if the conduit or destination implements the Assertor interface. It it does, they pass it the Message object so they confirn their support (or the lack thereof) for these assertions. The above described traveral of the AssertionInfo map only takes place after the conduit or destination had a chance to m ake their contribution.

2.6.2.3. WS-Policy Framework Overview

The WS-Policy framework provides infrastructure and APIs that allow CXF users and developers to use WS-Policy.

It is compliant with the November 2006 draft publications of the Web Services Policy 1.5 - Framework and Web Services Policy 1.5 - Attachment specifications.

The framework consists of a core runtime and APIs that allow developers to plug in support for their own domain assertions:

Core

The core is responsible for:

- retrieval of policies from different sources (wsdl documents, external documents)
- computation of effective policies for service, endpoint, operation and message objects
- on-the-fly provision of interceptors based on the effective policies for a particular message
- verification that one of the effective policy's alternatives is indeed supported.

Policy operations such as merge and normalisation (but not intersection) are based on Apache Neethi .

APIs

AssertionBuilder

The AssertionBuilder API is a concept from Neethi, slightly modified to avoid the dependency on the Axis object model, and extended to include support for domain specific behaviour of intersection and comparison.

```
public interface AssertionBuilder {
   // build an Assertion object from a given DOM element
   Assertion build(Element element);
  // return the schema type names of assertions understood by this builder
  Collection<QName> getSupportedTypes();
  // return an Assertion object that is
  // compatible with the specified assertions
  Assertion buildCompatible(Assertion a, Assertion b);
}
```

AssertionBuilder implementations are loaded dynamically and are automatically registered with the AssertionBuilderRegistry, which is available as a Bus extension. Currently, CXF supports AssertionBuilder and Assertion implementations for the following assertion types:

```
{http://schemas.xmlsoap.org/ws/2005/02/rm/policy}RMAssertion
{http://www.w3.org/2007/01/addressing/metadata}Addressing
{http://www.w3.org/2007/01/addressing/metadata}AnonymousResponses
{http://www.w3.org/2007/01/addressing/metadata}NonAnonymousResponses
{http://cxf.apache.org/transports/http/configuration}client
{http://cxf.apache.org/transports/http/configuration}server
```

along with the Section 2.6.6, “WS-SecurityPolicy” defined assertions.

They are all based on generic Assertion implementations (PrimitiveAssertion, NestedPrimitiveAssertion, JaxbAssertion) that developers can parameterize or extend when developing their own assertions, see Section 2.6.2.1, “Developing Assertions” .

PolicyInterceptorProvider

This API is used to automatically engage interceptors required to support domain specific assertions at runtime, thus simplifying interceptor configuration a lot.

```
public interface PolicyInterceptorProvider extends InterceptorProvider {
  // return the schema types of the asssertions that can be supported
  Collection<QName> getAssertionTypes()
}
```

Currently, CXF supports PolicyInterceptorProvider implementations for the following assertion types:

```
{http://schemas.xmlsoap.org/ws/2005/02/rm/policy}RMAssertion
{http://www.w3.org/2007/01/addressing/metadata}Addressing
{http://www.w3.org/2007/01/addressing/metadata}AnonymousResponses
{http://www.w3.org/2007/01/addressing/metadata}NonAnonymousResponses
```

along with the Section 2.6.6, “WS-SecurityPolicy” defined assertions.

In addition, the framework offers an API to refine domain expression(s) (xml elements describing policy subjects within a policy scope) in policy attachments. There is currently only one implementation for EndpointReferenceType domain expressions (matching over the address). Another implementation, using XPath expressions, is in work.

Interaction with the Framework

Components interact with the policy framework mainly in order to:

1. retrieve the assertions pertaining to the underlying message (at least the ones known to the component) so the component can operate on the message accordingly

2. confirm that the assertions pertaining to the underlying message are indeed supported.

Like most other CXF features, the policy framework is itself largely interceptor based. Thus, most interaction with the framework is indirect through the Message object: Policy interceptors make AssertionInfo objects (stateful representations of assertions) available to subsequently executing, policy-aware interceptors by inserting them into the Message object. Extracting the AssertionInfo objects from the Message allows interceptors to perform steps 1. and 2. above:

```
import org.apache.neethi.Assertion;

public class AssertionInfo {
  ...
  public boolean isAsserted() {...}
  public void setAsserted(boolean asserted) {...}
  public Assertion getAssertion() {...}
}
```

The WS-Addressing and WS-RM interceptors are examples for this style of intercation.

Somtimes, Conduits and destinations also want to assert their capabilities. But they cannot normally wait for Assertion information being made available to them via the Message object: Conduits may exhibit message specific behaviour (for example, apply message specific receive timeouts), but decisions made during the initialisation phase may limit their capability to do so. And Destinations cannot normally exhibit message or operation specific behaviour at all. But both may still be able to support assertions in the effective endpoint's policy.

Their interaction with the policy framework therefore typically involves the PolicyEngine through which they obtain the effective policy for the underlying endpoint (for step 1.):

```
public interface PolicyEngine {
    ...
    EndpointPolicy getClientEndpointPolicy(EndpointInfo ei,
        Conduit conduit);
    EndpointPolicy getServerEndpointPolicy(EndpointInfo ei,
        Destination destination);
}

public interface EndpointPolicy {
    ...
    Policy getPolicy();
    Collection<Assertion> getChosenAlternative();
}
```

To perform step 2. they implement the Assertor interface (namely its assertMessage method):

```
public class Assertor {
  ...
  public boolean canAssert(QName name);
  public void assertMessage(Message message);
}
```

An example for policy aware conduits and destinations in CXF are the HTTP conduit and destination. They do support assertions of element type HTTPClientPolicy and HTTPServerPolicy respectively.

2.6.3. WS-ReliableMessaging

CXF supports the February 2005 version of the Web Services Reliable Messaging Protocol (WS-ReliableMessaging) specification. Like most other features in CXF, it is interceptor based. The WS-Reliable Messaging implementation consists of 4 interceptors in total:

Interceptor	Task
org.apache.cxf.ws.rm.RMOutInterceptor	Responsible for sending CreateSequence requests and waiting for their CreateSequenceResponse responses, and and aggregating the sequence properties (id and message number) for an application message.
org.apache.cxf.ws.rm.RMInInterceptor	Intercepting and processing RM protocol messages (these will not the application level), as well as SequenceAcknowledgments piggybacked on application messages.
org.apache.cxf.ws.rm.soap.RMSoapInterceptor	Encoding and decoding the RM headers
org.apache.cxf.ws.rm.soap.RetransmissionInterceptor	Responsible for creating copies of application messages for future resends.

2.6.3.1. Interceptor Based QOS

The presence of the RM interceptors on the respective interceptor chains alone will take care that RM protocol messages are exchanged when necessary. For example, upon intercepting the first application message on the outbound interceptor chain, the RMOutInterceptor will send a CreateSequence request and only proceed with processing the original application message after it has the CreateSequenceResponse response. Furthermore, the RM interceptors are responsible for adding the Sequence headers to the application messages and, on the destination side, extracting them from the message.

This means that no changes to application code are required to make the message exchange reliable!

You can still control sequence demarcation and other aspects of the reliable exchange through configuration however. For example, while CXF by default attempts to maximize the lifetime of a sequence, thus reducing the overhead incurred by the RM protocol messages, you can enforce the use of a separate sequence per application message by configuring the RM source's sequence termination policy (setting the maximum sequence length to 1). See the Reliable Messaging Configuration Guide for more details on configuring this and other aspects of the reliable exchange.

2.6.4. WS-SecureConversation

WS-SecureConversation support in CXF builds upon the Section 2.6.6, “WS-SecurityPolicy” implementation to handle the SecureConverstationToken policy assertions that could be found in the WS-SecurityPolicy fragment.

Note: Because the WS-SecureConversation support builds on the WS-SecurityPolicy support, this is currently only available to "wsdl first" projects.

One of the "problems" of WS-Security is that the use of strong encryption keys for all communication extracts a hefty performance penalty on the communication. WS-SecureConversation helps to aleviate that somewhat by allowing the client and service to use the strong encryption at the start to negotiatate a set of new security keys that will be used for furthur communication. This can be a huge benefit if the client needs to send many requests to the service. However, if the client only needs to send a single request and then is discarded, WS-SecureConversation is actually slower as the key negotiation requires and extra request/response to the server.

With WS-SecureConversation, there are two Security policies that come into affect:

1. The "outer" policy that describes the security requirements for interacting with the actual endpoint. This will contain a SecureConversationToken in it someplace.

2. The "bootstrap" policy that is contained in the SecureConverstationToken. This policy is the policy in affect when the client is negotiating the SecureConversation keys.

Configuring the WS-SecurityPolicy properties for WS-SecureConversation works exactly like the configuration for straight WS-SecurityPolicy. The only difference is that there needs to be a way to specify which properties are intended for the bootstrap policy in the SecureConversationToken and which are intended for the actual service policy. To accomplish this, properties intended for the SecureConversationToken bootstrap policy are appended with ".sct". For example:

```
<jaxws:client name="{http://InteropBaseAddress/interop} ...
          XDC-SEES_IPingService" createdFromAPI="true">
    <jaxws:properties>
        <!-- properties for the external policy -->
        <entry key="ws-security.username" value="abcd"/>

        <!-- properties for SecureConversationToken bootstrap policy -->
        <entry key="ws-security.username.sct" value="efgh"/>
        <entry key="ws-security.callback-handler.sct"
            value="interop.client.KeystorePasswordCallback"/>
        <entry key="ws-security.encryption.properties.sct"
            value="etc/bob.properties"/>
    </jaxws:properties>
</jaxws:client>
```

Via the Java API, use code similar to the following:

```
org.apache.cxf.endpoint.Client client;
client.getRequestContext().put("ws-security.username.sct", username);
client.getRequestContext().put("ws-security.password.sct", password);
```

Via the Java API, use code similar to the following:

```
org.apache.cxf.endpoint.Client client;
client.getRequestContext().put("ws-security.username.sct", username);
client.getRequestContext().put("ws-security.password.sct", password);
```

Note: In most common cases of WS-SecureConversation, you won't need any configuration for the service policy. All of the "hard" stuff is used for the bootstrap policy and the service provides new keys for use by the service policy. This keeps the communication with the service itself as simple and efficient as possible.

2.6.5. WS-Security

2.6.5.1. WS-Security

WS-Security provides means to secure your services above and beyond transport level protocols such as HTTPS. Through a number of standards such as XML-Encryption, and headers defined in the WS-Security standard, it allows you to:

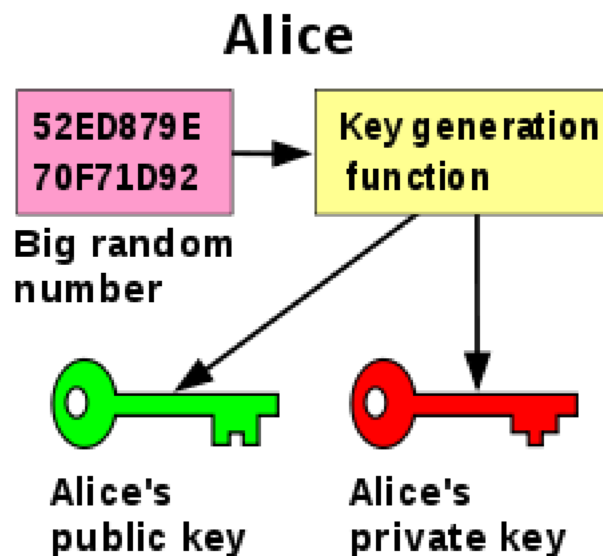
- Pass authentication tokens between services
- Encrypt messages or parts of messages
- Sign messages
- Timestamp messages

Currently, CXF implements WS-Security by integrating WSS4J . To use the integration, you'll need to configure these interceptors and add them to your service and/or client.

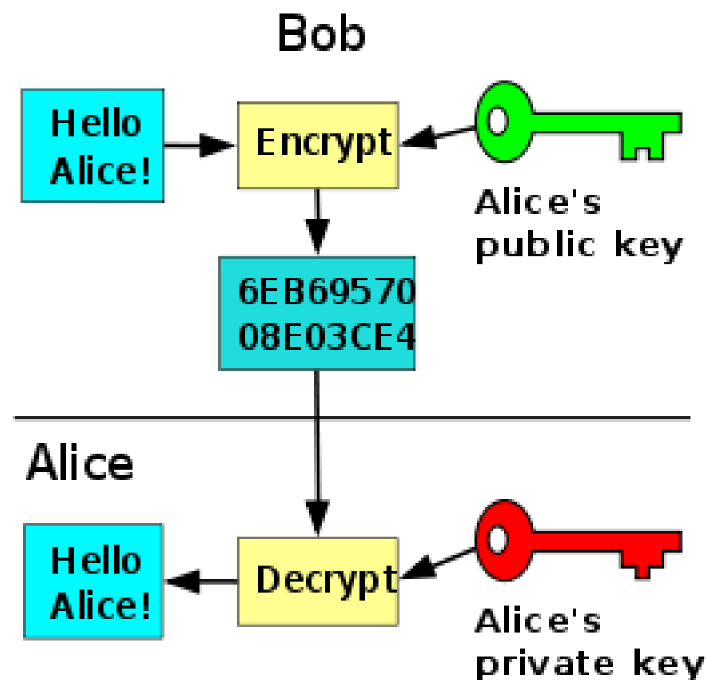
2.6.5.2. Overview of encryption and signing

WS-Security makes heavy use of public/private key cryptography. To really understand how to configure WS-Security, it is helpful - if not necessary - to understand these basics. The Wikipedia has an excellent entry on this, but we'll try to summarize the relevant basics here (This content is a modified version of the wikipedia content..)

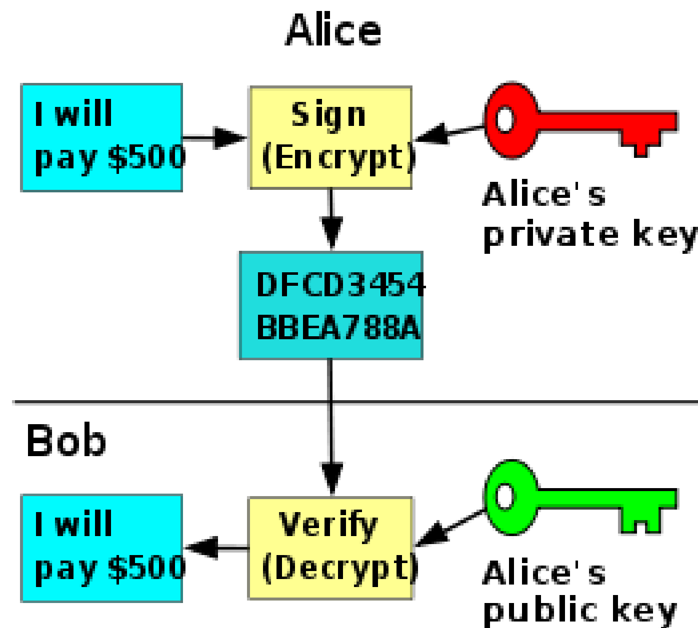
With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function.



The keys are related mathematically, but cannot be derived from one another. With these keys we can encrypt messages. For example, if Bob wants to send a message to Alice, he can encrypt a message using her public key. Alice can then decrypt this message using her private key. Only Alice can decrypt this message as she is the only one with the private key.



Messages can also be signed. This allows you to ensure the authenticity of the message. If Alice wants to send a message to Bob, and Bob wants to be sure that it is from Alice, Alice can sign the message using her private key. Bob can then verify that the message is from Alice by using her public key.



2.6.5.3. Configuring the WSS4J Interceptors

To enable WS-Security within CXF for a server or a client, you'll need to set up the WSS4J interceptors. You can either do this via the API for standalone web services or via Spring XML configuration for servlet-hosted ones. This section will provide an overview of how to do this, and the following sections will go into more detail about configuring the interceptors for specific security actions.

It is important to note that:

1. If you are using CXF 2.0.x, you must add the SAAJ(In/Out)Interceptors if you're using WS-Security (This is done automatically for you from CXF 2.1 onwards). These enable creation of a DOM tree for each request/ response. The support libraries for WS-Security require DOM trees.

2. The web service provider may not need both in and out WS-Security interceptors. For instance, if you are just requiring signatures on incoming messages, the web service provider will just need an incoming WSS4J interceptor and only the SOAP client will need an outgoing one.

Adding the interceptors via the API

On the Server side, you'll want to add the interceptors to your CXF Endpoint. If you're publishing your service using the JAX-WS APIs, you can get your CXF endpoint like this:

```
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.jaxws.EndpointImpl;

EndpointImpl jaxWsEndpoint =
    (EndpointImpl) Endpoint.publish("http://host/service", myServiceImpl);
Endpoint cxfEndpoint = jaxWsEndpoint.getServer().getEndpoint();
```

If you've used the (JaxWs)ServerFactoryBean, you can simply access it via the Server object:

```
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;

ServerFactoryBean factory = ...;
...
Server server = factory.create();
Endpoint cxfEndpoint = server.getEndpoint();
```

On the client side, you can obtain a reference to the CXF endpoint using the ClientProxy helper:

```
GreeterService gs = new GreeterService();
Greeter greeter = gs.getGreeterPort();
...
org.apache.cxf.endpoint.Client client =
    org.apache.cxf.frontend.ClientProxy.getClient(greeter);
org.apache.cxf.endpoint.Endpoint cxfEndpoint = client.getEndpoint();
```

Now you're ready to add the interceptors:

```
import org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor;
import org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor;
...

Map<String,Object> inProps= new HashMap<String,Object>();
... // how to configure the properties is outlined below;

WSS4JInInterceptor wssIn = new WSS4JInInterceptor(inProps);
cxfEndpoint.getInInterceptors().add(wssIn);

Map<String,Object> outProps = new HashMap<String,Object>();
... // how to configure the properties is outlined below;

WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor(outProps);
cxfEndpoint.getOutInterceptors().add(wssOut);
```

2.6.5.4. Spring XML Configuration

If you're using Spring to build endpoints (e.g., web services running on a servlet container such as Tomcat), you can easily accomplish the above using your bean definitions instead.

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath*:META-INF/cxf/cxf-extension-*.xml" />

<jaxws:endpoint id="myService"
   implementor="com.acme.MyServiceImpl"
   address="http://localhost:9001/MyService">

   <bean id="myPasswordCallback"
      class="com.mycompany.webservice.ServerPasswordCallback"/>

   <jaxws:inInterceptors>
      <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
         <constructor-arg>
```

```
            <map>
               <entry key="action" value="UsernameToken"/>
               <entry key="passwordType" value="PasswordDigest"/>
               <entry key="signaturePropFile" value="..."/>
               <entry key="passwordCallbackRef">
                  <ref bean="myPasswordCallback"/>
               </entry>
               ...
            </map>
         </constructor-arg>
      </bean>
   </jaxws:inInterceptors>
</jaxws:endpoint>
```

The entry keys and values given in the constructor-arg element above (action, signaturePropFile, etc.) map to the text strings in WSS4J's WSHandlerConstants and WSConstants classes for the corresponding WSHandlerConstants.XXXXX and WSConstants.XXXX constants you see in the section below. So by viewing WSHandlerConstants, for example, you can see that the WSHandlerConstants.USERNAME_TOKEN value given below would need to be "UsernameToken" instead when doing Spring configuration.

If you want to avoid looking up the text keys for the WSHandlerConstants.XXXXX and WSConstants.XXXX constants, you can also use the Spring util namespace to reference static constants in your Spring context as shown below.

```
<beans
  ...
  xmlns:util="http://www.springframework.org/schema/util"
  ...
  xsi:schemaLocation="
        ...
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">

  ...

  <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
      <map>
        <entry value="UsernameToken">
          <key>
            <util:constant static-field=
            "org.apache.ws.security.handler.WSHandlerConstants.ACTION"/>
          </key>
        </entry>
        ...
      </map>
    </constructor-arg>
  </bean>

  ...
```

Additional Configuration Options

While the CXF WSS4J interceptors support the standard configuration properties available in WSHandlerConstants.XXXXX and WSConstants.XXXX, CXF also provides access to some additional low level configuration capabilities in WSS4J and some other security related interceptors.

Validating Signature and/or Encryption of Message Contents

As of CXF 2.2.8, the CryptoCoverageChecker interceptor allows one to validate signature and encryption coverage of message contents without migrating to a WS-SecurityPolicy based configuration. The interceptor can support enforcement of signature and encryption coverage at both the element and content level (be aware that the combination of signature and content do not represent a valid combination of coverage type and coverage scope). To configure this interceptor using the API, follow the example below.

```
import org.apache.cxf.ws.security.wss4j.CryptoCoverageChecker;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageChecker. ...
   XPathExpression;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageUtil.CoverageScope;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageUtil.CoverageType;

Map<String, String> prefixes = new HashMap<String, String>();
        prefixes.put("ser", "http://www.sdj.pl");
        prefixes.put("soap", "http://schemas.xmlsoap.org/soap/envelope/");

List<XPathExpression> xpaths = Arrays.asList(
    new XPathExpression("//ser:Header", CoverageType.SIGNED,
        CoverageScope.ELEMENT),
    new XPathExpression("//soap:Body", CoverageType.ENCRYPTED,
        CoverageScope.CONTENT));

CryptoCoverageChecker checker = new CryptoCoverageChecker(prefixes,
    xpaths);
```

The interceptor can also be configured in Spring using the conventional bean definition format.

After configuring the interceptor as above, simply add the interceptor to your client or server interceptor chain as shown previsouly with the WSS4J interceptors. Ensure that you include the WSS4JInInterceptor in the chain or all requests will be denied if you enforce any coverage XPaths.

Custom Processors

As of CXF 2.0.10 and 2.1.4, you can specify custom WSS4J Processor configurations on the WSS4JInInterceptor. To activate this configuration option, one provides a non-WSS4J defined property, wss4j.processor.map, to the WSS4JInInterceptor as shown in the following Spring example. The same configuratoin can be acheieved through the API as well. The key value is an XML qualified name of the WS-S header element to process with the given processor implementation. The entry values can be a String representing a class name of the processor to instantiate, an Object implementing Processor, or null to disable processing of the given WS-S header element.

```
<bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
  <constructor-arg>
    <map>
      ...
      <!-- This reconfigures the processor implementation that WSS4j
              uses to process a WS-S Signature element. -->
      <entry key="wss4j.processor.map">
        <map key-type="javax.xml.namespace.QName">
          <entry value="my.class">
            <key>
              <bean class="javax.xml.namespace.QName">
                <constructor-arg
                   value="http://www.w3.org/2000/09/xmldsig#"/>
                <constructor-arg value="Signature"/>
              </bean>
```

```
            </key>
          </entry>
        </map>
      </entry>
      ...
    </map>
  </constructor-arg>
</bean>
```

Custom Actions

As of CXF 2.2.6, you can specify custom WSS4J Action configurations on the WSS4JOutInterceptor. To activate this configuration option, one provides a non-WSS4J defined property, wss4j.action.map, to the WSS4JOutInterceptor as shown in the following Spring example. The same configuratoin can be acheieved through the API as well. The key value is an integer representing the WSS4J action identifier. The entry values can be a String representing a class name of the action to instantiate or an Object implementing Action. This configuration option allows you to override built-in action implementations or add your own.

```
<bean class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
  <constructor-arg>
    <map>
      ...
      <!-- Redefines the action for SAMLTokenSigned to use
          a custom implementation.  -->
      <entry key="wss4j.action.map">
        <map key-type="java.lang.Integer" value-type="java.lang.Object">
          <entry key="0x10" value-ref="mySamlTokenSignedAction"/>
        </map>
      </entry>      ...
    </map>
  </constructor-arg>
</bean>
```

2.6.5.5. Configuring WS-Security Actions

Username Token Authentication

WS-Security supports many ways of specifying tokens. One of these is the UsernameToken header. It is a standard way to communicate a username and password or password digest to another endpoint. Be sure to review the OASIS UsernameToken Profile Specification for important security considerations when using UsernameTokens. Note that the nonce support recommended by the specification for guarding against replay attacks has not yet been implemented either in CXF or WSS4J.

For the server side, you'll want to set up the following properties on your WSS4JInInterceptor (see above [126] for code sample):

```
inProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.USERNAME_TOKEN);
// Password type : plain text
inProps.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_TEXT);
// for hashed password use:
//properties.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_DIGEST);
// Callback used to retrieve password for given user.
```

```
inProps.put(WSHandlerConstants.PW_CALLBACK_CLASS,
   ServerPasswordHandler.class.getName());
```

The password callback class allows you to retrieve the password for a given user so that WS-Security can determine if they're authorized. Here is a small example:

```
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        if (pc.getIdentifier().equals("joe")) {
            // set the password on the callback.
            // This will be compared to the
            // password which was sent from the client.
            pc.setPassword("password");
        }
    }

}
```

Note that for up to and including CXF 2.3.x, the password validation of the special case of a plain-text password (or any other yet unknown password type) is delegated to the callback class, see org.apache.ws.security.processor.UsernameTokenProcessor#handleUsernameToken() method javadoc of the WSS4J project. In that case, the ServerPasswordCallback should be something like the following one:

```
public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        if (pc.getIdentifier().equals("joe") {
           if (!pc.getPassword().equals("password")) {
                throw new IOException("wrong password");
           }
        }
    }

}
```

For CXF 2.4 onwards, the callback handler supplies the password for all cases, and the validation is done internally (but can be configured). See here for more information. On the Client side you'll want to configure the WSS4J outgoing properties:

```
outProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.USERNAME_TOKEN);
// Specify our username
outProps.put(WSHandlerConstants.USER, "joe");
// Password type : plain text
outProps.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_TEXT);
```

```
// for hashed password use:
//properties.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_DIGEST);
// Callback used to retrieve password for given user.
outProps.put(WSHandlerConstants.PW_CALLBACK_CLASS,
    ClientPasswordHandler.class.getName());
```

Once again we're using a password callback, except this time instead of specifying our password on the server side, we're specifying the password we want sent with the message. This is so we don't have to store our password in our configuration file.

```
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        // set the password for our message.
        pc.setPassword("password");
    }

}
```

In the case of multiple users with different passwords, use the WSPasswordCallback 's getIdentifier() method to obtain the username of the current SOAP request.

Here is an example of WS-Security implemented using annotations for interceptors (uses UsernameToken).

Using X.509 Certificates

The X.509 Certificate Token Profile (pdf) provides another option for implementing WS-Security. For the Signature and Encryption actions, you'll need to create a public & private key for the entities involved. You can generate a self-signed key pair for your development environment via the following steps. Keep in mind these will not be signed by an external authority like Verisign, so are inappropriate for production use.

1. Creating private key with given alias and password like "myAlias"/"myAliasPassword" in keystore (protected by password for security reasons)

```
keytool -genkey -alias myAlias -keypass myAliasPassword -keystore \
  privatestore.jks -storepass keyStorePassword -dname "cn=myAlias"
  -keyalg RSA
```

The alias is simply a way to identify the key pair. In this instance we are using the RSA algorithm.

2. Self-sign our certificate (in production environment this will be done by a company like Verisign).

```
keytool -selfcert -alias myAlias -keystore privatestore.jks -storepass
  keyStorePassword -keypass myAliasPassword
```

3. Export the public key from our private keystore to file named key.rsa

```
keytool -export -alias myAlias -file key.rsa -keystore privatestore.jks
```

```
  -storepass keyStorePassword
```

4. Import the public key to new keystore:

```
keytool -import -alias myAlias  -file key.rsa -keystore publicstore.jks
  -storepass keyStorePassword
```

So now we have two keystores containing our keys - a public one (publicstore.jks) and a private one (privatestore.jks). Both of them have keystore password set to keyStorePass (this not recommended for production but ok for development) and alias set to myAlias. The file key.rsa can removed from filesystem, since it used only temporarily. Storing keys in keystores is strongly advised because a keystore is protected by a password.

A more detailed description of key generation can be found here: http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html

How to create a production certificate can be found here: http://support.globalsign.net/en/objectsign/java.cfm

Signing

Signing a message is used to validate to the recipient that the message could only have come from a certain sender, and that the message was not altered in transit. It involves the sender encrypting a digest (hash) of the message with its private key, and the recipient unencrypting the hash with the sender's public key, and recalculating the digest of the message to make sure the message was not altered in transit (i.e., that the digest values calculated by both the sender and recipient are the same). For this process to occur you must ensure that the Client's public key has been imported into the server's keystore using keytool.

On the client side, our outgoing WS-Security properties will look like so (see above [126] for code sample):

```
outProps.put(WSHandlerConstants.ACTION, "Signature");
outProps.put(WSHandlerConstants.USER, "myAlias");
outProps.put(WSHandlerConstants.PW_CALLBACK_CLASS,
   ClientCallbackHandler.class.getName());
outProps.put(WSHandlerConstants.SIG_PROP_FILE, "client_sign.properties");
```

The USER that is specified is the key alias for the client. The password callback class is responsible for providing that key's password.

Tip

For X.509 support you will normally have multiple actions, e.g. Encryption with Signature. For these cases, just space-separate the actions in the ACTION property as follows:

```
outProps.put(WSHandlerConstants.ACTION,
WSHandlerConstants.TIMESTAMP + " " +
WSHandlerConstants.SIGNATURE + " " +
WSHandlerConstants.ENCRYPT);
```

Alternatively, you may space-separate the string literals you see above in the Spring configuration (e.g., "Signature Encrypt")

Our client_sign.properties file contains several settings to configure WSS4J:

```
org.apache.ws.security.crypto.provider= \
org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=keyStorePassword
```

```
org.apache.ws.security.crypto.merlin.keystore.alias=myAlias
org.apache.ws.security.crypto.merlin.file=client_keystore.jks
```

On the server side, we need to configure our incoming WSS4J interceptor to verify the signature using the Client's public key.

```
inProps.put(WSHandlerConstants.ACTION, "Signature");
inProps.put(WSHandlerConstants.SIG_PROP_FILE, "server.properties");
```

Our server_sign.properties file contains several settings to configure WSS4J:

```
org.apache.ws.security.crypto.provider= \
org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=amex123
org.apache.ws.security.crypto.merlin.file=server_keystore.jks
```

Encryption

Encryption involves the sender encrypting the message with the recipient's public key to ensure that only the recipient can read the message (only the recipient has its own private key, necessary for decrypting the message.) This requires the sender to have the recipient's public key in its keystore.

The process for encrypting is very similar to and indeed usually combined with the signature process above. Our WS-Security test sample provides an example of encrypting requests and responses.

2.6.6. WS-SecurityPolicy

CXF 2.2 introduced support for using WS-SecurityPolicy to configure WSS4J instead of the custom configuration documented on the Section 2.6.5, “WS-Security” page. However, all of the "background" material on the Section 2.6.5, “WS-Security” page still applies and is important to know. WS-SecurityPolicy just provides an easier and more standards based way to configure and control the security requirements. With the security requirements documented in the WSDL as Section 2.6.2, “WS-Policy” fragments, other tools such as .NET can easily know how to configure themselves to inter-operate with CXF services.

2.6.6.1. Enabling WS-SecurityPolicy

In CXF 2.2, if the cxf-rt-ws-policy and cxf-rt-ws-security modules are available on the classpath, the WS-SecurityPolicy stuff is automatically enabled. Since the entire security runtime is policy driven, the only requirement is that the policy engine and security policies be available.

If you are using the full "bundle" jar, all the security and policy stuff is already included.

2.6.6.2. Policy description

With WS-SecurityPolicy, the binding and/or operation in the wsdl references a Section 2.6.2, “WS-Policy” fragment that describes the basic security requirements for interacting with that service. The WS-SecurityPolicy specification allows for specifying things like asymmetric/symmetric keys, using transports (https) for encryption, which parts/headers to encrypt or sign, whether to sign then encrypt or encrypt then sign, whether to include

timestamps, whether to use derived keys, etc... Basically, it describes what actions are necessary to securely interact with the service described in the WSDL.

However, the WS-SecurityPolicy fragment does not include "everything" that is required for a runtime to be able to able to create the messages. It does not describe things such as locations of key stores, user names and passwords, etc... Those need to be configured in at runtime to augment the WS-SecurityPolicy fragment.

2.6.6.3. Configuring the extra properties

With CXF 2.2, there are several extra properties that may need to be set to provide the additional bits of information to the runtime:

ws-security.username	The username used for UsernameToken policy assertions
ws-security.password	The password used for UsernameToken policy assertions. If not specified, the callback handler will be called.
ws-security.callback-handler	The WSS4J security CallbackHandler that will be used to retrieve passwords for keystores and UsernameTokens.
ws-security.signature.properties	The properties file/object that contains the WSS4J properties for configuring the signature keystore and crypto objects
ws-security.encryption.properties	The properties file/object that contains the WSS4J properties for configuring the encryption keystore and crypto objects
ws-security.signature.username	The username or alias for the key in the signature keystore that will be used. If not specified, it uses the the default alias set in the properties file. If that's also not set, and the keystore only contains a single key, that key will be used.
ws-security.encryption.username	The username or alias for the key in the encryption keystore that will be used. If not specified, it uses the the default alias set in the properties file. If that's also not set, and the keystore only contains a single key, that key will be used. For the web service provider, the useReqSigCert keyword can be used to accept (encrypt to) any client whose public key is in the service's truststore (defined in ws-security.encryption.properties.)
ws-security.signature.crypto	Instead of specifying the signature properties, this can point to the full WSS4J Crypto object. This can allow easier "programmatic" configuration of the Crypto information."
ws-security.encryption.crypto	Instead of specifying the encryption properties, this can point to the full WSS4J Crypto object. This can allow easier "programmatic" configuration of the Crypto information."

Note: for Symmetric bindings that specify a protection token, the ws-security-encryption properties are used.

Configuring via Spring

The properties are easily configured as client or endpoint properties--use the former for the SOAP client, the latter for the web service provider.

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

```
   http://cxf.apache.org/schemas/jaxws.xsd">

   <jaxws:client name="{http://cxf.apache.org}MyPortName"
      createdFromAPI="true">
      <jaxws:properties>
         <entry key="ws-security.callback-handler"
            value="interop.client.KeystorePasswordCallback"/>
         <entry key="ws-security.signature.properties"
            value="etc/client.properties"/>
         <entry key="ws-security.encryption.properties"
            value="etc/service.properties"/>
         <entry key="ws-security.encryption.username"
            value="servicekeyalias"/>
      </jaxws:properties>
   </jaxws:client>

</beans>
```

For the jaxws:client's *name* attribute above, use the namespace of the WSDL along with the *name* attribute of the desired wsdl:port element under the WSDL's service section. (See here and here for an example.)

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

   <jaxws:endpoint
      id="MyService"
      address="https://localhost:9001/MyService"
      serviceName="interop:MyService"
      endpointName="interop:MyServiceEndpoint"
      implementor="com.foo.MyService">

      <jaxws:properties>
         <entry key="ws-security.callback-handler"
            value="interop.client.UTPasswordCallback"/>
         <entry key="ws-security.signature.properties"
            value="etc/keystore.properties"/>
         <entry key="ws-security.encryption.properties"
            value="etc/truststore.properties"/>
         <entry key="ws-security.encryption.username"
            value="useReqSigCert"/>
      </jaxws:properties>

   </jaxws:endpoint>
</beans>
```

Configuring via API's

Configuring the properties for the client just involves setting the properties in the client's RequestContext:

```
Map<String, Object> ctx = ((BindingProvider)port).getRequestContext();
ctx.put("ws-security.encryption.properties", properties);
port.echoString("hello");
```

2.6.7. WS-Trust

WS-Trust support in CXF builds upon the Section 2.6.6, “WS-SecurityPolicy” implementation to handle the IssuedToken policy assertions that could be found in the WS-SecurityPolicy fragment.

Note: Because the WS-IssuedToken support builds on the WS-SecurityPolicy support, this is currently only available to "wsdl first" projects.

WS-Trust extends the WS-Security specification to allow issuing, renewing, and validation of security tokens. A lot of what WS-Trust does centers around the use of a "Security Token Service", or STS. The STS is contacted to obtain security tokens that are used to create messages to talk to the services. The primary use of the STS is to acquire SAML tokens used to talk to the service. Why is this interesting?

When using "straight" WS-Security, the client and server need to have keys exchanged in advance. If the client and server are both in the same security domain, that isn't usually a problem, but for larger, complex applications spanning multiple domains, that can be a burden. Also, if multiple services require the same security credentials, updating all the services when those credentials change can by a major operation.

WS-Trust solves this by using security tokens that are obtained from a trusted Security Token Service. A client authenticates itself with the STS based on policies and requirements defined by the STS. The STS then provides a security token (example: a SAML token) that the client then uses to talk to the target service. The service can validate that token to make sure it really came from the trusted STS.

When the WS-SecurityPolicy runtime in CXF encounters an IssuedToken assertion in the policy, the runtime requries an instance of org.apache.cxf.ws.security.trust.STSClient to talk to the STS to obtain the required token. Since the STSClient is a WS-SecurityPolicy client, it will need configuration items to be able to create it's secure SOAP messages to talk to the STS.

2.6.7.1. General Configuration

There are several ways to configure the STSClient:

Direct configuration of an STS bean in the properties: In this scenario, a STSClient object is created directly as a property of the client object. The wsdlLocation, service/endpoint names, etc., are all configured in line for that client.

```
<jaxws:client name="{http://cxf.apache.org/}MyService"
      createdFromAPI="true">
   <jaxws:properties>
      <entry key="ws-security.sts.client">
         <!-- direct STSClient config and creation -->
         <bean class="org.apache.cxf.ws.security.trust.STSClient">
            <constructor-arg ref="cxf"/>
            <property name="wsdlLocation" value="target/wsdl/trust.wsdl"/>
            <property name="serviceName" value=
         "{http://cxf.apache.org/securitytokenservice} ...
             SecurityTokenService"/>
            <property name="endpointName" value=
         "{http://cxf.apache.org/securitytokenservice} ...
             SecurityTokenEndpoint"/>
            <property name="properties">
               <map>
                 <entry key="ws-security.username" value="alice"/>
                 <entry key="ws-security.callback-handler"
                    value="client.MyCallbackHandler"/>
                 <entry key="ws-security.signature.properties"
                    value="clientKeystore.properties"/>
```

```
                    <entry key="ws-security.encryption.properties"
                       value="clientKeystore.properties"/>
                    <entry key="ws-security.encryption.username"
                       value="mystskey"/>
                  </map>
               </property>
            </bean>
         </entry>
      </jaxws:properties>
</jaxws:client>
```

The above example shows a configuration where the STS uses the UsernameToken profile to validate the client. It is assumed the keystore identified within clientKeystore.properties contains both the private key of the client and the public key (identified above as mystskey) of the STS; if not, create separate property files for the signature properties and the encryption properties, pointing to the keystore and truststore respectively.

Remember the jaxws:client createdFromAPI attribute needs to be set to true (as shown above) if you created the client programmatically via the CXF API's--i.e., Endpoint.publish() or Service.getPort().

This also works for "code first" cases as you can do:

```
STSClient sts = new STSClient(...);
sts.setXXXX(....)
.....
((BindingProvider)port).getRequestContext().
     put("ws-security.sts.client", sts);
```

Sample clientKeystore.properties format:

```
org.apache.ws.security.crypto.provider= \
org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=KeystorePasswordHere
org.apache.ws.security.crypto.merlin.keystore.alias=ClientKeyAlias
org.apache.ws.security.crypto.merlin.file=NameOfKeystore.jks
```

Indirect configuration based on endpoint name: If the runtime does not find a STSClient bean configured directly on the client, it checks the configuration for a STSClient bean with the name of the endpoint appended with ".sts-client". For example, if the endpoint name for your client is "{ ??? }TestEndpoint", then it can be configured as:

```
<bean name="{http://cxf.apache.org/}TestEndpoint.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient" abstract="true">
    <property name="wsdlLocation" value="WSDL/wsdl/trust.wsdl"/>
    <property name="serviceName" value=
        "{http://cxf.apache.org/securitytokenservice} ...
           SecurityTokenService"/>
    <property name="endpointName" value=
        "{http://cxf.apache.org/securitytokenservice} ...
           SecurityTokenEndpoint"/>
    <property name="properties">
        <map>
            <entry key="ws-security.signature.properties"
               value="etc/alice.properties"/>
            <entry key="ws-security.encryption.properties"
               value="etc/bob.properties"/>
            <entry key="ws-security.encryption.username"
               value="stskeyname"/>
        </map>
    </property>
```

```
</bean>
```

This properties configured in this example demonstrate STS validation of the client using the X.509 token profile. The abstract="true" setting for the bean defers creation of the STSClient object until it is actually needed. When that occurs, the CXF runtime will instantiate a new STSClient using the values configured for this bean.

Default configuration: If an STSClient is not found from the above methods, it then tries to find one configured like the indirect, but with the name "default.sts-client". This can be used to configure sts-clients for multiple services.

2.6.7.2. WS-Trust 1.4 Support

CXF provides limited support of WS-Trust 1.4. The currently supported features are listed below.

ActAs (2.2.10)

The ActAs capability allows an initiator to request a security token that allows it to act as if it were somebody else. This capability becomes important in composite services where intermediate services make additional requests on-behalf of the true initiator. In this scenario, the relying party (the final destination of an indirect service request) may require information about the true origin of the request. The ActAs capability allows an intermediary to request a token that can convey this information.

The following code fragment demonstrates how to use an interceptor to dynamically set the content of the ActAs element in the STS RST. The value may be a string containing well-formed XML or a DOM Element. The contents will be added to the RST verbatim. Note that this interceptor is applied to the secured client, the initiator, and not to the STSClient's interceptor chain.

```
public class ActAsOutInterceptor extends
    AbstractPhaseInterceptor<Message> {

    ActAsOutInterceptor () {
        // This can be in any stage before the WS-SP interceptors
        // setup the STS client and issued token interceptor.
        super(Phase.SETUP);
    }

    @Override
    public void handleMessage(Message message) throws Fault {

        message.put(SecurityConstants.STS_TOKEN_ACT_AS, ...);

    }
}
```

Alternatively, the ActAs content may be set directly on the STS as shown below.

```
<bean name="{http://cxf.apache.org/}TestEndpoint.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient" abstract="true">
   <property name="wsdlLocation" value="WSDL/wsdl/trust.wsdl"/>
   <property name="serviceName" value=
     "{http://cxf.apache.org/securitytokenservice}SecurityTokenService"/>
   <property name="endpointName" value=
     "{http://cxf.apache.org/securitytokenservice}SecurityTokenEndpoint"/>
   <property name="actAs" value="..."/>
   <property name="properties">
```

```
        <map>
           <entry key="ws-security.sts.token.properties"
               value="etc/bob.properties"/>
           <entry key="ws-security.callback-handler"
               value="interop.client.KeystorePasswordCallback"/>
           <entry key="ws-security.signature.properties"
               value="etc/alice.properties"/>
           <entry key="ws-security.encryption.properties"
               value="etc/bob.properties"/>
        </map>
    </property>
</bean>
```

2.7. CXF Customizations

2.7.1. Annotations

CXF provides several custom annotations that can be used to configure and customize the CXF runtime.

2.7.1.1. org.apache.cxf.feature.Features

The @Features annotation is used to add Features, something that is able to customize a Server, Client, or Bus, typically by adding capabilities. See the CXF Features List for those provided "out of the box" by CXF. You can also create your own features. In many cases, however, those features have Annotations themselves which can be used and provide greater control over configuration.

2.7.1.2. org.apache.cxf.interceptor.InInterceptors, org.apache.cxf.interceptor.OutInterceptors, org.apache.cxf.interceptor.OutFaultInterceptors, org.apache.cxf.interceptor.InFaultInterceptors

Add interceptors to the various chains used to process messages.

2.7.1.3. org.apache.cxf.annotations.WSDLDocumentation org.apache.cxf.annotations.WSDLDocumentationCollection (since 2.3)

For "java first" scenarios where the WSDL is derived from the Java interfaces/code, these annotations allow adding wsd:documentation elements to various locations in the generated wsdl.

For example:

```
@WebService
@WSDLDocumentationCollection(
```

```
    {
        @WSDLDocumentation("My portType documentation"),
        @WSDLDocumentation(value = "My top level documentation",
            placement = WSDLDocumentation.Placement.TOP),
        @WSDLDocumentation(value = "My binding doc",
            placement = WSDLDocumentation.Placement.BINDING)
    }
)
public interface MyService {

    @WSDLDocumentation("The docs for echoString")
    String echoString(String s);

}
```

2.7.1.4. org.apache.cxf.annotations.SchemaValidation (since 2.3)

Turns on SchemaValidation for messages. By default, for performance reasons, CXF does not validate message against the schema. By turning on validation, problems with messages not matching the schema are easier to determine.

2.7.1.5. org.apache.cxf.annotations.DataBinding (since 2.2.4)

Sets the DataBinding class that is associated with the service. By default, CXF assumes you are using the JAXB data binding. However, CXF supports different databindings such as XMLBeans, Aegis, SDO, and possibly more. This annotation can be used in place of configuration to select the databinding class.

```
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface MyService {
    public commonj.sdo.DataObject echoStruct(
        commonj.sdo.DataObject struct
    );
}
```

2.7.1.6. org.apache.cxf.annotations.Logging (since 2.3)

Turns on logging for the endpoint. Can be used to control the size limits of what gets logged as well as the location. It supports the following attributes:

limit	Sets the size limit after which the message is truncated in the logs. Default is 64K
inLocation	Sets the location to log incoming messages. Can be <stderr>, <stdout>, <logger>, or a file: URL. Default is <logger>
outLocation	Sets the location to log outgoing messages. Can be <stderr>, <stdout>, <logger>, or a file: URL. Default is <logger>

```
@Logging(limit=16000, inLocation="<stdout>")
public interface MyService {
```

```
    String echoString(String s);
}
```

2.7.1.7. org.apache.cxf.annotations.GZIP (since 2.3)

Enables GZIP compression of on-the-wire data. Supported attributes:

threshold	the threshold under which messages are not gzipped

GZIP is a negotiated enhancement. An initial request from a client will not be gzipped, but an Accept header will be added and if the server supports it, the response will be gzipped and any subsequent requests will be.

2.7.1.8. org.apache.cxf.annotations.FastInfoset (since 2.3)

Enables FastInfoset of on-the-wire data. Supported attributes:

force	forces the use of fastinfoset instead of negotiating. Default is false

FastInfoset is a negotiated enhancement. An initial request from a client will not be in fastinfoset, but an Accept header will be added and if the server supports it, the response will be in fastinfoset and any subsequent requests will be.

2.7.1.9. org.apache.cxf.annotations.EndpointProperty org.apache.cxf.annotations.EndpointProperties (since 2.3)

Adds a property to an endpoint. Many things such as WS-Security related things and such can be configured via endpoint properties. Traditionally, these would be set via the <jaxws:properties> element on the <jaxws:endpoint> element in the spring config, but these annotations allow these properties to be configured into the code.

```
@WebService
@EndpointProperties(
    {
       @EndpointProperty(key = "my.property", value="some value"),
       @EndpointProperty(key = "my.other.property",
          value="some other value"),
    })
public interface MyService {
    String echoString(String s);
}
```

2.7.1.10. org.apache.cxf.annotations.Policy org.apache.cxf.annotations.Policies (since 2.3)

Used to attach WS-Policy fragments to a service or operation. The Policy supports the attributes:

uri	**REQUIRED** the location of the file containing the Policy definition

includeInWSDL	Whether to include the policy in the generated WSDL when generating a wsdl. Default it true
placement	Specify where to place the policy
faultClass	if placement is a FAULT, this specifies which fault the policy would apply to

Note

When using a custom Spring configuration, you'll need to import META-INF/cxf/cxf-extension-policy.xml

```
@Policies({
    @Policy(uri = "annotationpols/TestInterfacePolicy.xml"),
    @Policy(uri = "annotationpols/TestImplPolicy.xml",
       placement = Policy.Placement.SERVICE_PORT),
    @Policy(uri = "annotationpols/TestPortTypePolicy.xml",
       placement = Policy.Placement.PORT_TYPE)
}
)
@WebService
public static interface TestInterface {
    @Policies({
        @Policy(uri = "annotationpols/TestOperationPolicy.xml"),
        @Policy(uri = "annotationpols/TestOperationInputPolicy.xml",
           placement = Policy.Placement.BINDING_OPERATION_INPUT),
        @Policy(uri = "annotationpols/TestOperationOutputPolicy.xml",
           placement = Policy.Placement.BINDING_OPERATION_OUTPUT),
        @Policy(uri = "annotationpols/TestOperationPTPolicy.xml",
           placement = Policy.Placement.PORT_TYPE_OPERATION),
        @Policy(uri = "annotationpols/TestOperationPTInputPolicy.xml",
           placement = Policy.Placement.PORT_TYPE_OPERATION_INPUT),
        @Policy(uri = "annotationpols/TestOperationPTOutputPolicy.xml",
           placement = Policy.Placement.PORT_TYPE_OPERATION_OUTPUT)
    }
    )
    int echoInt(int i);
}
```

2.7.1.11. org.apache.cxf.annotations.UseAsyncMethod (since 2.6.0)

Used on the JAX-WS service implementation object to mark a method as preferring the 'async' version of the method instead of the synchronous version. With JAX-WS, services default to the synchronous methods that require the returning value to be returned from the method. By marking a method with the @UseAsyncMethod annotation, if the transport supports it, CXF will call the async version that takes an AsynHandler object and the service can call that handler when the response is ready. If the transport does not support the CXF continuations, the synchronous method will be called as normal.

2.7.2. Dynamic Clients

The usual way to construct a web service client is to include the Java interface for the service (the SEI) and any classes that are used for inputs and output in the client application. This is not always desirable or practical.

CXF supports several alternatives to allow an application to communicate with a service without the SEI and data classes. JAX-WS specified the Dispatch API, as well as the Provider interface for reading and writing XML. This page, however, describes the dynamic client facility of CXF. With dynamic clients, CXF generates SEI and bean classes at runtime, and allows you to invoke operations via APIs that take Objects, or by using reflection to call into full proxies.

Note that, in general, CXF only supports WSI-BP services. If you attempt to create a dynamic client for a WSDL that uses features outside of WSI-BP, CXF may throw an exception.

2.7.2.1. DynamicClientFactory and JaxWsDynamicClientFactory

CXF provides two factory classes for dynamic classes. If your service is defined in terms of JAX-WS concepts, you should use the JaxWsDynamicClientFactory. If you do not want or need JAX-WS semantics, use the DynamicClientFactory. The remainder of this page uses the JaxWs version.

Let's pretend for a moment that you have a WSDL which defines a single operation "echo" which takes an input of a string and outputs a String. You could use the JaxWsDynamicClientFactory for it like this:

```
JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
Client client = dcf.createClient("echo.wsdl");

Object[] res = client.invoke("echo", "test echo");
System.out.println("Echo response: " + res[0]);
```

Many WSDLs will have more complex types though. In this case the JaxWsDynamicClientFactory takes care of generating Java classes for these types. For example, we may have a People service which keeps track of people in an organization. In the sample below we create a Person object that was generated for us dynamically and send it to the server using the addPerson operation:

```
JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
Client client = dcf.createClient("people.wsdl", classLoader);


Object person = Thread.currentThread().getContextClassLoader().
   loadClass("com.acme.Person").newInstance();

Method m = person.getClass().getMethod("setName", String.class);
m.invoke(person, "Joe Schmoe");

client.invoke("addPerson", person);
```

You may be asking yourself the following question: "Where did the class name 'com.acme.Person' come from?"

One way to get the class names is to run wsdl2java and examine the results. The dynamic client factory uses the same code generator as that tool. Another way is to walk the CXF service model. This has the advantage that it delivers Class<?> objects directly, so you don't need to obtain the correct class loader reference and run loadClass.

The wsdl_first_dynamic_client sample uses this approach. Read the file 'ComplexClient.java' to see the process, which uses some of the java.bean classes to simplify the code slightly.

Note

The JaxWsDynamicClientFactory sets the Thread context ClassLoader to a new ClassLoader that contains the classes for the generated types. If you need the original ClassLoader, make sure you save it prior to calling createClient.

2.8. CXF Command-Line Tools

The most up-to-date instructions for building SOAP web-services are maintained in the CXF User's Guide. In particular, see the wsdl2java utility for contract-first development and java2ws tool for the start-from-Java approach. For Maven-based projects, CXF offers plugins for both WSDL-first (cxf-codegen-plugin) and Java-first (cxf-java2ws-plugin) development.

2.8.1. WSDL to Java

wsdl2java creates JAX-WS and JAXB (or other databinding framework) objects from a service WSDL. It has the following parameters:

Parameter	Option
-h	Displays the online help for this utility and exits.
-fe frontend-name	Specifies the frontend. Default is JAXWS. Currently supports only JAXWS frontend and a "jaxws21" frontend to generate JAX-WS 2.1 compliant code.
-db databinding-name	Specifies the databinding. Default is jaxb. Currently supports jaxb, xmlbeans, sdo (sdo-static and sdo-dynamic), and jibx.
-wv wsdl-version	Specifies the wsdl version .Default is WSDL1.1. Currently supports only WSDL1.1 version.
-p [wsdl-namespace=] PackageName	Specifies zero, or more, package names to use for the generated code. Optionally specifies the WSDL namespace to package name mapping.
-sn service-name	The WSDL service name to use for the generated code.
-b binding-name	Specifies JAXWS or JAXB binding files or XMLBeans context files. Use multiple -b flags to specify multiple entries.
-catalog catalog-file-name	Specify catalog file to map the imported wsdl/schema
-d output-directory	Specifies the directory into which the generated code files are written.
-compile	Compiles generated Java files.
-classdir compile-class-dir	Specifies the directory into which the compiled class files are written.
-client	Generates starting point code for a client mainline.
-server	Generates starting point code for a server mainline.
-impl	Generates starting point code for an implementation object.
-all	Generates all starting point code: types, service proxy, service interface, server mainline, client mainline, implementation object, and an Ant build.xml file.
-ant	Specify to generate an Ant build.xml script.
-autoNameResolution	Automatically resolve naming conflicts without requiring the use of binding customizations.
-defaultValues=[DefaultValueProvider impl]	Specifies that default values are generated for the impl and client. You can also provide a custom default value provider. The default provider is RandomValueProvider
-nexclude schema-namespace [=java-packagename]	Ignore the specified WSDL schema namespace when generating code. This option may be specified multiple times. Also, optionally specifies the Java package name used by types described in the excluded namespace(s).
-exsh (true/false)	Enables or disables processing of implicit SOAP headers (i.e. SOAP headers defined in the wsdl:binding but not wsdl:portType section.) Default is false.

Parameter	Option
-dns (true/false)	Enables or disables the loading of the default namespace package name mapping. Default is true and [http://www.w3.org/2005/08/addressing=org.apache.cxf.ws.addressing] namespace package mapping will be enabled.
-dex (true/false)	Enables or disables the loading of the default excludes namespace mapping. Default is true.
-validate	Enables validating the WSDL before generating the code.
-keep	Specifies that the code generator will not overwrite any preexisting files. You will be responsible for resolving any resulting compilation issues.
-wsdlLocation wsdlLocation	Specifies the value of the @WebServiceClient annotation's wsdlLocation property.
-xjc<xjc args>	Specifies a comma separated list of [arguments|https://jaxb.dev.java.net/nonav/2.2/docs/xjc.html] that are passed directly to the XJC processor when using the JAXB databinding. A list of available XJC plugins can be obtained using -xjc -X.
-noAddressBinding	For compatibility with CXF 2.0, this flag directs the code generator to generate the older CXF proprietary WS-Addressing types instead of the JAX-WS 2.1 compliant WS-Addressing types.
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.
-exceptionSuper	superclass for any fault beans generated from wsdl:fault elements (defaults to java.lang.Exception)
-reserveClass classname	Used with -autoNameResolution, defines a class names for wsdl-to-java not to use when generating classes. Use this option multiple times for multiple classes.
-allowElementReferences	If true, disregards the rule given in section 2.3.1.2(v) of the JAX-WS 2.2 specification disallowing element references when using wrapper-style mapping.
-asyncMethods=foo,bar,...	List of subsequently generated Java class methods to allow for client-side asynchronous calls, similar to enableAsyncMapping in a JAX-WS binding file.
-bareMethods=foo,bar,...	List of subsequently generated Java class methods to have wrapper style (see below), similar to enableWrapperStyle in JAX-WS binding file.
-mimeMethods=foo,bar,...	List of subsequently generated Java class methods to enable mime:content mapping, similar to enableMIMEContent in JAX-WS binding file.
-faultSerialVersionUID <fault-serialVersionUID>	How to generate suid of fault exceptions. Use NONE, TIMESTAMP, FQCN, or a specific number. Default is NONE.
-mark-generated	Adds the @Generated annotation to classes generated.
wsdlurl	The path and name of the WSDL file to use in generating the code.

Examples of wsdl2java in use:

- **`wsdl2java HelloWorld.wsdl`**
- **`wsdl2java -p com.mycompany.greeting Greeting.wsdl`**
- **`wsdl2java -client HelloWorld.wsdl`**

2.8.2. Java to WS

2.8.2.1. Name

`java2ws` - uses a Web service endpoint's implementation (SEI) class and associated types classes to generate a WSDL file, wrapper bean ,server side code to start the web service and client side code.

2.8.2.2. Synopsis

```
java2ws -databinding <jaxb or aegis> -frontend <jaxws or simple>
           -wsdl -wrapperbean -client -server -ant -o <output-file>
           -d <resource-directory> -classdir <compile-classes-directory>
           -cp <class-path> -soap12 -t <target-namespace>
           -beans <ppathname of the bean definition file>*
           -address <port-address> -servicename <service-name>
           -portname <port-name> -createxsdimports -h -v -verbose
           -quiet {classname}
```

2.8.2.3. Description

java2ws uses a Web service endpoint's implementation (SEI) class and associated types classes to generate a WSDL file, wrapper bean ,server side code to start the web service and client side code.

2.8.2.4. Options

The options used to manage the code generation process are reviewed in the following table.

Option	Interpretation
`-?` , `-h` , `-help`	Displays the online help for this utility and exits.
-o	Specifies the name of the generated WSDL file.
-databinding	Specify the data binding (aegis or jaxb). Default is jaxb for jaxws frontend, and aegis for simple frontend.
-frontend	Specify the frontend to use. jaxws and the simple frontend are supported.
-wsdl	Specify to generate the WSDL file.
-wrapperbean	Specify to generate the wrapper and fault bean
-client	Specify to generate client side code
-server	Specify to generate server side code
-ant	Specify to generate an Ant build.xml script
-cp	Specify the SEI and types class search path of directories and zip/jar files.
-soap12	Specifies that the generated WSDL is to include a SOAP 1.2 binding.
-t	Specifies the target namespace to use in the generated WSDL file.

Option	Interpretation
-servicename	Specifies the value of the generated service element's name attribute.
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.
-s	The directory in which the generated source files(wrapper bean ,fault bean ,client side or server side code) are placed.
-classdir	The directory in which the generated sources are compiled into. If not specified, the files are not compiled.
-portname	Specify the port name to use in the generated wsdl.
-address	Specify the port address.
-beans	Specify the pathname of a file defining additional Spring beans to customize databinding configuration.
-createxsdimports	Output schemas to separate files and use imports to load them instead of inlining them into the wsdl.
-d	The directory in which the resource files are placed, wsdl file will be placed into this directory by default
classname	Specifies the name of the SEI class.

You must include the `classname` argument. All other arguments are optional and may be listed in any order. This tool will search and load the service endpoint class and types classes. Make certain these classes are on the `CLASSPATH` or in a location identified through the `-cp` flag. If none of "-wsdl , - wrapperbean, -client, -server" flags are specified, java2ws will generate nothing.

2.8.2.5. Examples

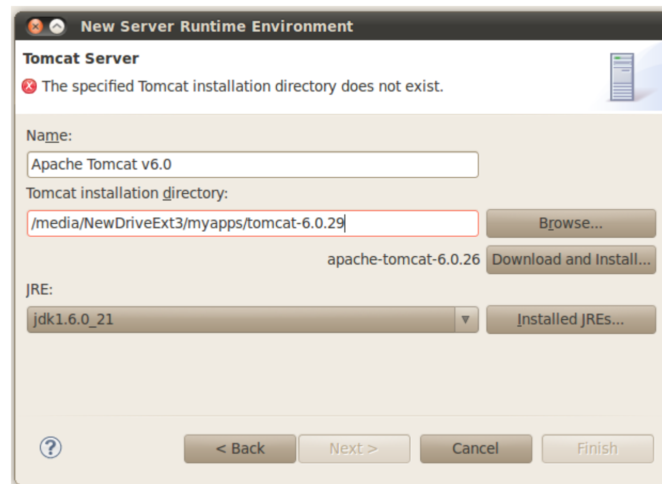
```
java2ws -wsdl -d ./resource org.apache.hello.Greeter
java2ws -cp ./tmp org.apache.hello.Greeter -wsdl
java2ws -o hello.wsdl -wsdl org.apache.hello.Greeter
java2ws -client -server -s ./src org.apache.hello.Greeter
java2ws -wrapperbean -classdir ./classes org.apache.hello.Greeter
```

2.9. JAX-WS Development With Eclipse

Development using the Eclipse IDE is another option. The Java Enterprise Edition version of Eclipse allows for rapid development, testing, and debugging, making it a compelling environment for web service development. Steps to follow for building web services with Eclipse:

1. **Download and attach Tomcat to Eclipse.**

   Tomcat needs to be linked into the IDE so Eclipse can start and stop the servlet container as well as deploy web services to it. (See the Eclipse/Tomcat FAQ to learn more.) Download and extract the Tomcat binary into its own directory if you haven't already. Next, from the Eclipse Preferences window (Menu item Windows->Preferences), select the Server -> Runtime Environments section and add your Tomcat installation to the IDE:

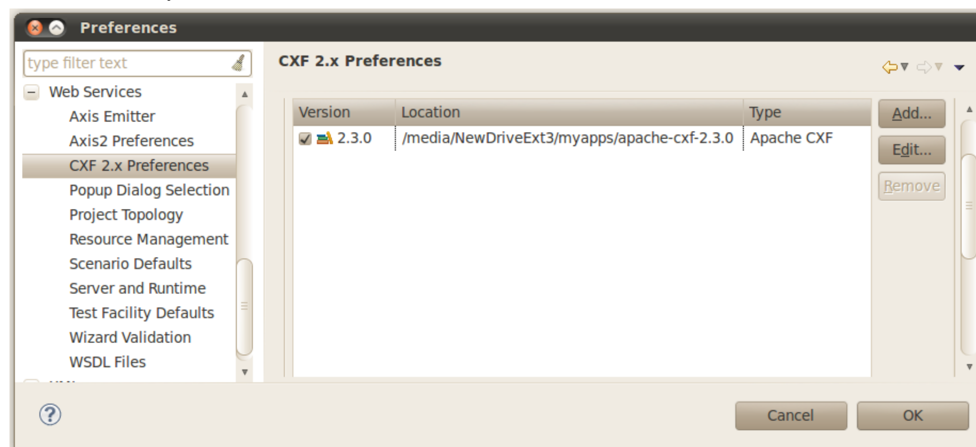


Attaching Tomcat To Eclipse

Ensure that the JRE being used is an actual Java Development Kit (JDK), which Tomcat needs in order to compile JSPs and potentially other source files.

2. **Download and attach CXF to Eclipse.**

Similar to Tomcat, download the latest release version of Apache CXF and expand into a directory on your computer. Next, from Eclipse Preferences, open up the Web Services -> CXF 2.x Preferences section and enter the CXF directory as shown below.

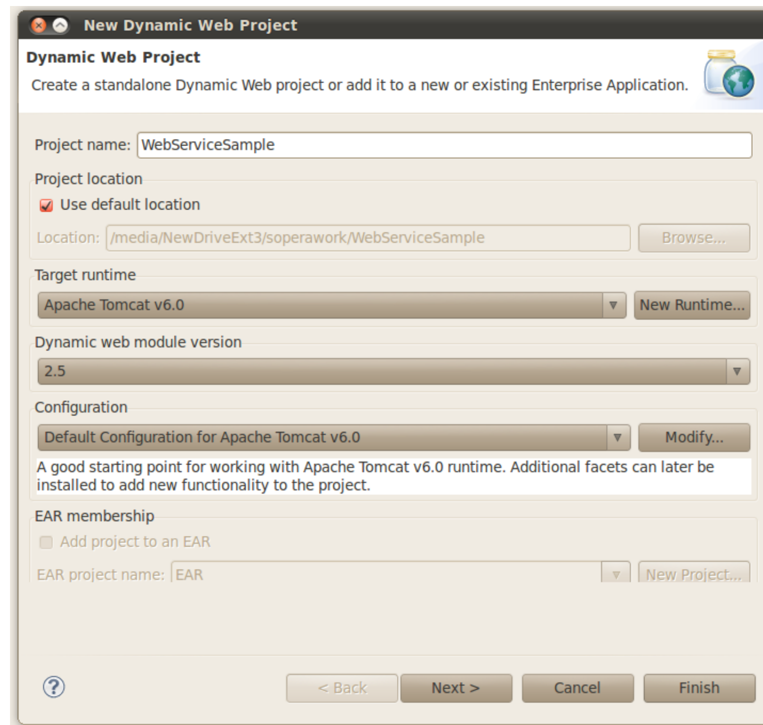


Attaching CXF to Eclipse

Also, in the Web Services -> Server and Runtime category, make sure the Server and Web Service Runtimes are set to your version of Tomcat and CXF respectively.

3. **Create a dynamic web project.**

We will next create a Eclipse dynamic web project to contain the web service provider and client. From the Eclipse File menu, Select New -> Dynamic Web Project (or New -> Other, and from the subsequent popup dialog Web -> Dynamic Web Project). Give the project any desired name ("WebServiceSample" is used below), make sure the proper Tomcat server is selected, and then press the Finish button as illustrated:



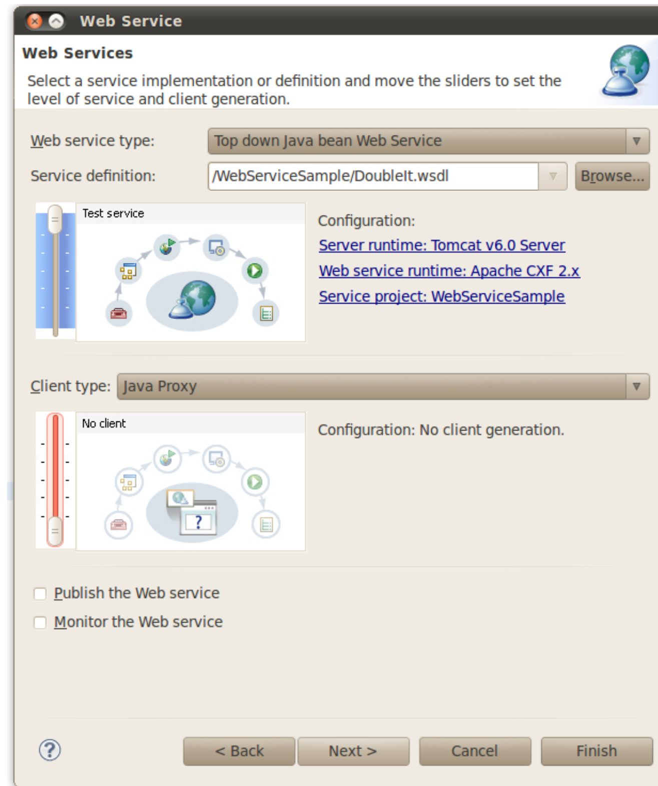
Creating an Eclipse Dynamic Web Project

4. **Create the web service provider.**

   We'll start from a WSDL-first approach.

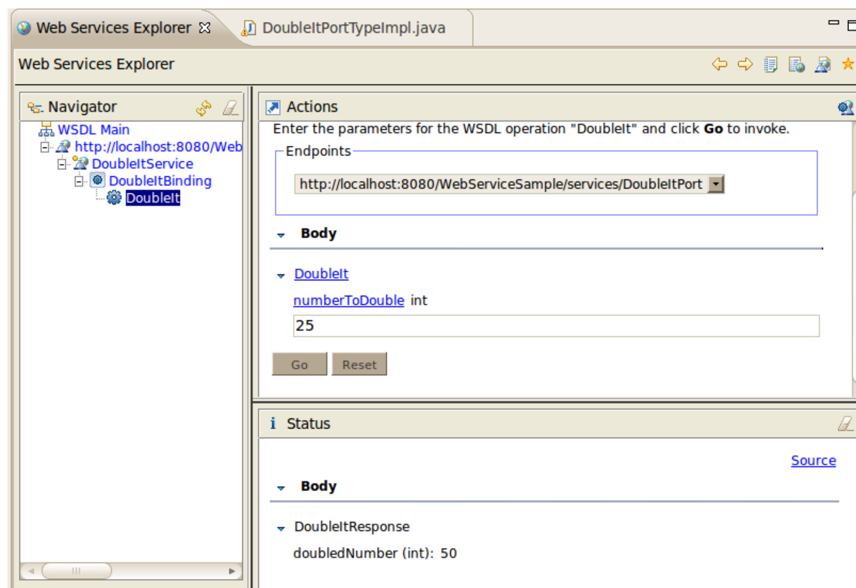
   a. First we'll create the WSDL. From the Eclipse Project Explorer, right-click WebServiceSample, select New -> File, and enter DoubleIt.wsdl as the file name. Next, copy-and-paste the DoubleIt.wsdl from the CXF repository into this new file.

   b. Right-click WebServiceSample again and select New -> Other, and Web Services -> Web Service from the New dialog. Choose the Top Down (i.e., WSDL-first) web service approach and the DoubleIt.wsdl file you just created as the Service definition. Make sure the server and web service runtime (i.e., CXF) are correct. On the server thumbwheel, select the highest setting ("Test Service") as shown below.



New Web Service

Click finish. Eclipse will deploy the web service on Tomcat and provide a Web Services Explorer test window where you can enter numbers and see the doubled result, as shown below. However, the web service presently will just return a hardcoded number, we'll need to modify the web service provider to return the desired result, which we'll do next.



Web Services Explorer test window

c. From the Project Explorer, under WebServiceSample, open up `Java Resources: src` and open the DoubleItPortTypeImpl.java file. Change the business logic of the doubleIt method to return twice the input parameter (i.e., " `int _return = numberToDouble * 2;` ") and save the file. Once

saved, Eclipse automatically recompiles the class and redeploys it to the Tomcat server. You can return to the Web Services Explorer test window, enter a number and now see the properly doubled result.

5. **Create the web service client.**

We'll now use Eclipse to create a Java-based web service client. Right-click WebServiceSample from the Project Explorer and select New -> Other, and from the New dialog Web Services -> Web Service Client. From the window that appears enter the WSDL file and raise the thumbwheel all the way up to "test", as shown below. Then select "Next" and enter another package name for the client (say "org.example.doubleit.client") so it does not conflict with the web service provider package. Click Finish.

New web service client window

Two modifications to the autogenerated client code will need to be made. First, in the org.example.doubleit.client.DoubleItService file, the wsdlLocation and url fields highlighted below will need to be updated to the actual endpoint URL Eclipse is using for the web service provider. You can find the value by looking at the wsdlLocation field in the DoubleItService class of the web service provider (it's also viewable in the Web Services Explorer test view).

```
@WebServiceClient(name = "DoubleItService",
   wsdlLocation = "http://localhost:8080/WebServiceSample/"
      + services/DoubleItPort?wsdl",
   targetNamespace = "http://www.example.org/DoubleIt")
public class DoubleItService extends Service {

    public final static URL WSDL_LOCATION;
    public final static QName SERVICE =
        new QName("http://www.example.org/DoubleIt",
           "DoubleItService");
    public final static QName DoubleItPort =
        new QName("http://www.example.org/DoubleIt",
           "DoubleItPort");

    static {
        URL url = null;
        try {
            url = new URL(
               "http://localhost:8080/WebServiceSample/" +
               "services/DoubleItPort?wsdl");
        } catch (MalformedURLException e) {
```

```
                System.err.println(
                    "Can not initialize the default wsdl"
                    + from http://localhost:8080/doubleit/" +
                       "services/doubleit?wsdl");
                // e.printStackTrace();
            }
            WSDL_LOCATION = url;
        }
    ...
```

Second, in the DoubleItPortType_DoubleItPort_Client class modify the value of the _doubleIt_numberToDouble constant to a number you wish to double (say "55") and then save your change. Then right-click the file in the Eclipse editor and select Run As -> Java Application. You should see the result of the web service call (110) in the Eclipse Console window:

```
...
INFO: Creating Service {http://www.example.org/DoubleIt}DoubleItService
from WSDL: http://localhost:8080/WebServiceSample/services/DoubleItPort?wsdl
Invoking doubleIt...
doubleIt.result=110
```

Chapter 3. JAX-RS Development

3.1. JAX-RS Overview

CXF implements JAX-RS 1.1 API and passes the JAX-RS 1.1 TCK. This section provides the overview of the JAX-RS specification and API. You are encouraged to read JAX-RS spec (html version) to find out information not covered by this documentation. The specification introduces such terms as root resources, resource methods, sub-resources and sub-resource locators, message body readers and writers, contexts.
Please check the CXF JAX-RS documentation for an up-to-date guide on the new features and improvements.

3.1.1. Root Resources and Sub Resources

The Java class annotated with the @Path annotation represents a root resource class. The JAX-RS application may have more than one root resource class. Each root resource has one or more resource methods handling requests directly or delegating to subresources. Delegating resource methods are called subresource locators. Subresources are like root resource classes in that they can handle the request or delegate further with the exception being that their class-level @Path annotation is ignored during the method resolution. Effectively, the root resource class is a top level handler supporting a specific URI space of the RESTful web application. The default lifecycle of the root resource is per-request, that is, a new instance is created during every request. There is a number of options available to turn a root resource into the singleton.

Here is a sample JAX-RS root resource class:

```
package demo.jaxrs.server;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
```

```
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.core.UriBuilder;

@Path("/customerservice/")
@Produces("application/xml")
public class CustomerService {

    @Context
    private UriInfo uriInfo;

    @GET
    public Customers getCustomers() {
        return findAllCustomers();
    }

    @GET
    @Path("{id}")
    @Produces({"application/json", "application/json"})
    public Customer getCustomer(@PathParam("id") long id) {
        return findCustomer(id);
    }

    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public Response updateCustomer(@PathParam("id") Long id, Order order) {
        // update customer
        return Response.ok().build();
    }

    @POST
    public Response addCustomer(Customer customer) {
        // Create a new Customer resource and return a Location URI
        // pointing to it
        // Get the URIBuilder initialized with the base URI. For example,
        // if the servlet is listening on http://localhost:8080/webapp
        // then the builder will be set to this base URI
        UriBuilder builder = uriInfo.getBaseUriBuilder();

        // Append the path value which this root resource class uses and
        // the id of the new customer
        // The newCustomerURI will represent a link to the new customer
        // resource.
        URI newCustomerURI = builder.path(CustomerService.class).path(
           customer.getId()).build();

        return Response.status(201).location(newCustomerURI).build();
    }


    @DELETE
    @Path("{id}")
    public void deleteCustomer(@PathParam("id") String id) {
       // delete the customer
    }
```

```
    @Path("{id}/orders/{orderId}/")
    public Order getOrder(@PathParam("id") long customerId,
       @PathParam("orderId") long orderId) {
       return findCustomer(customerId).getOrder(orderId);
    }

    private Customers findAllCustomers() {
        // find all the customers
    }

    private Customer findCustomer(long id) {
        // find the customer with the given id
    }
}
```

Customer resource class can handle requests with URI containig "/customerservice" path segments. For example, requests with the "http://localhost/8080/webapp/customerservice" URI and the GET HTTP verb will be handled by the getCustomers() method while requests with the "http://localhost/8080/webapp/customerservice/123" will be handled by one of the methods containing the @Path("{id}") annotation, depending on the HTTP verb such as GET, PUT or DELETE being used.

The getOrder() method is a subresource locator because it has no HTTP verb annotation, only the @Path annotation. The root resource may act as a sub-resource if one of its subresource locators delegates to it. For example, requests with the "http://localhost/8080/webapp/customerservice/123/orders/356/state" URI and the GET HTTP verb will be handled by the Order subresource instance. After the subresource locator getOrder() has returned, the runtime will use the remaining unmatched "/state" path segment to find the matching Order method.

3.1.2. Path, HTTP Method and MediaType annotations

The @Path annotation is applied to root resource classes or methods. The @Path value is a relative URI with one or more path segments. The value may be literal, for example, "customers/1/orders" or parameterized and contain one or more URITemplate variables: "customers/{id}/orders/{orderid}". In the latter example the "id" and "orderid" are template variables. The actual values will be available to resource methods with the help of @PathParam annotations. Template variables may also contain explicit regular expressions, for example: "/{id:.*}/" where "id" is the name of the variable and the expression after the semicolon is a custom regular expression.

When @Path is not available on the resource method then it is assumed to be equal to "/".

When selecting the root resource class, the runtime will try to match the relative request path available to it after the HTTP servlet matched the absolute request URI with the @Path value of the root resource. If the current root resource is not matched then the next root resource, if available, will be tried. After the successful match, the resource method will be selected given the remaining relative path and the HTTP method

Every resource method which is not a subresource locator must have an annotation indicating which HTTP verb is supported by this method. JAX-RS provides @GET, @DELETE, @POST, @PUT, @HEAD and @OPTIONS. Annotations for supporting other HTTP verbs can be created. The current resource method may only be selected if the value of its HTTP Method annotation is equal to the request HTTP verb.

The @Consumes annotation is used to specify the acceptable format of the request message. If it is not available on the resource method then it is inherited from a class, and if it's not available on the class then it's indirectly inherited from a corresponding JAX-RS MessageBodyReader, if any. The default value is '*/*' but it is recommended that a concrete media type is specified.

When attempting to select a given resource method, the runtime will try to match the value of the Content-Type HTTP header with the @Consumes value. If the match is successful then it will try to find the JAX-RS MessageBodyReader which can read the request message and convert it into the method parameter representing a

request body. When looking for the reader, the runtime will also try to match the @Consumes value of the current reader with the request media type.

The @Produces annotation is used to specify the format of the response. If it is not available on the resource method then it is inherited from a class, and if it's not available on the class then it's indirectly inherited from a corresponding JAX-RS MessageBodyWriter, if any. The initial default value is '*/*' which will set to "application/octet-stream" before writing the actula response, thus it is recommended that a concrete media type is specified. For example, getCustomers() method inherits @Produces annotation from its class, while getCustomer() method overrides it with its own value.

When attempting to select a given resource method, the runtime will try to match the value of the Accept HTTP header with the @Produces value. The Accept and @Produces values may list multiple media types - the most preferred media type is chosen during the intersection. Additionally, when the resource method returns an object, the runtime will try to find the suitable JAX-RS MessageBodyWriter. When looking for the writer, the runtime will also try to match the @Produces value of the current writer with the media type obtained during the earlier intersection.

3.1.3. Request Message, Parameters and Contexts

Usually, the resource method has one or more parameters representing a request message body, HTTP headers and different parts of the request URI. The resource method may only have one parameter representing a message body (and none in case of empty requests) but may have many parameters representing the headers or URI parts. @PathParam, @MatrixParam and @QueryParam capture various URI-related values while @HeaderParam and @Cookie capture various HTTP header values. @FormParam or JAX-RS MultivaluedMap can be used to capture name and value pairs of the form sequences.

The method parameter which has no JAX-RS annotations such as @PathParam does represent a request message. JAX-RS MessageBodyReader providers are responsible for reading the input stream and converting the data into this parameter object. JAXB-annotated, JAXP Source, InputStream and some other types are supported out of the box.

Method parameters representing URI parts or HTTP headers are the ones which have annotations such as @PathParam. These parameters are not handled by MessageBodyReader providers. The JAX-RS specification has the conversion rules which instruct the runtime to check the constructors or factory methods accepting Strings.

JAX-RS also introduces Contexts. Contexts represent the state of the request. UriInfo, HttpHeaders, SecurityContext, Request, HttpServeltRequest, etc are contexts which can be injected into the fields of the JAX-RS root resource instance or passed in as method parameters annotated with the @Context annotation. Contexts such as UriInfo, HttpHeaders and Request make it easy to introspect and use the information contained in the current request URI and headers while SecurityContext can be used to check if the current user is in the provided role. Injecting contexts in the fields is usually recommended as it may help simplify the method signatures, for example, given that the query values can be found from the UriInfo context adding explicit @QueryParam parameters to method signatures may not be needed. The runtime will ensure that the contexts injected into the fields of the singleton root resource are thread-safe. For example, here is how a resource method may look like before the introduction of contexts:

```
@Path("/")
public class BookResource {

    /**
     * Find the books with the id greater or equal to the passed id
     * and the authors first name equal to the passed name
     */
    @GET
    public Response findBook(@QueryParam("id") Long id,
        @QueryParam("name") String name) {
```

```
        // find the books
    }

}
```

In the above example, a method contains 2 query parameters. The alternative approach is to introduce a UriInfo context:

```
@Path("/")
public class BookResource {
    @Context
    private UriInfo context;
    /**
     * Find the books as requested by the user query
     */
    @GET
    public Response findBook() {
        MultivaluedMap<String, String> map = context.getQueryParameters();
        // iterate over the map, check all the query parameters,
        // find the books
    }

}
```

3.1.4. Responses from Resource Methods

The resource method may return a custom Java class, JAX-RS Response or void. JAX-RS MessageBodyWriter providers are responsible for writing the custom Java class instance to the output stream. The runtime will set the HTTP 200 status.

Returning JAX-RS Response is useful when no response message is available, the HTTP status code needs to be customized or some response HTTP headers set. Returning "void" will result in the HTTP 204 (No Content) status beiing returned to the client.

3.1.5. Exception Handling

JAX-RS resource methods and indeed the runtime may throw all kind of exceptions. JAX-RS WebApplicationException can be used to report an error from the application code or the custom JAX-RS providers. WebApplicationException can have the HTTP status, headers and error message included.

When the application code or provider throws the runtime or checked exception, what happens by default is that the uncaught exception is propagated up to the HTTP container level so that the existing exception-handling code in Servlet filters, if any, can deal with the exception. A JAX-RS ExceptionMapper implementation can be used instead to capture the exception and convert it into an appropriate HTTP response. For example, the following mapper catches a Spring Security exception and converts it into the HTTP 403 status:

```
package demo.jaxrs.server;

import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;

import org.springframework.security.access.AccessDeniedException;
```

```
public class SecurityExceptionMapper
    implements ExceptionMapper<AccessDeniedException> {

    public Response toResponse(AccessDeniedException exception) {
        return Response.status(Response.Status.FORBIDDEN).build();
    }

}
```

3.1.6. Custom JAX-RS Providers

The custom ExceptionMapper demonstrated in the previous section is a custom JAX-RS provider. Custom MessageBodyReader, MessageBodyWriter and ExceptionMapper providers are most often used.

Custom MessageBodyReader and/or MessageBodyWriter is needed when the JAX-RS runtime needs to read the request messages which has the format not understood or mapped to/from Java classes in a way not supported by by its default readers and writers. Writing a custom reader and writer is not a complex process. For example, a custom MessageBodyReader should be able to answer if it supports reading a message with given MediaType into an object instance of some specific type; if the response is yes then the runtime will ask the provider to read the message from the given input stream.

3.2. Client API

JAX-RS 1.1 does not have a standard Client API. JAX-RS 2.0 will introduce a standard API. CXF JAX-RS offers an HTTP-centric and Proxy-based API.CXF will continue supporting and enhancing its private API for those developers who may want to continue using it even after CXF JAX-RS implements JAX-RS 2.0.

3.2.1. HTTP Centric API

This API lets developers write the HTTP centric code. This fluent API makes it easy to modify the request URI, check the response status and headers. For example:

```
public class RESTClient {

    public void consumeRESTfulService(String baseAddress) {
        WebClient client = WebClient.create(baseAddress);
        client.type("application/xml").accept("application/xml");
        client.path("/book1");
        // get javax.ws.rs.core.Response directly and check the status,
        // headers and read the Response.getEntity() input stream
        Response response = client.get();
        // back to the base address
        client.back(true);
        client.path("/book2");

        // get a typed response
        Book response = client.get(Book.class);
    }
```

```
}
```

3.2.2. Proxy API

Proxy based API lets reuse the existing JAX-RS annotated interfaces or implementation classes as proxies. The proxy call will lead to a proper URI and HTTP headers passed to the server, exactly the way the JAX-RS annotated service expects. Proxies can also check the actual HTTP Response, modify request URI and headers, as well as be converted to WebClients. For example, given the following sample interface and the client code:

```
@Path("books")
public interface BookStore {

    @GET
    @Path("{id}")
    @Produces("application/xml")
    Book getBook(@PathParam("id") long id);

    @POST
    @Consumes("application/xml")
    Response addBook(Book newBook);

    @Path("all")
    BookStore getStore();

    @GET
    Books getAll();
}

public class RESTClient {

    public void consumeRESTfulService(String baseAddress) {
        BookStore proxy =
            JAXRSClientFactory.create(baseAddress, BookStore.class);

        // get a single book
        Book book = proxy.getBook(1);

        // further introspect the response headers
        Client theClient = WebClient.getClient(proxy);
        javax.ws.rs.core.Response response = theClient.getResponse();
        checkResponseHeaders(response.getMetadata());

        // add new book
        Response addNewBookResponse = proxy.addBook(new Book(
           "User Guide"));
        if (201 == addNewBookResponse.getStatus()) {
           URI bookResourceLocation =
               URI.create(addNewBookResponse.getMetadata().getFirst(
                  "Location"));
           Book theBookProxy = JAXRSClientFactory.create(
               bookResourceLocation, Book.class);
           theBookProxy.getName();
        } else {
           reportUnexpectedStatus(addNewBookResponse);
        }
```

```
        // get all books
        BookStore proxy2 = proxy.getStore();
        Books books = proxy2.getAll();
    }

}
```

the proxy.getBook(1) call will lead to a 'GET http://localhost:8080/store/books/1' HTTP request being sent to the server, with Accept header set to application/xml.

Note the proxy.getStore() and proxy2.getAll() calls. The BookStore.getStore() resource method is a subresource locator as it has only a @Path annotation. This locator returns the BookStore root instance itself. Thus proxy.getStore() only returns a proxy and the subsequent proxy2.getAll() call will lead to the 'GET http://localhost:8080/store/books/all' HTTP request being sent.

3.2.3. Reading and Writing HTTP Messages

JAX-RS MessageBodyReader and MessageBodyWriter providers are used to deal with output and input parameters representing HTTP request and response messages, the same way it is done on the server side. Both WebClient and proxy clients can register custom readers and writers if needed, when dealing with custom formats or when the default providers shipped with CXF JAX-RS need to be configured.

For example, a WebClient or proxy call may return a javax.ws.rs.Response instance which usually implies that the client wishes to deal with the HTTP response directly: the status and headers are checked and then the response InputStream returned by default from Response.getEntity() is manually read; in fact, in case of proxies, Response is something a proxy needs to deal with whenever a current proxy method is typed to return Response. Often the Response.getEntity() InputStream can be read by the registered JAX-RS MessageBodyReader providers. This example shows how registereing a custom CXF JAX-RS ResponseReader can help:

```
@Path("books")
public interface BookStore {
    @GET
    @Path("{id}")
    @Produces("application/xml")
    Response getBook(@PathParam("id") long id);
}

public class RESTClient {

    public void consumeRESTfulService(String baseAddress) {
        org.apache.cxf.jaxrs.client.ResponseReader responseReader =
          new org.apache.cxf.jaxrs.client.ResponseReader(Book.class);

        BookStore client = JAXRSClientFactory.create(
             baseAddress,
             BookStore.class,
             java.util.Collections.singletonList(responseReader));

        // get a single book
        Response response = client.getBook(1);
        checkStatusAndHeaders(response);
        Book book = (Book)response.getEntity();

    }
```

```
}
```

ResponseReader a custom JAX-RS MessageBodyReader provider but it delegates further to other registered providers which can handle reading the InputStream with a given Content-Type into a Book instance.

3.2.4. Exception Handling

WebClient or proxy based code dealing with explicit javax.ws.rs.core.Response responses is expected to handle the error HTTP responses itself. Usually, the code will check the response status and if the status indicates the error then some action is taken, for example, an exception can be thrown.

If the custom class such as Book is expected to be returned then org.apache.cxf.jaxrs.client.ServerWebApplicationException will be thrown in case of the HTTP response status being equal or greater than 400.

When working with proxies and in particular with methods which may throw custom checked exceptions, one or more org.apache.cxf.jaxrs.client.ResponseExceptionMapper providers can be registered during the proxy creation. These mappers can convert javax.ws.rs.core.Responses into custom exceptions as required by a given method signature.

Irrespectively of whether the Response or concrete custom class is expected in response to a given call, org.apache.cxf.jaxrs.client.ClientWebApplicationException will be thrown if the invocation has failed on the client side, for example, no connection has been established, no MessageBodyWriter has been found to write the request message or no MessageBodyReader has been found to read the response message, all on the client side.

3.3. Working With Attachments

CXF JAX-RS provides a comprehensive support for reading and writing multipart messages. Regular multiparts and XOP attachments are supported.

3.3.1. Reading Attachments

Individual parts can be mapped to StreamSource, InputStream, DataSource or custom Java types for which message body readers are available.

For example:

```
@POST
@Path("/books/jaxbjson")
@Produces("text/xml")
public Response addBookJaxbJson(
        @Multipart(value = "rootPart", type = "text/xml") Book2 b1,
        @Multipart(value = "book2", type = "application/json") Book b2)
        throws Exception {
}
```

Note that in this example it is expected that the root part named 'rootPart' is a text-xml Book representation, while a part named 'book2' is a Book JSON sequence.

All attachment parts can be accessed as a list of CXF JAX-RS Attachment objects with every Attachment instance providing all the information about a specific part. Similarly, the whole request body can be represented as a CXF JAX-RS MultipartBody:

```
@POST
public void addAttachments(MultipartBody body) throws Exception {
List<Attachment> all = body.getAllAtachments();
Attachment att = body.getRootAttachment();
}
```

When handling complex multipart/form-data submissions (such as those containing files) MultipartBody (and Attachment) need to be used directly.

When working with either List of Attachments or MultipartBody, one may want to process the individual parts with the help of some custom procedures. It is also possible to do the following:

```
@POST
public void addAttachments(MultipartBody body) throws Exception {
    Book book = body.getAttachmentObject("bookPart", Book.class);
}

@POST
public void addAttachments(List<Attachment> attachments)
   throws Exception {
    for (Attachment attachment : attachments) {
        Book book = attachment.getObject(Book.class);
    }
}
```

When reading large attachments, the "attachment-directory" and "attachment-memory-threshold" contextual properties can be used to control what folder the attachments exceeding a given threshold (in bytes) can be temporarily saved to.

3.3.2. Writing Attachments

It is possible to write attachments to the output stream, both on the client and server sides.

On the server side it is sufficient to update the @Produces value for a given method:

```
public class Resource {
   private List<Book> books;
   @Produces("multipart/mixed;type=text/xml")
   public List<Book> getBooksAsMultipart() {
      return booksList;
   }

   @Produces("multipart/mixed;type=text/xml")
   public Book getBookAsMultipart() {
      return booksList;
   }
}
```

Note that a 'type' parameter of the 'multipart/mixed' media type indicates that all parts in the multiparts response should have a Content-Type header set to 'text/xml' for both getBooksAsMultipart() and getBookAsMultipart() method responses. The getBooksAsMultipart() response will have 3 parts, the first part will have its Content-ID header set to "root.message@cxf.apache.org", the next parts will have '1' and '2' ids. The getBookAsMultipart() response will have a single part only with its Content-ID header set to "root.message@cxf.apache.org".

When returning mixed multiparts containing objects of different types, you can either return a Map with the media type string value to Object pairs or MultipartBody:

```
public class Resource {
   private List<Book> books;
   @Produces("multipart/mixed")
   public Map<String, Object> getBooks() {
      Map<String, Object> map = new LinkedHashMap<String, Object>();
      map.put("text/xml", new JaxbBook());
      map.put("application/json", new JSONBook());
      map.put("application/octet-stream", imageInputStream);
      return map;
   }

   @Produces("multipart/mixed")
   public MultipartBody getBooks2() {
      List<Attachment> atts = new LinkedList<Attachment>();
      atts.add(new Attachment("root", "application/json",
         new JSONBook()));
      atts.add(new Attachment("image", "application/octet-stream",
         getImageInputStream()));
      return new MultipartBody(atts, true);
   }

}
```

Similarly to the method returning a list in a previous code fragment, getBooks() will have the response serialized as multiparts, where the first part will have its Content-ID header set to "root.message@cxf.apache.org", the next parts will have ids like '1', '2', etc.

In getBooks2() one can control the content ids of individual parts.

You can also control the contentId and the media type of the root attachment by using a Multipart annotation:

```
public class Resource {
   @Produces("multipart/form-data")
   @Multipart(value = "root", type = "application/octet-stream")
   public File testGetImageFromForm() {
      return getClass().getResource("image.png").getFile();
   }
}
```

One can also have lists or maps of DataHandler, DataSource, Attachment, byte arrays or InputStreams handled as multiparts.

On the client side multiparts can be written the same way. For example:

```
WebClient client = WebClient.create("http://books");
client.type("multipart/mixed").accept("multipart/mixed");
List<Attachment> atts = new LinkedList<Attachment>();
atts.add(new Attachment("root", "application/json", new JSONBook()));
atts.add(new Attachment("image", "application/octet-stream",
   getImageInputStream()));
List<Attachment> atts = client.postAndGetCollection(atts,
   Attachment.class);
```

When using proxies, a Multipart annotation attached to a method parameter can also be used to set the root contentId and media type. Proxies do not support at the moment multiple method parameters annotated with Multipart (as opposed to the server side) but only a single multipart parameter:

```
public class Resource {
    @Produces("multipart/mixed")
```

```
    @Consumes("multipart/form-data")
    @Multipart(value = "root", type = "application/octet-stream")
    public File postGetFile(@Multipart(value = "root2",
       type = "application/octet-stream") File file) {}
}
```

A method-level Multipart annotation will affect the writing on the server side and the reading on the client side. A parameter-level Multipart annotation will affect writing on the client (proxy) side and reading on the server side. You don't have to use Multipart annotations.

3.3.3. Uploading files

At the moment the only way to upload a file is to use a MultipartBody, Attachment or File:

```
WebClient client = WebClient.create("http://books");
client.type("multipart/form-data");
ContentDisposition cd = new ContentDisposition(
   "attachment;filename=image.jpg");
Attachment att = new Attachment("root", imageInputStream, cd);
client.post(new MultipartBody(att));

// or just post the attachment if it's a single part request only
client.post(att);

// or just use a file
client.post(getClass().getResource("image.png").getFile());
```

Using File provides a simpler way as the runtime can figure out how to create a ContentDisposition from a File.

3.3.4. Forms and multiparts

The Forms in HTML documents recommendation suggests that multipart/form-data requests should mainly be used to upload files.

One way to deal with multipart/form-data submissions is to deal directly with a CXF JAXRS Attachment class and get a Content-Disposition header and/or the underlying input stream. It is also possible to have individual multipart/form-data parts read by registered JAX-RS MessageBodyReaders, something that is already possible to do for types like multipart/mixed or multipart/related. For example, this payload:

```
--bqJky99mlBWa-ZuqjC53mG6EzbmlxB
Content-Disposition: form-data; name="bookJson"
Content-Type: application/json; charset=US-ASCII
Content-Transfer-Encoding: 8bit
Content-ID: <jsonPart>

{"Book":{"name":"CXF in Action - 1","id":123}}

--bqJky99mlBWa-ZuqjC53mG6EzbmlxB
Content-Disposition: form-data; name="bookXML"
Content-Type: application/xml
Content-Transfer-Encoding: 8bit
```

```
Content-ID: <jaxbPart>
<Book><name>CXF in Action</name></Book>
--bqJky99mlBWa-ZuqjC53mG6EzbmlxB--
```

can be handled by the following method:

```
@POST
@Path("/books/jsonjaxbform")
@Consumes("multipart/form-data")
public Response addBookJaxbJsonForm(@Multipart("jsonPart") Book b1,
                                    @Multipart("bookXML") Book b2) {}
```

Note that once a request has more than two parts then one needs to start using @Mutipart, the values can refer to either ContentId header or to ContentDisposition/name. At the moment using @Multipart is preferred to using @FormParam unless a plain name/value submission is dealt with. The reason is that @Multipart can also specify an expected media type of the individual part and thus act similarly to a @Consume annotation.

When dealing with multiple parts one can avoid using @Multipart and just use List<Atachment>, List<Book>, etc.

Finally, multipart/form-data requests with multiple files (file uploads) can be supported. For example, this payload:

```
--bqJky99mlBWa-ZuqjC53mG6EzbmlxB
Content-Disposition: form-data; name="owner"
Content-Type: text/plain

Larry
--bqJky99mlBWa-ZuqjC53mG6EzbmlxB
Content-Disposition: form-data; name="files"
Content-Type: multipart/mixed; boundary=_Part_4_701508.1145579811786

--_Part_4_701508.1145579811786
Content-Disposition: form-data; name="book1"
Content-Type: application/json; charset=US-ASCII
Content-Transfer-Encoding: 8bit

{"Book":{"name":"CXF in Action - 1","id":123}}
--_Part_4_701508.1145579811786
Content-Disposition: form-data; name="book2"
Content-Type: application/json; charset=US-ASCII
Content-Transfer-Encoding: 8bit

{"Book":{"name":"CXF in Action - 2","id":124}}
--_Part_4_701508.1145579811786--
--bqJky99mlBWa-ZuqjC53mG6EzbmlxB--
```

can be handled by the following method:

```
@POST
@Path("/books/filesform")
@Produces("text/xml")
@Consumes("multipart/form-data")
public Response addBookFilesForm(@Multipart("owner") String name,
    @Multipart("files") List<Book> books) {}
```

If you need to know the names of the individual file parts embedded in a "files" outer part (such as "book1" and "book2"), then please use List<Attachment> instead. It is currently not possible to use a Multipart annotation to

refer to such inner parts but you can easily get the names from the individual Attachment instances representing these inner parts.

Note that it is only the last request which has been structured according to the recommendation on how to upload multiple files but it is more complex than the other simpler requests linked to in this section.

Please note that using JAX-RS FormParams is recommended for dealing with plain application/www-url-encoded submissions consisting of name/value pairs only.

3.3.5. XOP support

CXF JAXRS clients and endpoints can support XML-binary Optimized Packaging (XOP). What it means at the practical level is that a JAXB bean containing binary data is serialized using a multipart packaging, with the root part containing non-binary data only but also linking to co-located parts containing the actual binary payloads. Next it is deserialized into a JAXB bean on the server side.

If you would like to experiment with XOP then you need to set an "mtom-enabled" property on CXF jaxrs endpoints and clients.

3.4. Configuration

CXF JAX-RS endpoints and clients can be configured declaratively (using Spring or web.xml only) and programmatically.

3.4.1. Configuration of Endpoints

Providing a custom javax.ws.rs.core.Application implementation is the only portable way to register root resource and provider classes and indicate what lifecycle model the individual resources follow. For example:

```
package server;

import java.util.HashSet;
import java.util.Set;
import javax.ws.rs.core.Application;

public class BookApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(BookStore.class);
        return classes;
    }

    @Override
    public Set<Object> getSingletons() {
        Set<Object> classes = new HashSet<Object>();
        classes.add(new SearchService());
        classes.add(new BookExceptionMapper());
```

```
        return classes;
    }
}
```

The BookApplication indicates to the runtime that BookStore root resource has the per-request lifecycle. The SearchService root resource and BookExceptionMapper provider are singletons. In CXF one can register JAX-RS Applications in web.xml using a CXFNonSpringJaxrsServlet:

```
           <servlet>
             <servlet-name>CXFServlet</servlet-name>
             <display-name>CXF Servlet</display-name>
             <servlet-class>
              org.apache.cxf.jaxrs.servlet.CXFNonSpringJaxrsServlet
             </servlet-class>
             <init-param>
               <param-name>javax.ws.rs.Application</param-name>
               <param-value>
                 server.BookApplication
               </param-value>
             </init-param>
          </servlet>
```

Spring users can configure the JAX-RS endpoints using one or more jaxrs:server declarations:

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

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource=
      "classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

  <bean class="org.apache.cxf.systest.jaxrs.BookStore" id="serviceBean"/>
  <bean class="org.apache.cxf.systest.jaxrs.provider.JAXBElementProvider"
     id="jaxbProvider">
    <!-- customize the default JAXBElementProvider
       by setting some of its properties -->
  </bean>

  <jaxrs:server id="bookservice" address="/bookstore">
    <jaxrs:serviceBeans>
      <ref bean="serviceBean" />
    </jaxrs:serviceBeans>
    <jaxrs:providers>
       <ref bean="jaxbProvider"/>
    </jaxrs:providers>

    <jaxrs:features>
       <!-- Register CXF features such as FastInfoset or Logging -->
    </jaxrs:features>
```

```
    <jaxrs:inInterceptors>
       <!-- Register CXF in interceptors, example, reuse common in
            interceptors between JAX-WS and JAX-RS endpoints -->
    </jaxrs:inInterceptors>
    <jaxrs:outInterceptors>
       <!-- Register CXF out interceptors, example, reuse common out
            interceptors between JAX-WS and JAX-RS endpoints -->
    </jaxrs:outInterceptors>
  </jaxrs:server>
</beans>
```

A single JAX-RS endpoint is registered with a jaxrs:server declaration. This declaration may reference multiple root resource beans with jaxrs:serviceBeans and multiple providers using jaxrs:providers. Note a jaxrs:server/ @address attribute. It allows for registering multiple jaxrs:server endpoints with all of them referencing the same service beans but using differently configured JAX-RS providers.

The jaxrs:server endpoints can register CXF features, in and out CXF interceptors.

3.4.2. Configuration of Clients

CXF JAX-RS clients can be configured programmatically or from Spring.

Configuring the clients from Spring often implies that the client instances are injected into JAX-RS or JAX-WS endpoints so that the incoming request can be further delegated to the RESTful service. Both proxies and WebClient instances can be configured from Spring:

```
  <jaxrs:client id="restClient"
         address="http://localhost:9000/test/services/rest"
         serviceClass="server.BookStoreJaxrsJaxws">
         <jaxrs:headers>
             <entry key="Accept" value="text/xml"/>
         </jaxrs:headers>
  </jaxrs:client>

<bean id="webClient" class="org.apache.cxf.jaxrs.client.WebClient"
    factory-method="create">
        <constructor-arg type="java.lang.String"
value="http://localhost:9000/books/" />
</bean>
```

3.5. Tutorials

3.5.1. Creating a Basic JAX-RS endpoint

When starting with the JAX-RS development, trying to visualize or model the way the RESTful resources will be accessed can be helpful. One simple approach is to draw a table against each resource and list the HTTP verbs,

formats and URIs this resource will support. The other approach is to try to imagine what part of the application URI space will need to be supported by a dedicated handler.

These handlers will be mapped to JAX-RS root resource classes. For example, assuming "/books" and "/book" URI segments need to be supported, then one can imagine the developer starting to work on the following two root resource classes, alongside with a couple of simple beans:

```
/**
 * The Book JAXB bean.
 **/
@XmlRootElement(name = "book", namespace = "http://books")
public class Book {
   private String name;
   private String author;
   private Date publicationDate;
   private List<String> reviews;
   // setters and getters are omitted
}

/**
 * The Collection of Book instances.
 **/
@XmlRootElement(name = "books", namespace = "http://books")
public class Books {
  // XmlJavaTypeAdapter is available
  private Map<Long, Book> books =
      Collections.synchronizedMap(new HashMap<Long, Book>());

  public void addBook(Long id, Book b) {
      books.put(id, b);
  }

  public Book getBook(Long id) {
      return books.get(id);
  }

  public void deleteBook(Long id) {
      books.remove(id);
  }

  public void addBookReview(Long id, String review) {
      getBook(id).addReview(review);
  }
}

/**
 * BookStore root resource class is responsible for handling
 * URIs ending with '/books', '/books/{id}', etc. This resource
 * will let users get the list of all books and add new books
 **/
@Path("/books")
public class BooksStore {
   private static AtomicLong ID = new AtomicLong();
   private Books books;

   // Thread-safe UriInfo instance providing the
```

```
    // extended information about the current URI
    @Context
    private UriInfo uriInfo;

    /**
     * Injects the Books storage
     **/
    public void setBooks(Books books) {
        this.books = books;
    }

    /**
     * Returns the list of all the books
     **/
    @GET
    @Produces("application/xml")
    public Books getAllBooks() {
        retun books;
    }

    /**
     * Adds a new Book to the internal storage and returns
     * an HTTP Location header pointing to a new Book resource.
     **/
    @POST
    @Consumes("application/xml")
    public Response addBook(Book book) {
        // New Book ID
        Long id = ID.incrementAndGet();
        books.add(id, book);

        // Get the base URI of the application and wrap it into a builder.
        // UriBuilder makes it easy to compose new URIs.
        UriBuilder builder = uriInfo.getBaseUriBuilder();

        // Build a new book resource URI
        // with say a '/book/1' segment added to the base URI
        builder.path("/book/" +  id);
        URI newBookResourceURI = builder.build();

        // Return 201 and the Location header
        retun Response.created().location(uri).build();
    }
}

/**
 * BookStore root resource class is responsible for handling
 * URIs ending with '/book', '/book/{id}', etc. This resource
 * will let users get, update and delete individual books.
 **/
@Path("/book")
public class BookHandler {
     private Books books;
     /**
      * Injects the Books storage
      **/
     public void setBooks(Books books) {
        this.books = books;
```

```
    }

    @GET
    @Produces("application/xml")
    @Path("{id}")
    public Book getBook(@PathParam("id") Long id) {
        return books.getBook(id);
    }

    @PUT
    @Consumes("text/plain")
    @Path("{id}")
    public void setBookReview(@PathParam("id") Long id, String review) {
        books.addBookReview(review);
    }

    @DELETE
    @Path("{id}")
    public void deleteBook(@PathParam("id") Long id) {
        books.deleteBook(id);
    }
}
```

The developer has prototyped two root resource classes, BooksStore and BookHandler. Next the configuration for the new JAX-RS endpoint has been added (see below for the example), the store.war has been built and deployed to a servlet container listening on localhost:8080. Given that the name of the war, 'store' in this case, contributes to the URI path, the base URI of the Store application is 'http://localhost:8080/store'.

The BookStore will handle HTTP GET requests with the URIs such as 'http://localhost:8080/store/books' and return the list of Books in the XML format. It will also accept POST requests with new Books being submitted in the XML format to 'http://localhost:8080/store/books'.

The BookStore.getAllBooks() method implementation is simple, while BookStore.addBook(Book) is a bit more involved, but it simply follows a basic pattern to do with adding new resources. Particularly, POST usually adds new resources and the typical response is to return a 201 status with the Location header pointing to a new resource URI. For example, if the new Book id is "1" then given that the base URI is 'http://localhost:8080/store/', the unique resource URI of the new Book resource will be 'http://localhost:8080/store/book/1'.

The client code or browser script which was used to add the new Book can choose to follow the 'http://localhost:8080/store/book/1' using GET. In this case another root resource, BookHandler will handle GET, as well as PUT and DELETE requests, all targeted at the individual Book resources.

Note that instead of introducing a dedicated BookHandler root resource supporting the URIs such as '/book/1', the developer couild have opted for supporting 'books/1' URIs instead, and thus all the BookHandler methods could have been implemented as part of the BooksStore class. For example:

```
@Path("/books")
public class BooksStore {
    ...

    @GET
    @Produces("application/xml")
    public Books getAllBooks() {...}

    @POST
    @Consumes("application/xml")
```

```
    public Response addBook(Book book) {...}

    @GET
    @Produces("application/xml")
    @Path("{id}")
    public Book getBook(@PathParam("id") Long id) {...}

    @PUT
    @Consumes("text/plain")
    @Path("{id}")
    public void setBookReview(@PathParam("id") Long id, String review)
       {...}

    @DELETE
    @Path("{id}")
    public void deleteBook(@PathParam("id") Long id) {...}
}
```

Many options are available and JAX-RS makes it easy for developers to structure their Java web services applications as needed

And here is how the JAX-RS endpoint can be configured:

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

  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource=
      "classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

  <bean class="org.books.BooksStore" id="storeBean"/>
  <bean class="org.books.BookHandler" id="bookBean"/>

  <jaxrs:server id="bookservice" address="/">
    <jaxrs:serviceBeans>
      <ref bean="storeBean" />
      <ref bean="bookBean" />
    </jaxrs:serviceBeans>
  </jaxrs:server>
</beans>
```

Please also see a jaxrs_intro demo in the TSF Examples distribution.

Chapter 4. JAX-RS and OAuth2

4.1. Introduction to OAuth2

CXF 2.6.0 provides an initial implementation of OAuth 2.0. See also the CXF Website for information about OAuth 1.0. Authorization Code, Implicit and Client Credentials grants are currently supported with other grant handlers to be added later. Custom grant handlers can also be registered.

OAuth2 is a new protocol which offers a complex yet elegant solution toward helping end users (resource owners) authorize third-party providers to access their resources. The OAuth2 flow is closely related to the original OAuth 1.0 3-leg flow is called Authorization Code and involves 3 parties: the end user, the third party service (client) and the resource server which is protected by OAuth2 filters. Typically a client offers a service feature that an end user requests and which requires the former to access one or more protected resources on behalf of this user which are located at the resource server. For example, the client may need to access the end user's photos in order to print them and post to the user or read and possibly update a user's calendar in order to make a booking.

In order to make it happen, the third-party service application/client needs to register itself with the OAuth2 server. This happens out-of-band and after the registration the client gets back a client key and secret pair. Typically the client is expected to provide the name and description of the application, the application logo URI, one or more redirect URIs, and other information that may help the OAuth2 authorization server to identify this client to the end user at the authorization time. From then on, the authorization code flow works like this:

1. End User requests the third-party service using a browser.
2. The client redirects the end user to OAuth2 Authorization Service, adding its client id, the state, redirect URI and the optional scope to the target URI. The state parameter represents the current end user's request, redirect URI - where the authorization code is expected to be returned to, and the scope is the list of opaque permissions that the client needs in order to access the protected resources.
3. Authorization Service will retrieve the information about the client using its client id, build an HTML form and return it to the end user. The form will ask the user if a given third-party application can be allowed to access some resources on behalf of this user.

4. If the user approves it then Authorization Service will generate an authorization code and redirect the user back to the redirect uri provided by the client, also adding a state parameter to the redirect URI.

5. The client requests an access token from OAuth2 Access Token Service by providing an authorization code grant.

6. After getting an access token token, the service finally proceeds with accessing the current user's resources and completes the user's request.

As you can see the flow can be complex yet it is very effective. A number of issues may need to be taken care along the way such as managing expired tokens, making sure that the OAuth2 security layer is functioning properly and is not interfering with the end user itself trying to access its own resources, etc. Please check the specification and the Wikipedia article as well as other resources available on the WEB for more information you may need to know about OAuth2.

CXF JAX-RS gives the best effort to making this process as simple as possible and requiring only a minimum effort on behalf of OAuth2 server developers. It also offers the utility code for greatly simplifying the way the third-party application can interact with the OAuth2 service endpoints.

Maven dependency needed for CXF's implementation of OAuth2:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-security-oauth2</artifactId>
  <version>2.6.0</version>
</dependency>
```

4.2. Developing OAuth2 Servers

The OAuth2 server is the core piece of the complete OAuth2-based solution. Typically it contains 2 services for:

1. Authorizing request tokens by asking the end users to let clients access some of their resources and returning the grants back to the client (Authorization Service)

2. Exchanging the token grants for access tokens (Access Token Service)

CXF offers several JAX-RS service implementations that can be used to create the OAuth2 servers fast: AuthorizationCodeGrantService and ImplicitGrantService for managing the redirection-based flows, as well as AccessTokenService for exchanging the grants for new tokens. Note that some grants that do not require the redirection-based support, such as SAML2 one, etc, may only require an Access Token Service be operational.

All of these services rely on the custom OAuthDataProvider which persists the access tokens and converts the opaque scope values to the information that can be presented to the users. Additionally, AuthorizationCodeDataProvider is an OAuthDataProvider which can keep temporary information about the authorization code grants which needs to be removed after the tokens are requested in exchange.

Writing your own OAuthDataProvider implementation is what is needed to get the OAuth2 server up and running. In many cases all you need to do is to persist or remove the Authorization Code Grant data, use one of the available utility classes to create a new access token and also persist it or remove the expired one, and finally convert the optional opaque scope values (if any are supported) to a more view-able information.

4.2.1. Authorization Service

The main responsibility of OAuth2 Authorization Service is to present an end user with a form asking the user to allow or deny the client accessing some of the user resources. CXF offers AuthorizationCodeGrantService and

ImplicitGrantService for accepting the redirection requests, challenging the end users with the authorization forms, handling the end user decisions and returning the results back to the clients.

One of the differences between the AuthorizationCode and Implicit flows is that in the latter case the grant is the actual access token which is returned as the URI fragment value. The way the end user is asked to authorize the client request is similar between the two flows. In this section we will assume that the Authorization Code flow is being exercised.

A third-party client redirects the current user to AuthorizationCodeGrantService, for example, here is how a redirection may happen:

```
Response-Code: 303
Headers: {Location=[http://localhost:8080/services/social/authorize?
   client_id=123456789&scope=updateCalendar-7&response_type=code
   &redirect_uri=http%3A//localhost%3A8080/services/reservations/reserve
   /complete&state=1],Date=[Thu, 12 Apr 2012 12:26:21 GMT],
   Content-Length=[0]}
```

The client application asks the current user (the browser) to go to a new address provided by the Location header and the follow-up request to AuthorizationCodeGrantService will look like this:

```
Address: http://localhost:8080/services/social/authorize?client_id=
   123456789&scope=updateCalendar-7&response_type=code&redirect_uri=
   http%3A//localhost%3A8080/services/reservations/reserve/complete&state=1
Http-Method: GET
Headers: {
Accept=[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8],
Authorization=[Basic YmFycnlAc29jaWFsLmNvbToxMjM0],
Cookie=[JSESSIONID=suj2wyl54c4g],
Referer=[http://localhost:8080/services/forms/reservation.jsp]
...
}
```

Note that the end user needs to authenticate. The Request URI includes the client_id, custom scope value, response_type set to 'code', the current request state and the redirect uri. Note the scope is optional - the Authorization Service will usually allocate a default scope; however even if the client does include an additional custom scope the end user may still not approve it. The redirect uri is also optional, assuming one or more ones redirect URIs have been provided at the client registration time.

AuthorizationCodeGrantService will report a warning is no secure HTTPS transport is used:

```
12-Apr-2012 13:26:21
   org.apache.cxf.rs.security.oauth2.services.AbstractOAuthService
   checkTransportSecurity
WARNING: Unsecure HTTP, Transport Layer Security is recommended
```

It can also be configured to reject the requests over un-secure HTTP transport.

AuthorizationCodeGrantService will retrieve the information about the client application to populate an instance of OAuthAuthorizationData bean and return it. OAuthAuthorizationData contains application name and URI properties, optional list of Permissions and other properties which can be either presented to the user or kept in the hidden form fields in order to uniquely identify the actual authorization request when the end user returns the decision.

One important OAuthAuthorizationData property is "authenticityToken". It is used for validating that the current session has not been hijacked - AuthorizationCodeGrantService generates a random key, stores it in a Servlet

HTTPSession instance and expects the returned authenticityToken value to match it - this is a recommended approach and it also implies that the authenticityToken value is hidden from a user, for example, it's kept in a 'hidden' form field. The other properties which are meant to be hidden are clientId, state, redirectUri, proposedScope. The helper "replyTo" property is an absolute URI identifying the AuthorizationCodeGrantService handler processing the user decision and can be used by view handlers when building the forms or by other OAuthAuthorizationData handlers.

So the populated OAuthAuthorizationData is finally returned. Note that it's a JAXB XMLRootElement-annotated bean and can be processed by registered JAXB or JSON providers given that AuthorizationCodeGrantService supports producing "application/xml" and "application/json" (See the OAuth Without Browser section below for more). But in this case we have the end user working with a browser so an HTML form is what is really expected back.

AuthorizationCodeGrantService supports producing "text/html" and simply relies on a registered RequestDispatcherProvider to set the OAuthAuthorizationData bean as an HttpServletRequest attribute and redirect the response to a view handler (can be JSP or some other servlet) to actually build the form and return it to the user. Alternatively, registering XSLTJaxbProvider would also be a good option for creating HTML views.

Assuming RequestDispatcherProvider is used, the following example log shows the initial response from AuthorizationCodeGrantService:

```
12-Apr-2012 13:26:21 org.apache.cxf.jaxrs.provider.
    RequestDispatcherProvider logRedirection
INFO: Setting an instance of "org.apache.cxf.rs.security.oauth2.common.
    OAuthAuthorizationData" as HttpServletRequest attribute "data" and
    redirecting the response to "/forms/oauthAuthorize.jsp".
```

Note that a "/forms/oauthAuthorize.jsp" view handler will create an HTML view - this is a custom JSP handler and whatever HTML view is required can be created there, using the OAuthAuthorizationData bean for building the view. Most likely you will want to present a form asking the user to allow or deny the client accessing some of this user's resources. If OAuthAuthorizationData has a list of Permissions set then adding the information about the permissions is needed.

Next the user makes a decision and selects a button allowing or denying the client accessing the resources. The form data are submitted to AuthorizationCodeGrantService:

```
Address: http://localhost:8080/services/social/authorize/decision
Encoding: ISO-8859-1
Http-Method: POST
Content-Type: application/x-www-form-urlencoded
Headers: {
Authorization=[Basic YmFycnlAc29jaWFsLmNvbToxMjM0],
Content-Type=[application/x-www-form-urlencoded],
...
}
--------------------------------------
12-Apr-2012 15:36:29 org.apache.cxf.jaxrs.utils.FormUtils
    logRequestParametersIfNeeded
INFO: updateCalendar-7_status=allow&readCalendar_status=allow
&scope=updateCalendar-7+readCalendar
&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Fservices%2F
    reservations%2Freserve%2Fcomplete
&session_authenticity_token=4f0005d9-565f-4309-8ffb-c13c72139ebe
    &oauthDecision=allow
&state=1&client_id=123456789
```

AuthorizationCodeGrantService will use a session_authenticity_token to validate that the session is valid and will process the user decision next.

If the decision is "allow" then it will check the status of the individual scope values. It relies on the "scopename_status" convention, if the form has offered the user a chance to selectively enable individual scopes then name/value pairs such as "updateCalendar-7_status=allow" are submitted. If none of such pairs is coming back then it means the user has approved all the default and additional (if any) scopes. Next it will ask OAuthDataProvider to generate an authorization code grant and return it alongside with the state if any by redirecting the current user back to the redirect URI:

```
Response-Code: 303
Headers: {
 Location=[http://localhost:8080/services/reservations/reserve/complete
 ?state=1&code=5c993144b910bccd5977131f7d2629ab],
 Date=[Thu, 12 Apr 2012 14:36:29 GMT],
 Content-Length=[0]}
```

Which leads to a browser redirecting the user:

```
Address: http://localhost:8080/services/reservations/reserve/complete?
state=1&code=5c993144b910bccd5977131f7d2629ab
Http-Method: GET
Headers: {
Accept=[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8],
Authorization=[Basic YmFycnlAcmVzdGF1cmFudC5jb206NTY3OA==],
Cookie=[JSESSIONID=1c289vha0cxfe],
}
```

If a user decision was set to "deny" then the error will be returned to the client. Assuming the decision was "allow", the client has now received back the authorization code grant and is ready to exchange it for a new access token.

4.2.2. AccessTokenService

The role of AccessTokenService is to exchange a token grant for a new access token which will be used by the client to access the end user's resources. Here is an example request log:

```
Address: http://localhost:8080/services/oauth/token
Http-Method: POST

Headers: {
Accept=[application/json],
Authorization=[Basic MTIzNDU2Nzg5Ojk4NzY1NDMyMQ==],
Content-Type=[application/x-www-form-urlencoded]
}
Payload:

grant_type=authorization_code&code=5c993144b910bccd5977131f7d2629ab
   &redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Fservices%2Freservations
   %2Freserve%2Fcomplete
```

This request contains a client_id and client_secret (Authorization header), the grant_type, the grant value (code) plus the redirect URI the authorization grant was returned to which is needed for the additional validation. Note that the alternative client authentication methods are also possible, in this case the token service will expect a mapping between the client credentials and the client_id representing the client registration available.

After validating the request, the service will find a matching AccessTokenGrantHandler and request to create a ServerAccessToken which is a server-side representation of the access token. The grant handlers, such as

AuthorizationCodeGrantHandler may delegate the creation of the actual access token to data providers, which may use the available utility classes such as BearerAccessToken shipped with CXF or depend on other 3rd party libraries to create the tokens.

The data providers are also do not strictly required to persist the data such as access tokens, instead the token key may an encrypted bag capturing all the relevant information.

Now that the token has been created, it is mapped by the service to a client representation and is returned back as a JSON payload:

```
Response-Code: 200
Content-Type: application/json
Headers: {
 Cache-Control=[no-store],
 Pragma=[no-cache],
 Date=[Thu, 12 Apr 2012 14:36:29 GMT]
}

Payload:

{"access_token":"5b5c8e677413277c4bb8b740d522b378", "token_type":"bearer"}
```

The client will use this access token to access the current user's resources in order to complete the original user's request, for example, the request to access a user's calendar may look like this:

```
Address: http://localhost:8080/services/thirdPartyAccess/calendar
Http-Method: GET
Headers:
{
  Authorization=[Bearer 5b5c8e677413277c4bb8b740d522b378],
  Accept=[application/xml]
}
```

Note that the access token key is passed as the Bearer scheme value. Other token types such as MAC ones, etc, can be represented differently.

4.2.3. Writing OAuthDataProvider

Using CXF OAuth service implementations will help a lot with setting up an OAuth server. As you can see from the above sections, these services rely on a custom OAuthDataProvider implementation.

The main task of OAuthDataProvider is to persist and generate access tokens. Additionally, as noted above, AuthorizationCodeDataProvider needs to persist and remove the code grant registrations. The way it's done is really application-specific. Consider starting with a basic memory based implementation and then move on to keeping the data in some DB.

Note that OAuthDataProvider supports retrieving Client instances but it has no methods for creating or removing Clients. The reason for it is that the process of registering third-party clients is very specific to a particular OAuth2 application, so CXF does not offer a registration support service and hence OAuthDataProvider has no Client create/update methods. You will likely need to do something like this:

```
public class CustomOAuthProvider implements OAuthDataProvider {
   public Client registerClient(String applicationName,
       String applicationURI, ...) {}
   public void removeClient(String cliendId) {}
```

```
    // etc
    // OAuthDataProvider methods
}
```

CustomOAuthProvider will also remove all tokens associated with a given Client in removeClient(String cliendId).

Finally OAuthDataProvider may need to convert opaque scope values such as "readCalendar" into a list of OAuthPermissions. AuthorizationCodeGrantService and OAuth2 security filters will depend on it (assuming scopes are used in the first place). In the former case AuthorizationCodeGrantService will use this list to populate OAuthAuthorizationData - the reason this bean only sees Permissions is that some of the properties OAuthPermission keeps are of no interest to OAuthAuthorizationData handlers.

4.2.4. OAuth Server JAX-RS endpoints

With CXF offering OAuth service implementations and a custom OAuthDataProvider provider in place, it is time to deploy the OAuth2 server. Most likely, you'd want to deploy AccessTokenService as an independent JAX-RS endpoint, for example:

```
<!-- implements OAuthDataProvider -->
<bean id="oauthProvider" class="oauth.manager.OAuthManager"/>

<bean id="accessTokenService"
  class="org.apache.cxf.rs.security.oauth2.services.AccessTokenService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<jaxrs:server id="oauthServer" address="/oauth">
   <jaxrs:serviceBeans>
      <ref bean="accessTokenService"/>
  </jaxrs:serviceBeans>
</jaxrs:server>
```

AccessTokenService listens on a relative "/token" path. Given that jaxrs:server/@adress is "/oauth" and assuming a context name is "/services", the absolute address of AccessTokenService would be something like "http://localhost:8080/services/oauth/token".

AuthorizationCodeGrantService is better to put where the main application endpoint is. It can be put alongside AccessTokenService - but the problem is that the end user is expected to authenticate itself with the resource server after it has been redirected by a third-party client to AuthorizationCodeGrantService. That would make it more complex for the OAuth server endpoint to manage both OAuth (third-party client) and the regular user authentication - that can be done, see more on it below in the Design considerations section, but the simpler option is to simply get AuthorizationCodeGrantService under the control of the security filter enforcing the end user authentication:

```
<bean id="authorizationService"
  class="org.apache.cxf.rs.security.oauth2.services.  //
         AuthorizationCodeGrantService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<bean id="myApp" class="org.myapp.MyApp">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<jaxrs:server id="oauthServer" address="/myapp">
```

```
    <jaxrs:serviceBeans>
       <ref bean="myApp"/>
       <ref bean="authorizationService"/>
   </jaxrs:serviceBeans>
</jaxrs:server>
```

AuthorizationCodeGrantService listens on a relative "/authorize" path so in this case its absolute address will be something like "http://localhost:8080/services/myapp/authorize". This address and that of AccessTokenService will be used by third-party clients.

4.3. Protecting resources with OAuth2 filters

OAuthRequestFilter request handler can be used to protect the resource server when processing the requests from the third-party clients. Add it as a jaxrs:provider to the endpoint which deals with the clients requesting the resources. When checking a request like this:

```
Address: http://localhost:8080/services/thirdPartyAccess/calendar
Http-Method: GET
Headers:
{
  Authorization=[Bearer 5b5c8e677413277c4bb8b740d522b378],
  Accept=[application/xml]
}
```

the filter will do the following:

1. Retrieve a ServerAccessToken by delegating to a matching registered AccessTokenValidator. AccessTokenValidator is expected to check the validity of the incoming token parameters and possibly delegate to OAuthDataProvider to find the token representation - this is what the filter will default to if no matching AccessTokenValidator is found and the Authorization scheme is 'Bearer'.

2. Check the token has not expired

3. AccessToken may have a list of OAuthPermissions. For every permission it will:

   - If it has a uri property set then the current request URI will be checked against it

   - If it has an httpVerb property set then the current HTTP verb will be checked against it

4. Finally, it will create a CXF SecurityContext using this list of OAuthPermissions, the UserSubject representing the client or the end user who authorized the grant used to obtain this token.

This SecurityContext will not necessarily be important for some of OAuth2 applications. Most of the security checks will be done by OAuth2 filters and security filters protecting the main application path the end users themselves use. Only if you would like to share the same JAX-RS resource code and access URIs between end users and clients then it can become handy. More on it below.

4.4. How to get the user login name

When one writes a custom server application which needs to participate in OAuth2 flows, the major question which needs to be addressed is how one can access a user login name that was used during the end-user authorizing the third-party client. This username will help to uniquely identify the resources that the 3rd party client is now attempting to access. The following code shows one way of how this can be done:

```
@Path("/userResource")
public class ThirdPartyAccessService {

    @Context
    private MessageContext mc;

    @GET
    public UserResource getUserResource() {
        OAuthContext oauth = mc.getContent(OAuthContext.class);
        if (oauth == null || oauth.getSubject() == null ||
           oauth.getSubject().getLogin() == null) {
    throw new WebApplicationException(403);
 }
 String userName = oauth.getSubject().getLogin();
 return findUserResource(userName)
    }

    private UserResource findUserResource(String userName) {
        // find and return UserResource
    }
}
```

The above shows a fragment of the JAX-RS service managing the access to user resources from authorized 3rd-party clients (see the Design Considerations section for more information).

The injected MessageContext provides an access to OAuthContext which has been set by OAuth2 filters described in the previous section. OAuthContext will act as a container of the information which can be useful to the custom application code which do not need to deal with the OAuth2 internals.

4.5. Client-side support

When developing a third party application which needs to participate in OAuth2 flows one has to write the code that will redirect users to OAuth2 AuthorizationCodeGrantService, interact with AccessTokenService in order to exchange code grants for access tokens as well as correctly build Authorization OAuth2 headers when accessing the end users' resources. JAX-RS makes it straightforward to support the redirection, while OAuthClientUtils class makes it possible to encapsulate most of the complexity away from the client application code.

For example, the following custom code can be used by the third-party application:

```
public class OAuthClientManager {

   private WebClient accessTokenService;
   private String authorizationServiceURI;
   private Consumer consumer;

   // inject properties, register the client application...

   public URI getAuthorizationServiceURI(ReservationRequest request,
          URI redirectUri,
          /* state */String reservationRequestKey) {
      String scope = OAuthConstants.UPDATE_CALENDAR_SCOPE +
          request.getHour();
      return OAuthClientUtils.getAuthorizationURI(authorizationServiceURI,
          consumer.getKey(),
```

```
            redirectUri.toString(),
            reservationRequestKey,
            scope);
    }

    public ClientAccessToken getAccessToken(AuthorizationCodeGrant
       codeGrant) {
       try {
          return OAuthClientUtils.getAccessToken(accessTokenService,
          consumer, codeGrant);
       } catch (OAuthServiceException ex) {
          return null;
       }
    }

    public String createAuthorizationHeader(ClientAccessToken token) {
       return OAuthClientUtils.createAuthorizationHeader(consumer,
          token);
    }
}
```

The reason such a simple wrapper can be introduced is to minimize the exposure to OAuth2 of the main application code to the bare minimum, this is why in this example OAuthServiceExceptions are caught, presumably logged and null values are returned which will indicate to the main code that the request failed. Obviously, OAuthClientUtils can be used directly as well.

4.6. OAuth2 without Explicit Authorization

Client Credentials is one of OAuth2 grants that does not require the explicit authorization and is currently supported by CXF.

4.7. OAuth2 without a Browser

When an end user is accessing the 3rd party application and is authorizing it later on, it's usually expected that the user is relying on a browser. However, supporting other types of end users is easy enough. Writing the client code that processes the redirection requests from the 3rd party application and AuthorizationRequestService is simple with JAX-RS and additionally CXF can be configured to do auto-redirects on the client side.

Also note that AuthorizationRequestService can return XML or JSON OAuthAuthorizationData representations. That makes it easy for a client code to get OAuthAuthorizationData and offer a pop-up window or get the input from the command-line. Authorizing the third-party application might even be automated in this case - which can lead to a complete 3-leg OAuth flow implemented without a human user being involved.

4.8. Controlling the Access to Resource Server

One of the most important issues one needs to resolve is how to partition a URI space of the resource server application. We have two different parties trying to access it, the end users which access the resource server to

get to the resources which they own and 3rd party clients which have been authorized by the end users to access some of their resources. In the former case the way the authentication is managed is completely up to the resource server application: basic authentication, two-way TLS, OpenId (more on it below), you name it.

In the latter case an OAuth filter must enforce that the 3rd party client has been registered using the provided client key and that it has a valid access token which represents the end user's approval. It's kind of the authentication and the authorization check at the same time.

Letting both parties access the resource server via the same URI(s) complicates the life for the security filters but all the parties are only aware of the single resource server URI which all of them will use.

Providing different access points to end users and clients may significantly simplify the authentication process - the possible downside is that multiple access points need to be mantained by the resource server. Both options are discussed next.

4.8.1. Sharing the same access path between end users and clients

The first problem which needs to be addressed is how to distinguish end users from third-party clients and get both parties authenticated as required. Perhaps the simplest option is to extend a CXF OAuth2 filter (JAX-RS or servlet one), check Authorization header, if it is OAuth2 then delegate to the superclass, alternatively - proceed with authenticating the end users:

```
public class SecurityFilter
   extends org.apache.cxf.rs.security.oauth2.filters.OAuthRequestFilter {
   @Context
   private HttpHeaders headers;

   public Response handleRequest(ClassResourceInfo cri, Message message) {
       String header = headers.getRequestHeaders().getFirst("Authorization");
       if (header.startsWith("Bearer ")) {
           return super.handleRequest(cri, message);
       } else {
           // authenticate the end user
       }
   }

}
```

The next issue is how to enforce that the end users can only access the resources they've been authorized to access. For example, consider the following JAX-RS resource class:

```
@Path("calendar")
public class CalendarResource {

   @GET
   @Path("{id}")
   public Calendar getPublicCalendar(@PathParam("id") long id) {
       // return the calendar for a user identified by 'id'
   }

   @GET
   @Path("{id}/private")
   public Calendar getPrivateCalendar(@PathParam("id") long id) {
```

```
        // return the calendar for a user identified by 'id'
    }

    @PUT
    @Path("{id}")
    public void updateCalendar(@PathParam("id") long id, Calendar c) {
        // update the calendar for a user identified by 'id'
    }
}
```

Let's assume that the 3rd party client has been allowed to read the public user Calendars at "/calendar/\{id}" only, how to make sure that the client won't try to:

1. update the calendar available at the same path
2. read the private Calendars available at "/calendar/\{id}/private"

As noted above, OAuthPermission has an optional URIs property. Thus one way to solve the problem with the private calendar is to add, say, a uri "/calendar/\{id}" or "/calendar/1" (etc) property to OAuthPermission (representing a scope like "readCalendar") and the OAuth filter will make sure no subresources beyond "/calendar/\{id}" can be accessed. Note, adding a "*" at the end of a given URI property, for example, "/a*" will let the client access "/a", "/a/b", etc.

Solving the problem with preventing the update can be easily solved by adding an httpVerb property to a given OAuthPermission.

One more option is to rely on the role-based access control and have @RolesAllowed allocated such that only users in roles like "client" or "enduser" can invoke the getCalendar() method and let only those in the "enduser" role access getPrivateCalendar() and updateCalendar(). OAuthPermission can help here too as described in the section on using OAuth fiters.

4.8.2. Providing different access points to end users and clients

Rather than letting both the end users and 3rd party clients use the same URI such as "http://myapp.com/service/calendars/\{id}", one may want to introduce two URIs, one for end users and one for third-party clients, for example, "http://myapp.com/service/calendars/\{id}" - for endusers, "http://myapp.com/partners/calendars/\{id}" - for the 3rd party clients and deploy 2 jaxrs endpoints, where one is protected by the security filter checking the end users, and the one - by OAuth filters.

Additionally the endpoint managing the 3rd party clients will deploy a resource which will offer a resticted URI space support. For example, if the application will only allow 3rd party clients to read calendars then this resource will only have a method supporting @GET and "/calendar/\{id}".

Chapter 5. Combining JAX-WS and JAX-RS

Starting with the JAX-RS development is not necessarily all or nothing decision. Some users may want to start developing new web services using JAX-RS while some will prefer continue building on their SOAP WS experience. Sometimes migrating the advanced SOAP services is not an option at all. CXF provides the production-quality environment for SOAP and RESTful web services be developed and combined if needed.

The real world Java-based web services projects will often combine all sort of web services, some of them written using JAX-WS and some JAX-RS. One obvious option for combining JAX-WS and JAX-RS services in CXF is to register multiple JAX-WS and JAX-RS service endpoints all referencing the same service bean. The SOAP WS and REST approaches are different but nothing prevents JAX-WS and JAX-RS service beans delegating to some shared implementation, for example, the one reading or writing data to the database.

Often enough, the SOAP developers who would like to experiment with JAX-RS or make their SOAP services more HTTP-centric, wish to reuse the same code serving both SOAP and plain HTTP requests. CXF lets do the combinataion using either the Java-First or WSDL-first approach.

5.1. Using Java-First Approach

Combining JAX-RS and JAX-WS using the Java-First approach is about annotating the interface or concrete class with both JAX-WS and JAX-RS annotations:

```
package server;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
```

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

@WebService
@Path("/bookstore")
@Consumes("application/xml")
@Produces("application/xml")
public interface BookStoreJaxrsJaxws {

    @WebMethod
    @GET
    @Path("/{id}")
    @Consumes("application/xml")
    Book getBook(@PathParam("id") @WebParam(name = "id") Long id);

    @WebMethod
    @POST
    @Path("/books")
    Book addBook(@WebParam(name = "book") Book book);
}
```

In this example, the BookStoreJaxrsJaxws implementation class will need to be declared as a standalone bean and referenced from JAX-WS and JAX-RS endpoints. Both JAX-WS and JAX-RS proxies will be able to reuse this interface for consuming SOAP and RESTful web services. Please see the jaxrs_jaxws_java_first demo in the Talend ESB Examples distribution for a concrete example.

5.2. Using Document-First Approach

Many SOAP developers prefer a document-first (or WSDL-first) approach toward developing WS services. After a WSDL document has been created, the code generator produces the boilerplate server and possibly the client code, with the generated interface such as BookStoreJaxrsJaxws in the previous section containing the JAX-WS annotations only.

Attempting to reuse the same interface by adding JAX-RS annotations is not realistic given that the interface will be re-generated the next time the code generator runs. CXF JAX-RS provides an advanced no-annotations feature which can be used to apply the external JAX-RS like model to a given interface or class and turn it into a RESTful resource without modifying it directly. Please check the jaxrs_jaxws_document_first demo in the Talend ESB examples distribution and see how this feature is used.

Chapter 6. Talend ESB Service Recommended Project Structure

Developers are encouraged, whenever feasible, to have the following project structure for developing JAX-WS and JAX-RS applications:

Folder	Description
client	Provides the client for interacting with the deployed service. Usually activated by navigating to this folder and entering **`mvn exec:java`** (see the README file for the particular project for any project-specific information.)
common	Consists of common resource classes, JAXB objects, and any other objects usable by both the client and service modules.
service	Provides the service implementation code. Used both by the war project for servlet deployment and as an OSGi bundle for the Talend OSGi container.
war	Provides a deployable WAR that can be used with servlet containers such as Jetty or Tomcat. Consists mainly of the web.xml and Spring beans.xml file indicating resources and providers that need loading. The Talend OSGi container does not need this module.

Chapter 7. Talend ESB Service Examples

The samples folder of the Talend ESB download contain examples that are provided by the Apache CXF project, as well as Talend ESB-specific examples showing multiple usages of JAX-RS, JMS, Security and CXF interceptors. Each Talend ESB sample has its own README file providing a full description of the sample along with deployment information using embedded Jetty or Talend OSGi container.

The examples provided by the Apache CXF project and bundled with the Talend ESB are listed and summarized on the CXF website; the below listing provides a summary of additional CXF examples provided in the Talend ESB distribution.

Example	Description
jaxws-cxf-sts	Demonstrates having a SOAP client use CXF's stsclient to make a call to a Tomcat-hosted CXF Security Token Service (STS) and subsequently using the SAML token received to make a web service call to a CXF web service provider. Both standalone and OSGi-based clients are shown, as well as standalone, Tomcat-based, and OSGi-based web service provider options given.
jaxws-cxf-sts-advanced	More advanced version of the above showing OSGi deployment of the STS, token providers, token validation, and WSP authorization based on attributes within the SAML token.
jaxws-ws-secpol	Demonstrates using WS-SecurityPolicy and configuration to secure communication between CXF client and servers using various security requirements and including tokens like UsernameToken and SAML assertions.
jaxws-jms-spec	Demonstrates using JAX-WS clients and servers to talk SOAP over JMS, but using the SOAP/JMS Specification for configuration.
interceptors	Demonstrates how a message changes and is manipulated as it passes through the various CXF interceptors.
jaxrs-intro	Shows basic features of the the JAX-RS 1.1 specification and API such as root resources, subresources and HTTP verbs (GET/PUT/POST).

Example	Description
jaxrs-advanced	Building on the jaxrs-intro sample, this demo additionally demonstrates multiple root resource classes, recursive subresources, resource methods consuming and producing data in different formats (XML and JSON), using JAX-RS Response to return status, headers and optional entity, using UriInfo and UriBuilder for working with URI and ExceptionMappers for handling application exceptions.
jaxrs-attachments	Demonstrates how JAX-RS consumers and providers can read and write multipart attachments.
jaxrs-jaxws-authorization	Shows how a Role-Base-Access-Control policy can be applied to JAX-WS and JAX-RS services with the help of the container-managed authentication and CXF security filters enforcing the authorization rules.
jaxrs-jaxws-description-first	Shows how SOAP services created as part of the document (WSDL) first approach process can get RESTified by having a CXF JAX-RS user model resource added which describes how an interface generated by the wsdl-to-java tool can be treated as the JAX-RS root resource.
jaxrs-jaxws-java-first	Shows how a single service instance can be exposed as both JAX-RS and JAX-WS services at the same time and how CXF JAX-RS and JAX-WS proxies can reuse the same code for invoking on the corresponding endpoints.
jaxrs-jms-http	Demonstrates how a JAX-RS HTTP server can be enhanced to receive JMS messages.
jaxrs-oauth	Provides an example of a REST application protected using OAuth security.
jaxrs-attachments	Demonstrates how JAX-RS providers and consumers read and write XOP and regular multipart/mixed attachments
jaxrs-transformations	Demonstrates how CXF can help with maintaining backward and forward compatibility between JAX-RS and JAX-WS consumers and endpoints by using the Transformation Feature of CXF.

Chapter 8. Configuring JMX Integration

To enable JMX integration, register an InstrumentationManager extension with the CXF bus. Using Spring XML on Tomcat, the following minimal XML snippet will enable JMX integration.

```
<import resource="classpath:META-INF/cxf/cxf.xml"/>
...
<bean id="org.apache.cxf.management.InstrumentationManager"
    class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
    <property name="enabled" value="true" />
    <property name="bus" ref="cxf" />
    <property name="usePlatformMBeanServer" value="true" />
</bean>
```

The default InstrumentationManager accepts the following configuration options:

Name	Value	Default
enabled	If the JMX integration should be enabled or not	false
bus	The CXF bus instance to register the JMX extension with	None
server	An optional reference to an MBeanServer instance to register MBeans with. If not supplied, an MBeanServer is resolved using the "usePlatformMBeanServer" and/or "serverName" options.	None
usePlatformMBeanServer	If true and no reference to an MBeanServer is supplied, the JMX extension registers MBeans with the platform MBean server.	false
serverName	If supplied, usePlatformMBeanServer is false, and no reference to an MBeanServer is supplied, the JMX extension registers MBeans with the MBean server carrying this name.	None
createMBServerConnector-Factory	If true, a connector is created on the MBeanServer.	true

Name	Value	Default
threaded	Determines if the creation of the MBean connector is performed in this thread or in a separate thread. Only relevant if createMBServerConnectorFactory is true.	false
daemon	Determines if the MBean connector creation thread is marked as a daemon thread or not. Only relevant if createMBServerConnectorFactory is true.	false
JMXServiceURL	The URL of the connector to create on the MBeanServer. Only relevant if createMBServerConnectorFactory is true.	service:jmx:rmi:///jndi/rmi://localhost:9913/jmxrmi

The MBean instrumentation provided by the above configuration will provide generic information about the WSDL supported by the web service as well as web service administration commands. To see performance metrics of the SOAP call processing, further configuration (see `http://cxf.apache.org/docs/jmx-management.html`) is required – these are disabled by default to avoid unnecessary runtime overhead.

If you're using Maven, make sure you have the following dependency added to the pom.xml for the web service provider:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-management</artifactId>
    <version>${cxf.version}</version>
</dependency>
```

8.1. Example Configuration

Enable JMX integration by adding the following XML to your CXF Spring context:

```
<bean id="org.apache.cxf.management.InstrumentationManager"
    class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
    <property name="bus" ref="cxf" />
    <property name="enabled" value="true" />
    <property name="JMXServiceURL "
            value="service:jmx:rmi:///jndi/rmi://localhost:9914/jmxrmi" />
</bean>
```

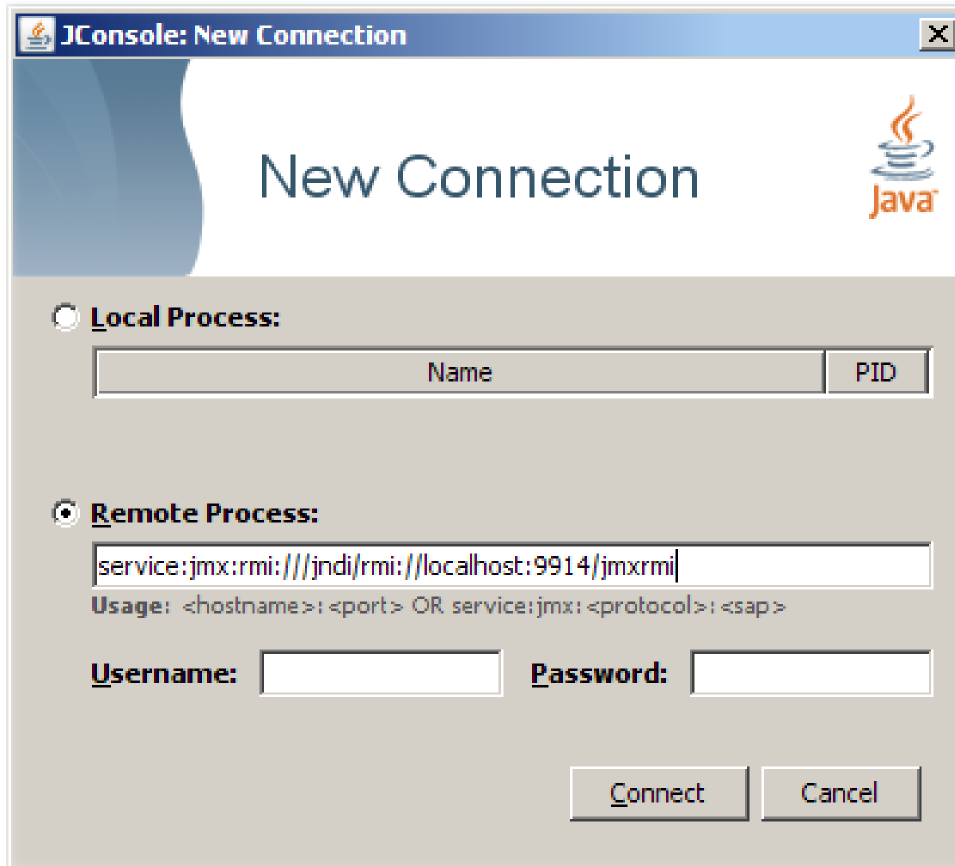
An equivalent configuration of the above instrumentation manager can be directly made within the bus configuration using the corresponding property names having the "bus.jmx" prefix, as in:

```
<cxf:bus bus="cxf">
  <cxf:properties>
    <entry key="bus.jmx.enabled" value="true"/>
    <entry key="bus.jmx.JMXServiceURL"
            value="service:jmx:rmi:///jndi/rmi://localhost:9914/jmxrmi" />
  </cxf:properties>
</cxf:bus>
```

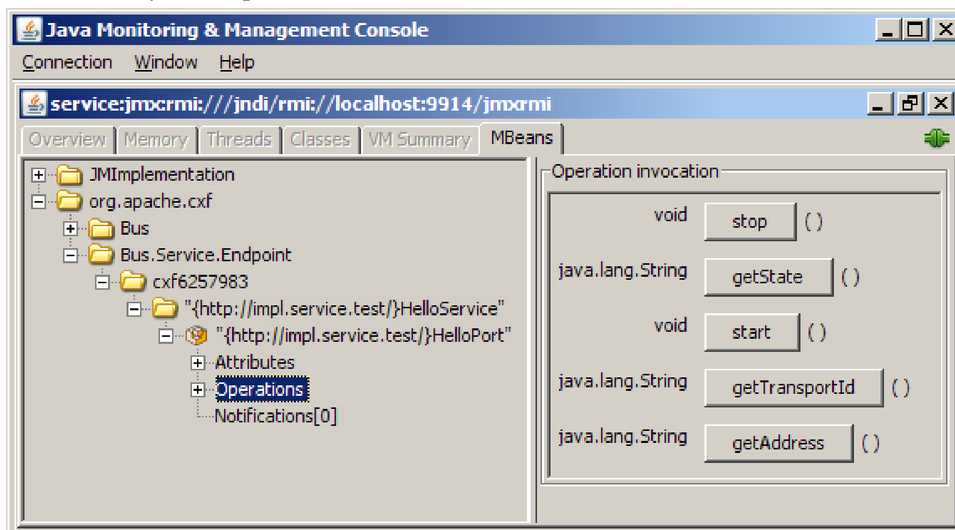
Changes in CXF 2.5.x

If a MBeanServer is available in the Spring context or as an OSGi server (when running in OSGi), the InstrumentationManger will be automatically enabled and will use that MBeanServer and the CXF MBeans will be registered. Therefore, the instrumentation manager configuration shown above is not needed in such cases.

To test the configuration start up your service and connect to it by using JConsole from the JDK.



Then you can browse to your endpoint:



8.2. How to get web service performance metrics

These metrics include Request/Response time, number of calls, and so on.

The CXF management module also provides a feature (the Performance.Counter.Server MBean) which provides aggregate statistics for services running in the CXF Bus. It is not enabled by default to avoid unnecessary runtime overhead during web service call processing.

Here is the configuration snippet that you should add to your Spring context file to be able to view this information:

```
<!-- Wiring the counter repository -->
<bean id="CounterRepository"
        class="org.apache.cxf.management.counters.CounterRepository">
    <property name="bus" ref="cxf" />
</bean>
```

The CounterRepository collects the following metrics: invocations, checked application faults, unchecked application faults, runtime faults, logical runtime faults, total handling time, max handling time, and min handling time. Note a SOAP call will need to occur against the web service before you will see the MBean within your JMX monitoring software.