JBoss Enterprise Application Platform Common Criteria Certification 5

JBoss WS CXF User Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification

Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5
JBoss WS CXF User Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Alessio Soldano
asoldano@redhat.com

Edited by

Rebecca Newton
Red Hat, Engineering Content Services
rnewton@redhat.com

Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This is a guide for installing and running JBoss WS CXF with JBoss Enterprise Application Platform 5.1.0. It includes installation, configuration and tutorials.

Table of Contents

CHAPTER 1. INTRODUCTION 3

CHAPTER 2. INSTALLATION 4

CHAPTER 3. SERVER SIDE INTEGRATION CUSTOMIZATION 5

CHAPTER 4. WS ADDRESSING 7
4.1. USING JAX-WS FOR ENABLING WS-ADDRESSING 7
4.1.1. Using CXF proprietary WSAddressingFeature 7

CHAPTER 5. ADDRESSING TUTORIAL 9
5.1. TURNING ON WS-ADDRESSING 1.0 12

CHAPTER 6. WS-RELIABLE MESSAGING 14

CHAPTER 7. USING WS-RELIABLE MESSAGING 15

CHAPTER 8. WS-RELIABLE MESSAGING TUTORIAL 18
8.1. GENERATING WSDL AND JAX-WS ENDPOINT ARTIFACTS 18
8.2. GENERATING JAX-WS CLIENT ARTIFACTS 21
8.3. TURNING ON WS-RM 1.0 22

CHAPTER 9. WS POLICY FRAMEWORK 29
9.1. USING THE POLICIES FEATURE 29
9.2. SPECIFYING THE LOCATION OF EXTERNAL ATTACHMENTS 29

CHAPTER 10. WS-SECURITY 31
10.1. OVERVIEW OF ENCRYPTION AND SIGNING 31

CHAPTER 11. WSS4J SECURITY ON JBOSS 32
11.1. CREATING THE WEB SERVICE ENDPOINT 32
11.2. TURN ON WS-SECURITY 32
11.2.1. Package and deploy 35
11.3. WS-SECURITY POLICIES 36
11.4. AUTHENTICATION 38
11.5. FURTHER INFORMATION 41
11.5.1. Samples 41
11.5.2. Username/password configuration 41
11.5.3. Crypto Algorithms 41

CHAPTER 12. SOAP MESSAGE LOGGING 42
12.1. DEBUGGING TOOLS 43

APPENDIX A. REVISION HISTORY 44

CHAPTER 1. INTRODUCTION

JBoss Web Services CXF (JBossWS-CXF) is the JBoss Web Services stack implementation internally based on Apache CXF. Apache CXF is an open source services framework. CXF helps you build and develop services using frontend programming Application Programming Interfaces (APIs), like JAX-WS.

CXF includes a broad feature set, but it is primarily focused on the following areas:

Web Services Standard Support

CXF supports a variety of web service standards including:

- SOAP
- WSI Basic Profile.
- WSDL
- WS-Addressing
- WS-Policy
- WS-ReliableMessaging
- WS-Security
- WS-SecurityPolicy
- WS-SecureConversation

Frontends

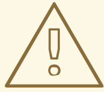
CXF supports a variety of frontend programming models. CXF implements the JAX-WS APIs (TCK compliant). It also includes a simple frontend which allows creation of clients and endpoints without annotations. CXF supports both contract first development with WSDL and code first development starting from Java.

Ease of use

CXF is designed to be intuitive and easy to use.

- There are simple APIs to quickly build code-first services.
- Maven plug-ins to make tooling integration easy.
- JAX-WS API support.
- Spring 2.x XML support to make configuration easier.

CHAPTER 2. INSTALLATION



WARNING

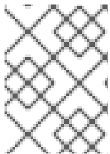
Installing JBoss Web Services CXF is irreversible. You should make a complete backup of your JBoss Enterprise Application Platform installation before installing JBoss Web Services CXF.

Follow these steps to install JBoss Web Services CXF:

Procedure 2.1. Installing CXF

1. **Download the Installer**
   Download and unzip the `jboss-ep-ws-cxf-5.1.0-installer.zip` in the home jboss-as directory directly under the Platform installation root.

2. **Replace WS Native with WS CXF**
   Run `ant` in the created directory, `jbossws-cxf-installer`.



NOTE

This step will replace JBoss Web Services Native with JBoss Web Services CXF in every configuration that contains JBoss Web Services Native.

After completing the process, you should be able to access JBossWS under `http://localhost:8080/jbossws`

CHAPTER 3. SERVER SIDE INTEGRATION CUSTOMIZATION

JBossWS-CXF allows users to deploy their webservice endpoints by simply providing their archives the same way they used to do with JBossWS-Native. However, it is possible to customize the JBossWS and CXF integration by incorporating a CXF configuration file to the endpoint deployment archive. The convention is the following:

- The file name must be **jbossws-cxf.xml**
- For POJO deployments it is located in **WEB-INF** directory
- For EJB3 deployments it is located in **META-INF** directory

If the user does not provide their own CXF configuration file, the default one will be automatically generated during the runtime. For POJO deployments the generated **jbossws-cxf.xml** has the following content:

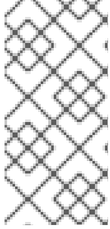
```
<beans
  xmlns='http://www.springframework.org/schema/beans'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:beans='http://www.springframework.org/schema/beans'
  xmlns:jaxws='http://cxf.apache.org/jaxws'
  xsi:schemaLocation='http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd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

  <!-- one or more jaxws:endpoint POJO declarations -->
  <jaxws:endpoint
    id='POJOEndpoint'
    address='http://localhost:8080/pojo_endpoint_archive_name'
    implementor='my.package.POJOEndpointImpl'>
    <jaxws:invoker>
      <bean class='org.jboss.wsf.stack.cxf.InvokerJSE'/>
    </jaxws:invoker>
  </jaxws:endpoint>
</beans>
```

For EJB3 deployments the generated **jbossws-cxf.xml** has the following content:

```
<beans
  xmlns='http://www.springframework.org/schema/beans'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:beans='http://www.springframework.org/schema/beans'
  xmlns:jaxws='http://cxf.apache.org/jaxws'
  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'>

  <!-- one or more jaxws:endpoint EJB3 declarations -->
  <jaxws:endpoint
    id='EJB3Endpoint'
```

```
    address='http://localhost:8080/ejb3_endpoint_archive_name'
    implementor='my.package.EJB3EndpointImpl'>
    <jaxws:invoker>
      <bean class='org.jboss.wsf.stack.cxf.InvokerEJB3'/>
    </jaxws:invoker>
  </jaxws:endpoint>
</beans>
```

Providing custom CXF configuration to the endpoint deployment is useful in cases when users want to use features that are not part of standard JAX-WS specification but CXF still implements them. See Chapter 8, *WS-Reliable Messaging Tutorial* for more information. We provide custom CXF endpoint configuration there to turn on WS-RM feature for the endpoint.



NOTE

When the user incorporates their own CXF configuration to the endpoint archive they must reference either **`org.jboss.wsf.stack.cxf.InvokerJSE`** or the **`org.jboss.wsf.stack.cxf.InvokerEJB3 JAX-WS`** invoker bean there for each JAX-WS endpoint.

CHAPTER 4. WS ADDRESSING

CXF provides support for the 2004-08 and 1.0 versions of WS-Addressing. Users can enable WS-Addressing either using the standard JAX-WS approach or using the Apache CXF WS Addressing Feature on their service.

4.1. USING JAX-WS FOR ENABLING WS-ADDRESSING

As per JAX-WS 2.1 specification, users can enable WS-Addressing on a web service endpoint by simply adding the @javax.xml.soap.Addressing annotation.

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;
@WebService
@Addressing(enabled=true, required=true)
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Hello World!";
   }
}
```

On the client side, WS-Addressing can be explicitly enabled by providing **`org.apache.cxf.ws.addressing.WSAddressingFeature`** when getting the proxy instance from the service:

```
  ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class,
new AddressingFeature());
proxy.sayHello());
```

4.1.1. Using CXF proprietary WSAddressingFeature

To enable WS-Addressing, enable the WSAddressingFeature on your service. If you wish to use XML to configure this, 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>
```

CHAPTER 5. ADDRESSING TUTORIAL

This tutorial will show you how to create client and endpoint communication with WS-Addressing enabled. Creating a WS-Addressing based service and client is very simple. The first step is to create regular JAX-WS service and client configuration; the last step is to configure the addressing on both sides.

The Service

We will start with the following endpoint implementation.

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;
  @WebService
(
   portName = "AddressingServicePort",
   serviceName = "AddressingService",
   targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/wsaddressing",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
)
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Hello World!";
   }
}
```

The above endpoint implements the following endpoint interface:

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/wsaddressing"
)
public interface ServiceIface
{
   @WebMethod
   String sayHello();
}
```

Let's say that compiled endpoint and interface classes are located in directory **`/home/username/wsa/cxf/classes`**. The next step is to generate the JAX-WS artifacts and WSDL that will be part of the endpoint archive.

Generating WSDL and JAX-WS Endpoint Artifacts

We will use the **`wsprovide`** commandline tool to generate WSDL and JAX-WS artifacts. Here's the command:

```
cd JBOSS_HOME/bin
./wsprovide.sh --keep --wsdl \
   --classpath=/home/username/wsa/cxf/classes \
   --output=/home/username/wsa/cxf/wsprovide/generated/classes \
   --resource=/home/username/wsa/cxf/wsprovide/generated/wsdl \
   --source=/home/username/wsa/cxf/wsprovide/generated/src \
   org.jboss.test.ws.jaxws.samples.wsa.ServiceImpl
```

The above command generates the following artifacts:

Compiled classes

SayHello.class

SayHelloResponse.class

Java Sources

SayHello.java

SayHelloResponse.java

Contract Artifacts

AddressingService.wsdl

All previously mentioned generated artifacts will be part of the endpoint archive, but before we create the endpoint archive, we need to reference generated WSDL from the endpoint. We will use the **`wsdlLocation`** annotation attribute. This is the updated endpoint implementation before it is packaged to the **`war`** file:

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;

@WebService
(
   portName = "AddressingServicePort",
   serviceName = "AddressingService",
   wsdlLocation = "WEB-INF/wsdl/AddressingService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/wsaddressing",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
)
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Hello World!";
   }
}
```

The created endpoint war archive consists of the following entries:

```
jar -tvf jaxws-samples-wsa.war
```

```
      0 Mon Apr 21 20:39:30 CEST 2008 META-INF/
    106 Mon Apr 21 20:39:28 CEST 2008 META-INF/MANIFEST.MF
      0 Mon Apr 21 20:39:30 CEST 2008 WEB-INF/
    593 Mon Apr 21 20:39:28 CEST 2008 WEB-INF/web.xml
      0 Mon Apr 21 20:39:30 CEST 2008 WEB-INF/classes/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-INF/classes/org/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-INF/classes/org/jboss/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-INF/classes/org/jboss/test/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-INF/classes/org/jboss/test/ws/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsa/
    374 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsa/ServiceIface.class
    954 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsa/ServiceImpl.class
      0 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsa/jaxws/
    703 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsa/jaxws/SayHello.class
   1074 Mon Apr 21 20:39:26 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsa/jaxws/SayHelloResponse.cla
ss
      0 Mon Apr 21 20:39:30 CEST 2008 WEB-INF/wsdl/
   2378 Mon Apr 21 20:39:28 CEST 2008 WEB-INF/wsdl/AddressingService.wsdl
```

The content of the **web.xml** file is:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app
   version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <servlet>
      <servlet-name>AddressingService</servlet-name>
      <servlet-
class>org.jboss.test.ws.jaxws.samples.wsa.ServiceImpl</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>AddressingService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Writing Regular JAX-WS Client

The following is the regular JAX-WS client using endpoint interface to lookup the webservice: package.org.jboss.test.ws.jaxws.samples.wsa:

```
package.org.jboss.test.ws.jaxws.samples.wsa:
```

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

public final class SimpleServiceTestCase
{
   private final String serviceURL = "http://localhost:8080/jaxws-samples-
wsa/AddressingService";

   public static void main(String[] args) throws Exception
   {
      // create service
      QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wsaddressing", "AddressingService");
      URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      ServiceIface proxy =
(ServiceIface)service.getPort(ServiceIface.class);

      // invoke method
      proxy.sayHello();
   }

}
```

We have both endpoint and client implementations but without WS-Addressing in place. Our next goal is to turn on the WS-Addressing feature.

5.1. TURNING ON WS-ADDRESSING 1.0

There are two steps remaining in order to turn on WS-Addressing in JbossWS-CXF.

- Annotate service endpoint with @Addressing annotation.
- Modify client to configure WS-Addressing using the JAX-WS webservice feature.

Updating Endpoint Code to Configure WS-Addressing

Now we need to update endpoint implementation to configure WS-Addressing. Here's the updated endpoint code:

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;

@WebService
(
   portName = "AddressingServicePort",
   serviceName = "AddressingService",
   wsdlLocation = "WEB-INF/wsdl/AddressingService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/wsaddressing",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
```

```
)
@Addressing(enabled=true, required=true)
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Hello World!";
   }
}
```

We have added the JAX-WS 2.1 Addressing annotation to configure WS-Addressing. The next step is to repackage the endpoint archive to apply this change.

Updating Client Code to Configure WS-Addressing

We need to update client implementation to configure WS-Addressing. Here's the updated client code:

```
package org.jboss.test.ws.jaxws.samples.wsa;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.AddressingFeature;

public final class AddressingTestCase
{
   private final String serviceURL = "http://localhost:8080/jaxws-samples-
wsa/AddressingService";

   public static void main(String[] args) throws Exception
   {
      // construct proxy
      QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wsaddressing", "AddressingService");
      URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      ServiceIface proxy =
(ServiceIface)service.getPort(ServiceIface.class,  new
AddressingFeature());
      // invoke method
      proxy.sayHello();
   }

}
```

We now have both JAX-WS client and endpoint communicating with each other using WS-Addressing.

CHAPTER 6. WS-RELIABLE MESSAGING

CXF supports the February 2005 version of the Web Service Reliable Messaging Protocol specification. Like most other features in CXF, it is interceptor based. The WS-Reliable Messaging implementation consists of four interceptors in total. These are listed below.

`org.apache.cxf.ws.rm.RMOutInterceptor`

Responsible for:

- Sending **`CreateSequence`** requests.
- Waiting for their **`CreateSequenceResponse`** responses.
- Collecting the sequence properties (id and message number) for an application message.

`org.apache.cxf.ws.rm.RMInInterceptor`

Intercepting and processing RM protocol messages, 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.

Interceptor Based QoS

The presence of the RM interceptors on the respective interceptor chains alone will ensure 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. The RM interceptors are also 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 the 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. By default, CXF attempts to maximize the lifetime of a sequence. This reduces the overhead incurred by the RM protocol messages, however you can choose to enforce the use of a separate sequence per application message by configuring the source of the RM sequence termination policy (setting the maximum sequence length to one). See the Chapter 7, *Using WS-Reliable Messaging* for more details on configuring this and other aspects of the reliable exchange.

CHAPTER 7. USING WS-RELIABLE MESSAGING

In order for JBoss WS-CXF/CXF to establish reliable messaging between two points, the CXF RM and addressing interceptors need to be added to the interceptor chains. This can be achieved in one of the ways outlined below.

Using the RMAssertion and the CXF WS-Policy Framework

The RM interceptors will be automatically added to their respective interceptor chains by the policy framework if the following occurs:

1. A Policy with an RMAssertion element is attached to the **wsdl:service** element (or any other WSDL element that is an attachment point for Policy or PolicyReference elements according to the rules for WS-Policy Attachments).

2. The CXF WS-Policy Framework is enabled

The assertion attributes control the behavior of the source or destination. For example, to enable the WS-Policy Framework on the server side, your configuration file will look like this:

```
<jaxws:endpoint ...>
    <jaxws:features>
        <p:policies/>
    </jaxws:features>
</jaxws:endpoint>
```

Your WSDL will look like this:

```
<wsp:Policy wsu:Id="RM" xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd">
    <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
    </wsam:Addressing>
    <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
    </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
    <wsdl:port binding="tns:GreeterSOAPBinding" name="GreeterPort">
        <soap:address
location="http://localhost:9020/SoapContext/GreeterPort"/>
        <wsp:PolicyReference URI="#RM"
xmlns:wsp="http://www.w3.org/2006/07/ws-policy"/>
    </wsdl:port>
</wsdl:service>
```

Instead of attaching the PolicyReference to the **wsdl:port** element, you can also specify it as a child element of the policies featured, such as the server endpoint.

```
<wsp:Policy wsu:Id="="RM" xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
...>
</wsp:Policy>

<jaxws:endpoint ...>
  <jaxws:features>
 <p:policies>
   <wsp:PolicyReference URI="#RM" xmlns:wsp="http://www.w3.org/2006/07/ws-
policy"/>
 </p:policies>
  </jaxws:features>
</jaxws:endpoint>
```

Using the Reliable Messaging Feature

You can use the ReliableMessaging feature if you do not want to involve the WS-Policy Framework, or want to configure additional parameters such as the sequence termination policy or the persistent store. The supported child elements are listed below.

RMAssertion

An element of type RMAssertion.

deliveryAssurance

An element of type DeliveryAssuranceType that describes the delivery assurance that should apply (**`AtMostOnce`**, **`AtLeastOnce`**, **`InOrder`**).

sourcePolicy

An element of type SourcePolicyType that allows you to configure details of the RM source, such as whether an offer should always be included in a **`CreateSequence`** request, or the sequence termination policy.

destinationPolicy

An element of type DestinationPolicyType that allows you to configure details of the RM destination, such as whether inbound offers should be accepted.

store

The store to use (default: **`null`**). This must be an element of type jdbcStore (in the same namespace), or a bean or a reference to a bean that implements the RMStore interface.

The jbdcStore element type is described below.

The following example is applied at bus level.

```
<cxf:bus>
    <cxf:features>
        <wsa:addressing/>
        <wsrm-mgr:reliableMessaging>
            <wsrm-policy:RMAssertion>
                <wsrm-policy:BaseRetransmissionInterval
Milliseconds="4000"/>
                <wsrm-policy:AcknowledgementInterval
Milliseconds="2000"/>
            </wsrm-policy:RMAssertion>
```

```
                <wsrm-mgr:sourcePolicy>
                    <wsrm-mgr:sequenceTerminationPolicy maxLength="5"/>
                </wsrm-mgr:sourcePolicy>
                <wsrm-mgr:destinationPolicy acceptOffers="false">
                <wsrm:store>
                   <ref bean="myStore"/>
                </wsrm:store>
            </wsrm-mgr:reliableMessaging>
        </cxf:features>
</cxf:bus>
```

Configuring the Reliable Messaging Store

To enable persistence, you must specify the object implementing the persistent store for RM. You can develop your own, or use the JDBC based store that comes with CXF (**`class org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore`**). You can configure the latter using a custom jdbcStore bean. The supported attributes are in the table below.

Table 7.1. Attributes

Attribute name	String	Default
driverClassName	String	org.apache.derby.jdbc.Embedded Driver
userName	String	null
passWord	String	null
url	String	jdbc:derby:rmdb;create=true

Here is an example:

```
  <wsrm-mgr:jdbcStore id="myStore"
    driverClassName="org.apache.derby.jdbc.ClientDriver"
    url="jdbc:derby://localhost:1527/rmdb;create=true"
    password="password"/>
```

Configuring the Reliable Messaging Manager Manually

To configure properties of the RM Manager, you can use the RMManager element. It supports the same child elements as the ReliableMessaging feature element above. For example, without using features, you can determine that sequences should have a maximum length of five as follows:

```
<wsrm-mgr:rmManager xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager">
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="5"/>
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:rmManager>
```

CHAPTER 8. WS-RELIABLE MESSAGING TUTORIAL

In this sample we show you how to create client and endpoint communication using WS-Reliable Messaging 1.1. Creating the WS-RM based service and client is very simple. The user needs to create regular JAX-WS service and client first. The last step is to configure WS-RM.

We will start with the following endpoint implementation:

```
package org.jboss.test.ws.jaxws.samples.wsrm.service;

import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   name = "SimpleService",
   serviceName = "SimpleService",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsrm"
)
public class SimpleServiceImpl
{
   @Oneway
   @WebMethod
   public void ping()
   {
      System.out.println("ping()");
   }

   @WebMethod
   public String echo(String s)
   {
      System.out.println("echo(" + s + ")");
      return s;
   }
}
```

Let's say that compiled endpoint class is in directory **`/home/username/wsrm/cxf/classes`**. Our next step is to generate JAX-WS artifacts and WSSDL.

8.1. GENERATING WSDL AND JAX-WS ENDPOINT ARTIFACTS

We will use the **`wsprovide`** commandline tool to generate WSDL and JAX-WS artifacts. Here's the command:

```
cd $JBOSS_HOME/bin
./wsprovide.sh --keep --wsdl \
   --classpath=/home/username/wsrm/cxf/classes \
   --output=/home/username/wsrm/cxf/wsprovide/generated/classes \
   --resource=/home/username/wsrm/cxf/wsprovide/generated/wsdl \
   --source=/home/username/wsrm/cxf/wsprovide/generated/src \
   org.jboss.test.ws.jaxws.samples.wsrm.service.SimpleServiceImpl
```

The above command generates the following artifacts:

Compiled classes

Echo.class

Echo response.class

Ping.class

Java sources

Echo.java

EchoResponse.java

Ping.java

Contract artifacts

SimpleService.wsdl

The artifacts generated above will be part of the endpoint archive, but before we create the endpoint archive we need to reference generated WSDL from the endpoint. To achieve that we will use the **wsdlLocation** annotation attribute. Here's the updated endpoint implementation before it is packaged to the **war** file:

```
package org.jboss.test.ws.jaxws.samples.wsrm.service;

import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   name = "SimpleService",
   serviceName = "SimpleService",
   wsdlLocation = "WEB-INF/wsdl/SimpleService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsrm"
)
public class SimpleServiceImpl
{
   @Oneway
   @WebMethod
   public void ping()
   {
      System.out.println("ping()");
   }

   @WebMethod
   public String echo(String s)
   {
      System.out.println("echo(" + s + ")");
      return s;
   }
}
```

The created endpoint war archive consists of the following entries:

```
jar -tvf jaxws-samples-wsrm.war
     0 Wed Apr 16 14:39:22 CEST 2008 META-INF/
   106 Wed Apr 16 14:39:20 CEST 2008 META-INF/MANIFEST.MF
     0 Wed Apr 16 14:39:22 CEST 2008 WEB-INF/
   591 Wed Apr 16 14:39:20 CEST 2008 WEB-INF/web.xml
     0 Wed Apr 16 14:39:22 CEST 2008 WEB-INF/classes/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-INF/classes/org/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-INF/classes/org/jboss/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-INF/classes/org/jboss/test/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-INF/classes/org/jboss/test/ws/
     0 Wed Apr 16 14:39:20 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/
     0 Wed Apr 16 14:39:20 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/
     0 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/
  1235 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/SimpleServiceImpl
.class
   997 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/Echo.class
  1050 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/EchoRespons
e.class
   679 Wed Apr 16 14:39:18 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/Ping.class
     0 Wed Apr 16 14:39:22 CEST 2008 WEB-INF/wsdl/
  2799 Wed Apr 16 14:39:20 CEST 2008 WEB-INF/wsdl/SimpleService.wsdl
```

The content of **web.xml** file is:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app
   version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <servlet>
      <servlet-name>SimpleService</servlet-name>
      <servlet-
class>org.jboss.test.ws.jaxws.samples.wsrm.service.SimpleServiceImpl</serv
let-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>SimpleService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

8.2. GENERATING JAX-WS CLIENT ARTIFACTS

Before we write the regular JAX-WS client we need to generate client artifacts from WSDL. Here's the command to achieve that:

```
cd $JBOSS_HOME/bin
./wsconsume.sh --keep \
   --package=org.jboss.test.ws.jaxws.samples.wsrm.generated \
   --output=/home/username/wsrm/cxf/wsconsume/generated/classes \
   --source=/home/username/wsrm/cxf/wsconsume/generated/src \
   /home/username/wsrm/cxf/wsprovide/generated/wsdl/SimpleService.wsdl
```

The artifacts that have been generated are below.

Compiled classes

Echo.class

ObjectFactory.class

Ping.class

SimpleService_Service.class

EchoResponse.class

package-info.class

SimpleService.class

SimpleService_SimpleServicePort_Client.class

Java sources

Echo.java

ObjectFactory.java

Ping.java

SimpleService_Service.java

EchoResponse.java

package-info.java

SimpleService.java

SimpleService_SimpleServicePort_Client.java

The last step is to write the regular JAX-WS client using generated artifacts.

Writing Regular JAX-WS Client

The following is the regular JAX-WS client using generated artifacts:

```
package org.jboss.test.ws.jaxws.samples.wsrm.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import org.jboss.test.ws.jaxws.samples.wsrm.generated.SimpleService;

public final class SimpleServiceTestCase
{

   private static final String serviceURL = "http://localhost:8080/jaxws-
samples-wsrm/SimpleService";

   public static void main(String[] args) throws Exception
   {
      // create service
      QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wsrm", "SimpleService");
      URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      SimpleService proxy =
(SimpleService)service.getPort(SimpleService.class);

      // invoke methods
      proxy.ping(); // one way call
      proxy.echo("Hello World!"); // request responce call
   }

}
```

Now we have both endpoint and clinet implementations but without WS-Reliable Messaging in place. Our next goal is to turn on the WS-RM feature.

8.3. TURNING ON WS-RM 1.0

Four steps are necessary to turn on WS-RM in JBossWS-CXF. They are outline below.

- Extend WSDL with WS-Policy containing both WSRM and WS-Addressing policy.
- Provide **`jbossws-cxf.xml`** endpoint configuration file.
- Provide client CXF configuration.
- Update client code to read CXF configuration file.

Extending WSDL Using WS-Policy

To activate WSRM we need to extend WSDL with WSRM and addressing policy. Here is how it looks:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="SimpleService"
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wsrm"
xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wsrm"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsp="http://www.w3.org/2006/07/ws-policy">

  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Created WS-Policy with WSRM addressing assertions -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <wsp:UsingPolicy/>
  <wsp:Policy wsu:Id="wsrm10policy" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <wswa:UsingAddressing
xmlns:wswa="http://www.w3.org/2006/05/addressing/wsdl">
      <wsp:Policy/>
    <wswa:UsingAddressing>
    <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"/>
  </wsp:Policy>

  <wsdl:types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wsrm"
attributeFormDefault="unqualified" elementFormDefault="unqualified"
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wsrm">
<xsd:element name="ping" type="tns:ping"/>
<xsd:complexType name="ping">
<xsd:sequence/>
</xsd:complexType>
<xsd:element name="echo" type="tns:echo"/>
<xsd:complexType name="echo">
<xsd:sequence>
<xsd:element minOccurs="0" name="arg0" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="echoResponse" type="tns:echoResponse"/>
<xsd:complexType name="echoResponse">
<xsd:sequence>
<xsd:element minOccurs="0" name="return" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
  </wsdl:types>
  <wsdl:message name="echoResponse">
    <wsdl:part name="parameters" element="tns:echoResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="echo">
    <wsdl:part name="parameters" element="tns:echo">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="ping">
    <wsdl:part name="parameters" element="tns:ping">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="SimpleService">
    <wsdl:operation name="ping">
      <wsdl:input name="ping" message="tns:ping">
    </wsdl:input>
```

```
    </wsdl:operation>
    <wsdl:operation name="echo">
      <wsdl:input name="echo" message="tns:echo">
    </wsdl:input>
      <wsdl:output name="echoResponse" message="tns:echoResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SimpleServiceSoapBinding" type="tns:SimpleService">
    <!-- - - - - - - - - - - - - - - - - - - - - - -->
    <!-- Associated WS-Policy with the binding -->
    <!-- - - - - - - - - - - - - - - - - - - - - - -->
    <wsp:PolicyReference URI="#wsrm10policy"/>
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="ping">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="ping">
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
    <wsdl:operation name="echo">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="echo">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="echoResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SimpleService">
    <wsdl:port name="SimpleServicePort"
binding="tns:SimpleServiceSoapBinding">
      <soap:address location="http://localhost:9090/hello"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

We added **`wsp:UsingPolicy;wsp:Policy`** and **`wsp:PolicyReference`** elements to WSDL.

Providing `jbossws-cxf.xml` Endpoint Configuration File

This is the JBossWS CXF integration extension file: Chapter 3, *Server Side Integration Customization* . In our case, the relevant content is as follows:

```
<beans
  xmlns='http://www.springframework.org/schema/beans'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:beans='http://www.springframework.org/schema/beans'
  xmlns:jaxws='http://cxf.apache.org/jaxws'
  xmlns:wsp='http://www.w3.org/2006/07/ws-policy'
  xmlns:p='http://cxf.apache.org/policy'
  xsi:schemaLocation='http://www.springframework.org/schema/beans
```

```
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://cxf.apache.org/policy
    http://cxf.apache.org/schemas/policy.xsd
    http://www.w3.org/2006/07/ws-policy
    http://www.w3.org/2006/07/ws-policy.xsd
    http://cxf.apache.org/jaxws
    http://cxf.apache.org/schemas/jaxws.xsd'>

  <wsp:Policy wsu:Id="wsrm10policy" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <wswa:UsingAddressing
xmlns:wswa="http://www.w3.org/2006/05/addressing/wsdl"/>
    <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"/>
  </wsp:Policy>

  <jaxws:endpoint
    id='SimpleServiceImpl'
    address='http://localhost:8080/jaxws-samples-wsrm'

implementor='org.jboss.test.ws.jaxws.samples.wsrm.service.SimpleServiceImp
l'>
    <jaxws:invoker>
      <bean class='org.jboss.wsf.stack.cxf.InvokerJSE'/>
    </jaxws:invoker>
    <jaxws:features>
      <p:policies>
        <wsp:PolicyReference URI="#wsrm10policy"
xmlns:wsp="http://www.w3.org/2006/07/ws-policy"/>
      </p:policies>
    </jaxws:features>
  </jaxws:endpoint>

</beans>
```

We need to include this **jbossws-cxf.xml** CXF configuration file in the **WEB-INF** directory of the endpoint archive because we are creating a POJO deployment.

```
jar -tvf jaxws-samples-wsrm.war
     0 Wed Apr 16 19:05:38 CEST 2008 META-INF/
   106 Wed Apr 16 19:05:36 CEST 2008 META-INF/MANIFEST.MF
     0 Wed Apr 16 19:05:38 CEST 2008 WEB-INF/
   591 Wed Apr 16 19:05:36 CEST 2008 WEB-INF/web.xml
     0 Wed Apr 16 19:05:38 CEST 2008 WEB-INF/classes/
     0 Wed Apr 16 19:05:32 CEST 2008 WEB-INF/classes/org/
     0 Wed Apr 16 19:05:32 CEST 2008 WEB-INF/classes/org/jboss/
     0 Wed Apr 16 19:05:32 CEST 2008 WEB-INF/classes/org/jboss/test/
     0 Wed Apr 16 19:05:32 CEST 2008 WEB-INF/classes/org/jboss/test/ws/
     0 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/
     0 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/
     0 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/
     0 Wed Apr 16 19:05:34 CEST 2008 WEB-
```

```
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/
     0 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/
  1235 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/SimpleServiceImpl
.class
   997 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/Echo.class
  1050 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/EchoRespons
e.class
   679 Wed Apr 16 19:05:34 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsrm/service/jaxws/Ping.class
  1554 Wed Apr 16 19:05:36 CEST 2008 WEB-INF/jbossws-cxf.xml
     0 Wed Apr 16 19:05:38 CEST 2008 WEB-INF/wsdl/
  3237 Wed Apr 16 19:05:36 CEST 2008 WEB-INF/wsdl/SimpleService.wsdl
```

The next step is to create the client CXF configuration file that will be used by the client. It activates the WS-RM protocol for the CXF client. We will name this file **cxf.xml** in our sample. The content of this file is as follows:

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cxf="http://cxf.apache.org/core"
  xmlns:wsa="http://cxf.apache.org/ws/addressing"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
  xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
  xsi:schemaLocation="
    http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd
    http://cxf.apache.org/transports/http/configuration
    http://cxf.apache.org/schemas/configuration/http-conf.xsd
    http://schemas.xmlsoap.org/ws/2005/02/rm/policy
    http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd
    http://cxf.apache.org/ws/rm/manager
    http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <cxf:bus>
    <cxf:features>
      <cxf:logging/>
      <wsa:addressing/>
      <wsrm-mgr:reliableMessaging>
        <wsrm-policy:RMAssertion>
          <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
          <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
        </wsrm-policy:RMAssertion>
        <wsrm-mgr:destinationPolicy>
          <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
        </wsrm-mgr:destinationPolicy>
      </wsrm-mgr:reliableMessaging>
    </cxf:features>
```

```
  </cxf:bus>

</beans>
```

We are almost done. The client configuration needs to be picked up by the client classloader. In order to achieve that the **cxf.xml** has to be put in the **META-INF** directory of client jar. That jar should then be provided when setting the class loader.

Alternatively you can read the bus configuration programmatically.

Updating Client Code to Read Bus Configuration File

Here's the last piece of the updated CXF client:

```
package org.jboss.test.ws.jaxws.samples.wsrm.client;

import java.net.URL;
import java.io.File;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.jboss.test.ws.jaxws.samples.wsrm.generated.SimpleService;

public final class SimpleServiceTestCase
{

   private static final String serviceURL = "http://localhost:8080/jaxws-
samples-wsrm/SimpleService";

   public static void main(String[] args) throws Exception
   {
      // create bus
      SpringBusFactory busFactory = new SpringBusFactory();
      URL cxfConfig = new
File("resources/jaxws/samples/wsrm/cxf.xml").toURL();
      Bus bus = busFactory.createBus(cxfConfig);
      busFactory.setDefaultBus(bus);

      // create service
      QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wsrm", "SimpleService");
      URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      SimpleService proxy =
(SimpleService)service.getPort(SimpleService.class);

      // invoke methods
      proxy.ping(); // one way call
      proxy.echo("Hello World!"); // request responce call

      // shutdown bus
      bus.shutdown(true);
   }
```

```
}
```

CHAPTER 9. WS POLICY FRAMEWORK

The calculation of the effective policy for each message as well as verification that the alternatives for that policy are supported happens in interceptors.

9.1. USING THE POLICIES FEATURE

The policies feature supports the following attributes:

ignoreUnknownAssertions

Indicates an exception should be thrown when encountering assertions for which no AssertionBuilders are registered (default:**`true`**). When set to false, a warning will be logged instead.

namespace

The namespace of the WS-Policy Framework specification (default: http://www.w3.org/ns/ws-policy).

The element also supports the following child elements:

alternativeSelector

A bean or reference to a bean that implements the **`org.apache.cxf.ws.policy.selector.AlternativeSelector`** interface. The default selector chooses the minimal alternative; that is, the one with the least number of assertions.

In addition, the element can have any number of Policy or PolicyReference child elements. This has the same effect as if the Policy or PolicyReference elements were attached to the **`wsdl:port`** element of the WSDL contract of the client or server endpoint to which the feature is applied (or to all endpoints if the feature is applied to the bus).

For example, to apply this feature to the bus and prevent exceptions being thrown when encountering unknown assertions:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:p="http://cxf.apache.org/policy"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
http://cxf.apache.org/policy http://cxf.apache.org/schemas/policy.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <cxf:bus>
        <cxf:features>
            <p:policies ignoreUnknownAssertions="true"/>
        </cxf:features>
    </cxf:bus>
</beans>
```

9.2. SPECIFYING THE LOCATION OF EXTERNAL ATTACHMENTS

To specify the location of an external attachment that the policy framework should take into consideration when aggregating the policies applying to a specific message, you can use the **`<externalAttachment>`** element in the same namespace. The following attribute is supported.

location

Location of the external attachment document. This takes the form of http://static.springsource.org/spring/docs/2.0.x/reference/resources.html type property, for example, **`classpath:etc/policies.xml`** or **`file:///x1/resources/polcies.xml`**.

Below is an example:

```
<p:externalAttachment
location="classpath:org/apache/cxf/systest/ws/policy/addr-external.xml"/>
```

You can have any number of **`<externalAttachment>`** elements in your configuration file.

CHAPTER 10. WS-SECURITY

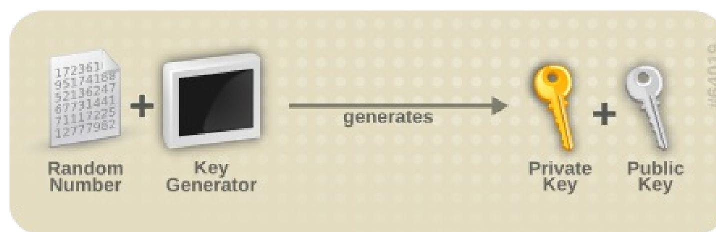
WS-Security provides the means to secure your services beyond transport level protocols such as **HTTPS**. Through a number of standards such as XML-Encryption, and headers defined in the WS-Security standard, it allows you to:

- Pass authentication tokens between services.
- Encrypt messages or parts of messages.
- Sign messages.
- Timestamp messages.

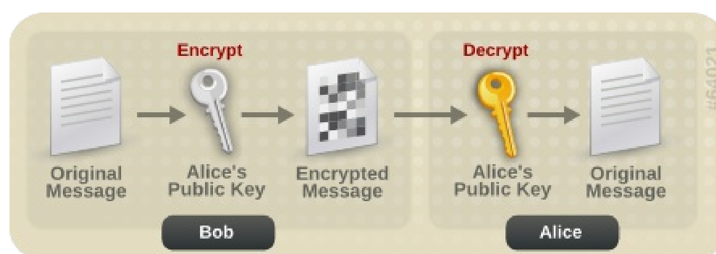
Currently, CXF implements WS-Security by integrating WSS4J. To use the integration, you'll need to configure these interceptors and add them to your service or client respectively.

10.1. OVERVIEW OF ENCRYPTION AND SIGNING

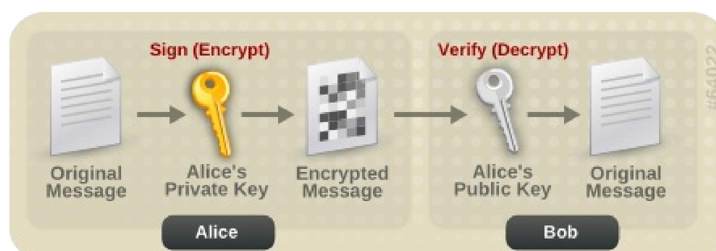
WS-Security makes heavy use of public and private key cryptography. It is helpful to understand these basics to really understand how to configure WS-Security. With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function.



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.



CHAPTER 11. WSS4J SECURITY ON JBOSS

Here is a brief chapter on how to use Chapter 10, *WS-Security* on JBossWS-CXF. Here you'll find some explanations on how to create a simple application and what you need to do to leverage WSS4J security on JBoss.

11.1. CREATING THE WEB SERVICE ENDPOINT

First of all you need to create the web service endpoint or client using JAX-WS. This can be achieved in many ways. For instance you might want to:

1. Write your endpoint implementation, then run the **wsprovide** JBoss commandline tool which generates the service contract.
2. Run the **wsconsume** JBoss commandline tool to get the client artifacts from the service contract (top-down approach).
3. Write your client implementation.

11.2. TURN ON WS-SECURITY

WSS4J security is triggered through interceptors that are added to the service and client individually or as required. These interceptors allow you to perform the most common WS-Security related processes:

- Pass authentication tokens between services.
- Encrypt messages or parts of messages.
- Sign messages.
- Timestamp messages.

Interceptors can be added either programmatically or through the Spring xml configuration of endpoints. For instance, on server side, you can configure signature and encryption in the **jbossws-cxf.xml** file this way:

```
<beans
  xmlns='http://www.springframework.org/schema/beans'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:beans='http://www.springframework.org/schema/beans'
  xmlns:jaxws='http://cxf.apache.org/jaxws'
  xsi:schemaLocation='http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd
    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="Sign_Request"
class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
      <map>
        <entry key="action" value="Timestamp Signature Encrypt"/>
        <entry key="signaturePropFile" value="bob.properties"/>
```

```
        <entry key="decryptionPropFile" value="bob.properties"/>
        <entry key="passwordCallbackClass"
value="org.jboss.test.ws.jaxws.samples.wsse.KeystorePasswordCallback"/>
      </map>
    </constructor-arg>
  </bean>

  <bean id="Sign_Response"
class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
    <constructor-arg>
      <map>
        <entry key="action" value="Timestamp Signature Encrypt"/>
        <entry key="user" value="bob"/>
        <entry key="signaturePropFile" value="bob.properties"/>
        <entry key="encryptionPropFile" value="bob.properties"/>
        <entry key="encryptionUser" value="Alice"/>
        <entry key="signatureKeyIdentifier" value="DirectReference"/>
        <entry key="passwordCallbackClass"
value="org.jboss.test.ws.jaxws.samples.wsse.KeystorePasswordCallback"/>
        <entry key="signatureParts" value="{Element}{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd}Timestamp;{Element}
{http://schemas.xmlsoap.org/soap/envelope/}Body"/>
        <entry key="encryptionParts" value="{Element}
{http://www.w3.org/2000/09/xmldsig#}Signature;{Content}
{http://schemas.xmlsoap.org/soap/envelope/}Body"/>
        <entry key="encryptionKeyTransportAlgorithm"
value="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        <entry key="encryptionSymAlgorithm"
value="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      </map>
    </constructor-arg>
   </bean>

  <jaxws:endpoint
    id='ServiceImpl'
    address='http://@jboss.bind.address@:8080/jaxws-samples-wsse-sign-
encrypt'
    implementor='org.jboss.test.ws.jaxws.samples.wsse.ServiceImpl'>
    <jaxws:invoker>
      <bean class='org.jboss.wsf.stack.cxf.InvokerJSE'/>
    </jaxws:invoker>
    <jaxws:outInterceptors>
        <bean
class="org.apache.cxf.binding.soap.saaj.SAAJOutInterceptor"/>
        <ref bean="Sign_Response"/>
    </jaxws:outInterceptors>
    <jaxws:inInterceptors>
        <ref bean="Sign_Request"/>
        <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor"/>
    </jaxws:inInterceptors>
  </jaxws:endpoint>
</beans>
```

This specifies the whole security configuration (including algorithms and elements to be signed or encrypted); moreover it references a properties file (**bob.properties**) providing the keystore-related information:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.c
rypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=bob
org.apache.ws.security.crypto.merlin.file=bob.jks
```

As you can see in the **jbossws-cxf.xml** file above, a keystore password callback handler is also configured; while the properties file has the password for the keystore, this callback handler is used to set password for each key (it has to match the one used when each key was imported in the store). Here is an example:

```
package org.jboss.test.ws.jaxws.samples.wsse;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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 KeystorePasswordCallback implements CallbackHandler
{
   private Map<String, String> passwords = new HashMap<String, String>();

   public KeystorePasswordCallback()
   {
      passwords.put("alice", "password");
      passwords.put("bob", "password");
   }

   public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException
   {
      for (int i = 0; i < callbacks.length; i++)
      {
         WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];
         String pass = passwords.get(pc.getIdentifer());
         if (pass != null)
         {
            pc.setPassword(pass);
            return;
         }
      }
   }

   public void setAliasPassword(String alias, String password)
   {
```

```
            passwords.put(alias, password);
        }
    }
```

On the client side, you can similarly setup the interceptors programmatically; here is an excerpt of the client for the above described endpoint:

```
Endpoint cxfEndpoint = client.getEndpoint();
Map<String,Object> outProps = new HashMap<String,Object>();
outProps.put("action", "Timestamp Signature Encrypt");
outProps.put("user", "alice");
outProps.put("signaturePropFile", "META-INF/alice.properties");
outProps.put("signatureKeyIdentifier", "DirectReference");
outProps.put("passwordCallbackClass",
"org.jboss.test.ws.jaxws.samples.wsse.KeystorePasswordCallback");
outProps.put("signatureParts", "{Element}{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd}Timestamp;{Element}
{http://schemas.xmlsoap.org/soap/envelope/}Body");
outProps.put("encryptionPropFile", "META-INF/alice.properties");
outProps.put("encryptionUser", "Bob");
outProps.put("encryptionParts", "{Element}
{http://www.w3.org/2000/09/xmldsig#}Signature;{Content}
{http://schemas.xmlsoap.org/soap/envelope/}Body");
outProps.put("encryptionSymAlgorithm",
"http://www.w3.org/2001/04/xmlenc#tripledes-cbc");
outProps.put("encryptionKeyTransportAlgorithm",
"http://www.w3.org/2001/04/xmlenc#rsa-1_5");
WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor(outProps); //request
cxfEndpoint.getOutInterceptors().add(wssOut);
cxfEndpoint.getOutInterceptors().add(new SAAJOutInterceptor());

Map<String,Object> inProps= new HashMap<String,Object>();
inProps.put("action", "Timestamp Signature Encrypt");
inProps.put("signaturePropFile", "META-INF/alice.properties");
inProps.put("passwordCallbackClass",
"org.jboss.test.ws.jaxws.samples.wsse.KeystorePasswordCallback");
inProps.put("decryptionPropFile", "META-INF/alice.properties");
WSS4JInInterceptor wssIn = new WSS4JInInterceptor(inProps); //response
cxfEndpoint.getInInterceptors().add(wssIn);
cxfEndpoint.getInInterceptors().add(new SAAJInInterceptor());
```

11.2.1. Package and deploy

To deploy your web service endpoint, you need to package the following files along with your service implementation and WSDL contract:

1. The **`jbossws-cxf.xml`** descriptor.
2. The properties file.
3. The keystore file (if required for signature/encryption).
4. The keystore password callback handler class.

For instance, here are the archive contents for the signature and encryption sample (POJO endpoint) mentioned before:

```
[cxf-tests]$ jar -tvf target/test-libs/jaxws-samples-wsse-sign-encrypt.war
   0 Tue Jun 03 19:41:26 CEST 2008 META-INF/
 106 Tue Jun 03 19:41:24 CEST 2008 META-INF/MANIFEST.MF
   0 Tue Jun 03 19:41:26 CEST 2008 WEB-INF/
   0 Tue Jun 03 19:41:26 CEST 2008 WEB-INF/classes/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/classes/org/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/classes/org/jboss/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/classes/org/jboss/test/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/classes/org/jboss/test/ws/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
1628 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/KeystorePasswordCallback.
class
 364 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/ServiceIface.class
 859 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/ServiceImpl.class
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/jaxws/
 685 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/jaxws/SayHello.class
1049 Tue Jun 03 19:41:24 CEST 2008 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/jaxws/SayHelloResponse.cl
ass
2847 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/jbossws-cxf.xml
   0 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/wsdl/
1575 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/wsdl/SecurityService.wsdl
 641 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/wsdl/SecurityService_schema1.xsd
1820 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/classes/bob.jks
 311 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/classes/bob.properties
 573 Tue Jun 03 19:41:24 CEST 2008 WEB-INF/web.xml
```

On client side, instead, you only need the properties and keystore files (assuming you set up the interceptors programmatically). You just need to deploy and test your WS-Security-enabled application.

11.3. WS-SECURITY POLICIES

JBossWS-CXF also includes CXF WS-Security Policy implementation, which can be used to configure WS-Security more easily. Instead of manually configuring interceptors in the client or through the **jbossws-cxf.xml** descriptor, you simply provide the right policies in the WSDL contract.

```
 ...
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#SecurityServiceSignPolicy"/>
  ...
  <wsp:Policy wsu:Id="SecurityServiceSignPolicy"
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
```

```
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:AsymmetricBinding
xmlns:sp='http://schemas.xmlsoap.org/ws/2005/07/securitypolicy'>
                <wsp:Policy>
                    <sp:InitiatorToken>
                        <wsp:Policy>
                            <sp:X509Token
sp:IncludeToken='http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/Incl
udeToken/AlwaysToRecipient'>
                                <wsp:Policy>
                                    <sp:WssX509V3Token10 />
                                </wsp:Policy>
                            </sp:X509Token>
                        </wsp:Policy>
                    </sp:InitiatorToken>
                    <sp:RecipientToken>
                        <wsp:Policy>
                            <sp:X509Token
sp:IncludeToken='http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/Incl
udeToken/Always'>
                                <wsp:Policy>
                                    <sp:WssX509V3Token10 />
                                </wsp:Policy>
                            </sp:X509Token>
                        </wsp:Policy>
                    </sp:RecipientToken>
                    <sp:AlgorithmSuite>
                        <wsp:Policy>
                            <sp:Basic256 />
                        </wsp:Policy>
                    </sp:AlgorithmSuite>
                    <sp:Layout>
                        <wsp:Policy>
                            <sp:Strict />
                        </wsp:Policy>
                    </sp:Layout>
                    <sp:OnlySignEntireHeadersAndBody />
                </wsp:Policy>
            </sp:AsymmetricBinding>
            <sp:Wss10
xmlns:sp='http://schemas.xmlsoap.org/ws/2005/07/securitypolicy'>
                <wsp:Policy>
                    <sp:MustSupportRefEmbeddedToken />
                </wsp:Policy>
            </sp:Wss10>
            <sp:SignedParts
xmlns:sp='http://schemas.xmlsoap.org/ws/2005/07/securitypolicy'>
                <sp:Body />
            </sp:SignedParts>
        </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
```

A few properties are also required to be set either in the message context or in the **jbossws-cxf.xml** descriptor.

1. ((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER, new KeystorePasswordCallback());

2. ((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_PROPERTIES, Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));

3. ((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_PROPERTIES, Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));

```
<beans
  xmlns='http://www.springframework.org/schema/beans'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:beans='http://www.springframework.org/schema/beans'
  xmlns:jaxws='http://cxf.apache.org/jaxws'
  xsi:schemaLocation='http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd
    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='ServiceImpl'
    address='http://@jboss.bind.address@:8080/jaxws-samples-wssePolicy-
sign'
    implementor='org.jboss.test.ws.jaxws.samples.wssePolicy.ServiceImpl'>

    <jaxws:properties>
       <entry key="ws-security.signature.properties"
value="bob.properties"/>
       <entry key="ws-security.encryption.properties"
value="bob.properties"/>
       <entry key="ws-security.callback-handler"
value="org.jboss.test.ws.jaxws.samples.wssePolicy.KeystorePasswordCallback
"/>
    </jaxws:properties>
  </jaxws:endpoint>
</beans>
```

11.4. AUTHENTICATION

Read this section to learn how to authenticate a web service user using a number of available methods.

Procedure 11.1. Authenticate a Web Service User

The following procedure describes how to authenticate a web service user with JBossWS.

1. **Secure access to the Stateless Session Bean**
   Secure access to the Stateless Session Bean (SLSB) using the **@RolesAllowed**, **@PermitAll**, **@DenyAll** annotations.

The allowed user roles can be set with these annotations both on the bean class and on any of its business methods.

```
@Stateless
@RolesAllowed("friend")
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

2. **Secure POJO endpoints**
   Secure Plain Old Java Object (POJO) endpoints by defining a <security-constraint> in the **WEB-INF/web.xml** file of the application.

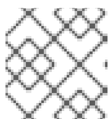
```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All resources</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>friend</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>friend</role-name>
</security-role>
```

3. **Define the security domain**
   Declare the security domain by appending the @SecurityDomain annotation

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

   - You can also modify **JBOSS_HOME/server/*PROFILE*/deploy/jbossws.sar/jboss-management.war/WEB-INF/jboss-web.xml** and specify the security domain.

```
<jboss-web>
  <security-domain>JBossWS</security-domain>
</jboss-web>
```



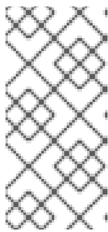
NOTE

For more information about Security Domains, refer to the *JBoss Security Guide*.

4. **Define the security context**

Configure the security context in the **JBOSS_HOME/server/*PROFILE*/conf/login-config.xml** file.

```
<!--
    A template configuration for the JBossWS security domain.
    This defaults to the UsersRolesLoginModule the same as other and
should be
    changed to a stronger authentication mechanism as required.
-->
<application-policy name="JBossWS">
  <authentication>
    <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
      <module-option name="usersProperties">props/jbossws-
users.properties</module-option>
      <module-option name="rolesProperties">props/jbossws-
roles.properties</module-option>
      <module-option
name="unauthenticatedIdentity">anonymous</module-option>
    </login-module>
  </authentication>
</application-policy>
```



NOTE

The default **UsersRolesLoginModule** should be changed to another login module that offers security suitable for your enterprise deployment. Refer to the *JBoss Security Guide* for more information about the available login modules, and how you can create your own custom login module.

A web service client can use the **javax.xml.ws.BindingProvider** interface to set the username and password combination.

Example 11.1. BindingProvider Configuration

```
URL wsdlURL = new File("resources/jaxws/samples/context/WEB-
INF/wsdl/TestEndpoint.wsdl").toURL();
QName qname = new QName("http://org.jboss.ws/jaxws/context",
"TestEndpointService");
Service service = Service.create(wsdlURL, qname);
port = (TestEndpoint)service.getPort(TestEndpoint.class);

BindingProvider bp = (BindingProvider)port;
bp.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "jsmith");
bp.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY,
"PaSSw0rd");
```

HTTP Basic Authentication

You can enable HTTP Basic Authentication by using the @WebContext annotation on the bean class, or by appending an <auth-method> element to the **JBOSS_HOME/server/*PROFILE*/deploy/jbossws.sar/jboss-management.war/WEB-**

INF/jboss-web.xml <login-config> element.

Example 11.2. @WebContext HTTP Basic Authentication

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
@WebContext(contextRoot="/my-cxt", urlPattern="/*", authMethod="BASIC",
transportGuarantee="NONE", secureWSDLAccess=false)
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

Example 11.3. jboss-web.xml HTTP Basic Authentication

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Test Realm</realm-name>
</login-config>
```

11.5. FURTHER INFORMATION

11.5.1. Samples

The JBossWS-CXF source distribution comes with some samples using X.509 certificate signature and encryption as well as Username Token Profile. You can find them in package org.jboss.test.ws.jaxws.samples.wsse

11.5.2. Username/password configuration

When using the Username Token Profile, the username and password are provided and verified through two callback handlers that you need to provide. As the keystore password callback handler, they need to implement javax.security.auth.callback.CallbackHandler; they are configured in **jbossws-cxf.xml** (or programmatically) through the **passwordCallbackClass** attribute.

11.5.3. Crypto Algorithms

When requiring encryption, you might need to install an additional JCE provider supporting the crypto algorithms Apache CXF uses. This usually means the Bouncy Castle provider needs to be configured in your Java Runtime Environment (JRE). Please refer to the Installing the BouncyCastle JCE provider section of the *Administration and Configuration guide* for further information about this.

CHAPTER 12. SOAP MESSAGE LOGGING

The **cxf-extension-jbossws.xml** file contains the JBossWS extensions to the Apache CXF stack. You need to manually add this file and link it in the **cxf.extensions** file. In **cxf-extension-jbossws-xml** you need to enable:

```
<cxf:bus>
  <cxf:inInterceptors>
    <ref bean="logInbound"/>
  </cxf:inInterceptors>
  <cxf:outInterceptors>
    <ref bean="logOutbound"/>
  </cxf:outInterceptors>
  <cxf:inFaultInterceptors>
    <ref bean="logOutbound"/>
  </cxf:inFaultInterceptors>
</cxf:bus>
```

Once you've uncommented the **cxf-extension-jbossws.xml** contents, you need to re-pack the jar or zip. Alternatively, Apache CXF offers multiple ways of configuring SOAP message logging; for programmatic configuration, the below annotations can be used on either the SEI or the SEI implementation class. If placed on the SEI, they activate logging both for client and server; if on the SEI implementation class, they are relevant just for server-side logging.

```
@javax.jws.WebService(portName = "MyWebServicePort", serviceName =
"MyWebService", ...)
@Features(features = "org.apache.cxf.feature.LoggingFeature")
public class MyWebServicePortTypeImpl implements MyWebServicePortType {
```

Or equivalent:

```
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.interceptor.OutInterceptors;

@javax.jws.WebService(portName = "WebServicePort", serviceName =
"WebServiceService", ...)
@InInterceptors(interceptors =
"org.apache.cxf.interceptor.LoggingInInterceptor")
@OutInterceptors(interceptors =
"org.apache.cxf.interceptor.LoggingOutInterceptor")
public class WebServicePortTypeImpl implements WebServicePortType {
```

For programmatic client-side logging, the following code snippet can be used as an example:

```
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;

public class WSClient {
    public static void main (String[] args) {
        MyService ws = new MyService();
```

```
        MyPortType port = ws.getPort();

        Client client = ClientProxy.getClient(port);
        client.getInInterceptors().add(new LoggingInInterceptor());
        client.getOutInterceptors().add(new LoggingOutInterceptor());

        // make WS calls...
```

Finally, you can also enable message logging using the Logging feature.

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

    <cxf:bus>
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>
</beans>
```

12.1. DEBUGGING TOOLS

Here is a list of tools that can be used to capture exchanged messages:

Tcpmon

TCPMon allows you to easily view messages as they go back and forth on the wire.

WSMonitor

WSMonitor is another option to Tcpmon with slightly more functionality.

SOAP UI

SOAP UI can also be used for debugging. In addition to viewing messages, it allows you send messages and load test your services. It also has plugins for Eclipse, IDEA and NetBeans.

Wireshark

Wireshark, a network packet analyzer, is useful for following the routing of SOAP messages. It can also help when you are getting an HTML error message from the server that your CXF client cannot normally process, by allowing you to see the non-SOAP error message.

APPENDIX A. REVISION HISTORY

Revision 5.1.0-110.33.400	2013-10-31	Rüdiger Landmann

Rebuild with publican 4.0.0

Revision 5.1.0-110.33	July 24 2012	Ruediger Landmann

Rebuild for Publican 3.0

Revision 5.1.0-103	Fri Dec 17 2010	Jared Morgan

JBOSSCC-71 - Added Section 11.4 Authentication to the guide to meet CC guidelines.