JBoss AS 6.0 WebServices Guide

WebServices with JBoss Application Server 6

by Alessio Soldano, Richard Opalka, and Jim Ma

Part I. WebServices Overview

JBossWS-WebServices

The Internet features a lot of pages about web services. They describe what web services are, how they work, which kind of technology is most suitable for their development and so on. This page's aim is not to provide another web service definition. We will instead highlight some key concepts about Web services and what they're useful for right now.

1.1. What is a web service?

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

From *W3C Web Services Architecture* *[1]* [http://www.w3.org/TR/2004/NOTE-ws-arch-20040211]

Technical details will be later explained in the *documentation*. What comes out is that web services provide a standard means of interoperating between different software applications. Each of these applications may run on a variety of platforms and/or frameworks providing a set of functionalities. The main concern is about interoperability between services.

- A service provider publishes a *service contract* that exposes the public functions (operations) it is able to perform and thus service consumers can use.
- Both service providers and service consumers features concrete softwares that *send and receive messages* according to the informations contained in the service contract they agreed before the communication.
- Basic Web services specifications define the standard way of *publishing a service contract* and *communicating*.
- Web services stacks (like *JBossWS*) conform to these specifications providing software layers to developers who want to either implement a service provider or service consumer. This way they almost only need to develop their own business logic in their preferred way, without dealing with the low-level details of message exchanges and so on.

1.2. Who needs web services?

Enterprise systems communication may benefit from a wise adoption of WS technologies. Exposing well designed contracts allows developers to extract an abstract view of their service capabilities. Considering the standardized way contracts are written, this definitely helps communication with third-party systems and eventually support business-to-business integration. No more agreement required on vendor specific implementation details, home-brew communication protocol or custom per-customer settings. Everything is clear and standardized in the contract the provider and consumer agree on. Of course this also reduces the dependencies between implementations allowing other consumers to easily use the provided service without major changes.

Enterprise system may benefit from web service technologies also for internal heterogenous subsystems communication. As a matter of fact their interoperability boosts service reuse and composition. No more need to rewrite whole functionalities only because they were developed by another enterprise department using another software language.

1.3. Service Oriented Architecture (SOA)

In case you think you already heard something like this... yes, those in previous paragraph are some of the principles *Service Oriented Architecture* [http://en.wikipedia.org/wiki/Service-oriented_architecture] is based on.

Transforming an enterprise business to Service Oriented Architecture includes obtaining standardized service contract, service reusability, service abstraction, service loose coupling, service composability and so on.

Of course SOA is an architectural model agnostic to technology platforms and every enterprise can pursue the strategic goals associated with service-oriented computing using different technologies. However in the current marketplace, Web Services are probably the technology platform that better suits SOA principles and are most used to get to this architecture.

1.4. What web services are not...

Needless to say that web services are not the solution for every software system communication.

Nowadays they are meant to be used for loosely-coupled coarse-grained communication, for message (document) exchange. Moreover during the last years a lot of specifications (*WS-** [http://community.jboss.org/docs/DOC-13554#Future_of_Web_Services]) were discussed and finally approved to standardize ws-related advanced aspects including reliable messaging, message-level security, cross-service transactions, etc. Finally web service specifications also include notion of registries to collect service contract references, to easily discover service implementations, etc.

This all means that the web services technology platform suits complex enterprise communication and is not simply the latest way of doing remote procedure calls.

JBossWS-Fromconceptstotechnology

This pages is meant to be something like a bridge from the very high level web service concepts highlighted *here* and the most important specifications the web service technology platform is based on.

2.1. Service contracts

Contracts carry technical constraints and requirements of the exposed service as well as information about data to be exchange to interact with the service. They comprise technical descriptions and optional non-technical documents. The latter might include human readable description of the service and the business process it is part of as well as service level agreement / quality of provided service information.

2.1.1. Technical description

Service description is mainly provided using the standard *Web Service Description Language (WSDL)* [http://www.w3.org/TR/wsdl]. Practically speaking this means one or more XML files containing information including the service location (*endpoint address*), the service functionalities (*operations*), the input/output messages involved in the communication and the business data structure. The latter is basically one or more *XML Schema definition* [http://www.w3.org/TR/xmlschema-0/]. Moreover recent specifications (like *WS-Policy* [http://schemas.xmlsoap.org/ws/2004/09/policy/]) allow for more advanced service capabilities to be stated in the contract through WSDL extensions.

Web service stacks like JBossWS usually have tools to both generate and consume technical contracts. This helps ensuring also from a practical point of view that owners of service producer (*server*) and consumer (*client*) only need contracts to establish the communication.

2.1.2. Contract delivery process

One of the main concerns about service contracts is the way they're obtained.

As previously said, tools allow developers to automatically generate WSDL contract files given their service implementation. Advantages and disadvantage of this delivery process include:

- Developers do not have to deal with contracts by hand thus deep knowledge of WSDL and XML is not required.
- Less effort and time required for services to be developed and go live to production environment.
- Contracts usually need frequent maintenance, refactoring and versioning.

Developers may write contracts first instead. This usually implies an initial collaboration of architects and business analysts to define a conceptual service design together.

- Services with contracts obtained this way may easily cooperate in a service oriented architecture
- More effort and time required for web service project start-up
- Deep knowledge of WSDL and related technology required
- Contracts tend to have longer lifespans and usually require less maintenance.

2.2. Message exchange

As stated by the *W3C definition* [http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#whatis], the communication between web services is standardized by the *SOAP* [http://www.w3.org/TR/soap/] specification. This means XML messages flow from the provider and consumer endpoints.

Messages' content is described in the wsdl contract. The WSDL file also states the transport protocol to be used for the transmission; the most common one is of course HTTP, however JMS, SMTP and other ones are allowed.

2.3. Registries

TODO

2.4. Future of Web Services

The above mentioned specifications are quite common nowadays in the IT industry and many enterprise have been using them since years.

However a real added value to the web service platform is coming from a lot of recent additional specifications. These cover features that are really relevant to deliver mission critical enterprise services. For example some of the most important agreements major vendors came to are those on security (*WS-Security* [http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss]) and reliable messaging (*WS-Reliable Messaging* [http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-rx]).

Unfortunately the web service platform is still being defined and many other specifications have not been implemented by all vendors yet. It is nevertheless important to know from a web service beginner point of view that many advanced features are actually supported and thus make possible to cope with many real world enterprise level issues. Moreover the platform is being continuously enriched and standardized.

2.5. References

Further knowledge is of course required to better understand the web service technology platform. This however goes beyond the aim of this web service introduction. The highlighted concepts and references above should nevertheless allow developers with no previous exposure to web service technology to go through the core of *JBossWS documentation* [http://community.jboss.org/docs/DOC-13504].

A rich list of specifications and articles can be found *here* and should be used to acquire deeper knowledge. Moreover the whole documentation refers to authoritative third-party documentation and official specifications whenever required.

Part II. Main Documentation

JBoss Application Server 6.0 comes with JBossWS-CXF already installed. JBossWS-CXF is basically the JBoss Web Service integration with Apache CXF stack. This way additional JBoss features and customizations are added on top of already existing Apache CXF functionalities. In particular the integration provides technologies for allowing the application server to support JavaEE requirements in terms of Web Services functionalities.

Below you find the essential documentation on JBossWS - CXF coming with JBoss AS 6. That covers a quick start, a full user guide and tooling.

JBossWS-QuickStart

3.1. Right on'

JBossWS uses the JBoss application server as its target container. The following examples focus on web service deployments that leverage EJB3 service implementations and the JAX-WS programming models. For further information on POJO service implementations and advanced topics you need consult the user guide.

In the following sections we will explore the samples that ship with the JBossWS distribution. They provide a build structure based on Ant to get you started quickly.

3.2. Developing web service implementations

JAX-WS does leverage *JDK 5 annotations* [http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html] in order to express web service meta data on Java components and to describe the mapping between Java data types and XML. When developing web service implementations you need to decide whether you are going start with an abstract contract (*WSDL* [http://www.w3.org/TR/wsdl]) or a Java component.

If you are in charge to provide the service implementation, then you are probably going to start with the implementation and derive the abstract contract from it. You are probably not even getting in touch with the WSDL unless you hand it to 3rd party clients. For this reason we are going to look at a service implementation that leverages *JSR-181 annotations* [http://jcp.org/en/jsr/detail?id=181].

Note

Note

Even though detailed knowledge of web service meta data is not required, it will definitely help if you make yourself familiar with it. For further information see

- *Web service meta data (JSR-181)* [http://jcp.org/en/jsr/detail?id=181]
- *Java API for XML binding (JAXB)* [http://java.sun.com/webservices/jaxb/]

When starting from Java you must provide the service implementation. A valid endpoint implementation class must meet the following requirements:

- It *must* carry a `javax.jws.WebService` annotation (see JSR 181)
- All method parameters and return types *must* be compatible with the JAXB 2.0

Let's look at a sample EJB3 component that is going to be exposed as a web service. (This is based on the Retail example).

Don't be confused with the EJB3 annotation `@Stateless`. We concentrate on the `@WebService` annotation for now.

The service implementation class

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

@Stateless                                                                    (1)
@WebService(                                                                  (2)
   name="ProfileMgmt",
   targetNamespace = "http://org.jboss.ws/samples/retail/profile",
   serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)               (3)
public class ProfileMgmtBean {

   @WebMethod                                                                 (4)
   public DiscountResponse getCustomerDiscount(DiscountRequest request) {
      return new DiscountResponse(request.getCustomer(), 10.00);
   }
}
```

1. We are using a stateless session bean implementation

2. Exposed a web service with an explicit namespace

3. It's a doc/lit bare endpoint

4. And offers an 'getCustomerDiscount' operation

What about the payload?

The method parameters and return values are going to represent our XML payload and thus require being compatible with *JAXB2* [http://java.sun.com/webservices/jaxb/]. Actually you wouldn't need any JAXB annotations for this particular example, because JAXB relies on meaningful defaults. For the sake of documentation we put the more important ones here.

Take a look at the request parameter:

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(                                                  (1)
  name = "discountRequest",
  namespace="http://org.jboss.ws/samples/retail/profile",
  propOrder = { "customer" }
)
public class DiscountRequest {

   protected Customer customer;

   public DiscountRequest() {
   }

   public DiscountRequest(Customer customer) {
      this.customer = customer;
   }

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer value) {
      this.customer = value;
   }

}
```

1. In this case we use `@XmlType` to specify an XML complex type name and override the namespace.

3.2.1. Deploying service implementations

Service deployment basically depends on the implementation type. As you may already know web services can be implemented as EJB3 components or plain old Java objects. This quick start leverages EJB3 components in all examples, thats why we are going to look at this case in the next sections.

EJB3 services

Simply wrap up the service implementation class, the endpoint interface and any custom data types in a JAR and drop them in the `deploy` directory. No additional deployment descriptors required. Any meta data required for the deployment of the actual web service is taken from the annotations provided on the implementation class and the service endpoint interface. JBossWS will intercept that EJB3 deployment (the bean will also be there) and create an HTTP endpoint at deploy-time:

The JAR package structure

```
jar -tf jaxws-samples-retail.jar

org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

Note

Note

If the deployment was successful you should be able to see your endpoint at http://localhost:8080/jbossws/services

3.3. Consuming web services

When creating web service clients you would usually start from the WSDL. JBossWS ships with a set of tools to generate the required JAX-WS artefacts to build client implementations. In the following section we will look at the most basic usage patterns. For a more detailed introductoin to web service client please consult the *user guide* [http://community.jboss.org/docs/DOC-13972].

Using wsconsume

The *wsconsume tool* is used to consume the abstract contract (WSDL) and produce annotated Java classes (and optionally sources) that define it. We are going to start with the WSDL from our retail example (*ProfileMgmtService.wsdl*). For a detailed tool reference you need to consult the user guide.

```
wsconsume is a command line tool that generates
portable JAX-WS artifacts from a WSDL file.

usage: org.jboss.ws.tools.jaxws.command.wsconsume [options] <wsdl-url>

options:
    -h, --help                  Show this help message
    -b, --binding=<file>        One or more JAX-WS or JAXB binding files
    -k, --keep                  Keep/Generate Java source
    -c  --catalog=<file>        Oasis XML Catalog file for entity resolution
    -p  --package=<name>        The target package for generated source
    -w  --wsdlLocation=<loc>    Value to use for @WebService.wsdlLocation
    -o, --output=<directory>    The directory to put generated artifacts
    -s, --source=<directory>    The directory to put Java source
    -q, --quiet                 Be somewhat more quiet
    -t, --show-traces           Show full exception stack traces
```

Let's try it on our retail sample:

```
~./wsconsume.sh -k
-p org.jboss.test.ws.jaxws.samples.retail.profile ProfileMgmtService.wsdl
 (1)

org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.java
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.java
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.java
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.java
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.java
```

1. As you can see we did use the `-p` switch to specify the package name of the generated sources.

The generated artifacts explained

File	Purpose
ProfileMgmt.java	Service Endpoint Interface
Customer.java	Custom data type
Discount*.java	Custom data type
ObjectFactory.java	JAXB XML Registry

File	Purpose
package-info.java	Holder for JAXB package annotations
ProfileMgmtService.java	Service factory

Basically `wsconsume` generates all custom data types (JAXB annotated classes), the service endpoint interface and a service factory class. We will look at how these artifacts can be used the build web service client implementations in the next section.

Constructing a service stub

Web service clients make use of a service stubs that hide the details of a remote web service invocation. To a client application a WS invocation just looks like an invocation of any other business component. In this case the service endpoint interface acts as the business interface. JAX-WS does use a service factory class to construct this as particular service stub:

```
import javax.xml.ws.Service;

Service service = Service.create(                                 (1)
  new URL("http://example.org/service?wsdl"),
  new QName("MyService")
);
ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);     (2)

// do something with the service stub here...                    (3)
```

1. Create a service factory using the WSDL location and the service name
2. Use the tool created service endpoint interface to build the service stub
3. Use the stub like any other business interface

Note

Note

*The creation of the service stub is quite costly. You should take care that it gets reused by your application code (However **it's not thread safe**). Within a EE5 environment you might want to investigate the `@WebServiceRef` functionality.*

3.4. Appendix

3.4.1. ProfileMgmtService.wsdl

```
<definitions
    name='ProfileMgmtService'
    targetNamespace='http://org.jboss.ws/samples/retail/profile'
    xmlns='http://schemas.xmlsoap.org/wsdl/'
    xmlns:ns1='http://org.jboss.ws/samples/retail'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:tns='http://org.jboss.ws/samples/retail/profile'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

   <types>

      <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
                 version='1.0' xmlns:xs='http://www.w3.org/2001/XMLSchema'>
         <xs:complexType name='customer'>
            <xs:sequence>
               <xs:element minOccurs='0' name='creditCardDetails'
 type='xs:string'/>
               <xs:element minOccurs='0' name='firstName' type='xs:string'/>
               <xs:element minOccurs='0' name='lastName' type='xs:string'/>
            </xs:sequence>
         </xs:complexType>
      </xs:schema>

      <xs:schema
          targetNamespace='http://org.jboss.ws/samples/retail/profile'
          version='1.0'
          xmlns:ns1='http://org.jboss.ws/samples/retail'
          xmlns:tns='http://org.jboss.ws/samples/retail/profile'
          xmlns:xs='http://www.w3.org/2001/XMLSchema'>

         <xs:import namespace='http://org.jboss.ws/samples/retail'/>
         <xs:element name='getCustomerDiscount'
                     nillable='true' type='tns:discountRequest'/>
         <xs:element name='getCustomerDiscountResponse'
                     nillable='true' type='tns:discountResponse'/>
         <xs:complexType name='discountRequest'>
            <xs:sequence>
               <xs:element minOccurs='0' name='customer'
 type='ns1:customer'/>

            </xs:sequence>
         </xs:complexType>
         <xs:complexType name='discountResponse'>
            <xs:sequence>
```

```
               <xs:element minOccurs='0' name='customer'
 type='ns1:customer'/>
               <xs:element name='discount' type='xs:double'/>
            </xs:sequence>
         </xs:complexType>
      </xs:schema>

   </types>

   <message name='ProfileMgmt_getCustomerDiscount'>
      <part element='tns:getCustomerDiscount' name='getCustomerDiscount'/>
   </message>
   <message name='ProfileMgmt_getCustomerDiscountResponse'>
      <part element='tns:getCustomerDiscountResponse'
            name='getCustomerDiscountResponse'/>
   </message>
   <portType name='ProfileMgmt'>
      <operation name='getCustomerDiscount'
                 parameterOrder='getCustomerDiscount'>

         <input message='tns:ProfileMgmt_getCustomerDiscount'/>
         <output message='tns:ProfileMgmt_getCustomerDiscountResponse'/>
      </operation>
   </portType>
   <binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
      <soap:binding style='document'
                    transport='http://schemas.xmlsoap.org/soap/http'/>
      <operation name='getCustomerDiscount'>
         <soap:operation soapAction=''/>
         <input>

            <soap:body use='literal'/>
         </input>
         <output>
            <soap:body use='literal'/>
         </output>
      </operation>
   </binding>
   <service name='ProfileMgmtService'>
      <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

         <soap:address

 location='http://<HOST>:<PORT>/jaxws-samples-retail/ProfileMgmtBean'/>
      </port>
   </service>
</definitions>
```

JBossWS-UserGuide

4.1. Common User Guide

Here below is the documentation that applies to all the JBossWS stack versions, hence including JBossWS-CXF. This includes basic JAX-WS usage as well as references to common additional functionalities the JBossWS Web Service Framework provides on top of the CXF stack.

4.1.1. Web Service Concepts

4.1.1.1. Document/Literal

With document style web services two business partners agree on the exchange of complex business documents that are well defined in XML schema. For example, one party sends a document describing a purchase order, the other responds (immediately or later) with a document that describes the status of the purchase order. No need to agree on such low level details as operation names and their associated parameters.

The payload of the SOAP message is an XML document that can be validated against XML schema.

Document is defined by the style attribute on the SOAP binding.

```
<binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
 <soap:binding style='document'
transport='http://schemas.xmlsoap.org/soap/http'/>
 <operation name='concat'>
  <soap:operation soapAction=''/>
  <input>
   <soap:body use='literal'/>
  </input>
  <output>
   <soap:body use='literal'/>
  </output>
 </operation></binding>
```

With document style web services the payload of every message is defined by a complex type in XML schema.

```
  <complexType name='concatType'>
   <sequence>
    <element name='String_1' nillable='true' type='string'/>
    <element name='long_1' type='long'/>
```

```
      </sequence>
     </complexType>
     <element name='concat' type='tns:concatType'/>
```

Therefore, message parts **must** refer to an **element** from the schema.

```
   <message name='EndpointInterface_concat'>
    <part name='parameters' element='tns:concat'/></message>
```

The following message definition **is invalid**.

```
   <message name='EndpointInterface_concat'>
    <part name='parameters' type='tns:concatType'/></message>
```

4.1.1.1.1. Document/Literal (Bare)

Bare is an implementation detail from the Java domain. Neither in the abstract contract (i.e. wsdl+schema) nor at the SOAP message level is a bare endpoint recognizable.

A bare endpoint or client uses a Java bean that represents the entire document payload.

```
@WebService
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public class DocBareServiceImpl
{
   @WebMethod
   public SubmitBareResponse submitPO(SubmitBareRequest poRequest)
   {
      ...
   }
}
```

The trick is that the Java beans representing the payload contain JAXB annotations that define how the payload is represented on the wire.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SubmitBareRequest",
 namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/", propOrder
 = { "product" })
@XmlRootElement(namespace="http://
soapbinding.samples.jaxws.ws.test.jboss.org/", name = "SubmitPO")
public class SubmitBareRequest
{

 @XmlElement(namespace="http://soapbinding.samples.jaxws.ws.test.jboss.org/
",  required = true)
   private String product;
```

```
    ...
}
```

4.1.1.1.2. Document/Literal (Wrapped)

Wrapped is an implementation detail from the Java domain. Neither in the abstract contract (i.e. wsdl+schema) nor at the SOAP message level is a wrapped endpoint recognizable.

A wrapped endpoint or client uses the individual document payload properties. Wrapped is the default and does not have to be declared explicitly.

```
@WebService
public class DocWrappedServiceImpl
{
   @WebMethod
   @RequestWrapper (className="org.somepackage.SubmitPO")
   @ResponseWrapper (className="org.somepackage.SubmitPOResponse")
   public String submitPO(String product, int quantity)
   {
      ...
   }
}
```

Note, that with JBossWS the request/response wrapper annotations are **not required**, they will be generated on demand using sensible defaults.

4.1.1.2. RPC/Literal

With RPC there is a wrapper element that names the endpoint operation. Child elements of the RPC parent are the individual parameters.

The SOAP body is constructed based on some simple rules:

- The port type operation name defines the endpoint method name
- Message parts are endpoint method parameters

RPC is defined by the style attribute on the SOAP binding.

```
 <binding name='EndpointInterfaceBinding' type='tns:EndpointInterface'>
  <soap:binding style='rpc'
 transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='echo'>
   <soap:operation soapAction=''/>
   <input>
```

```
    <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
 use='literal'/>
   </input>
   <output>
    <soap:body namespace='http://org.jboss.ws/samples/jsr181pojo'
 use='literal'/>
   </output>
  </operation></binding>
```

With rpc style web services the portType names the operation (i.e. the java method on the endpoint)

```
 <portType name='EndpointInterface'>
  <operation name='echo' parameterOrder='String_1'>
   <input message='tns:EndpointInterface_echo'/>
   <output message='tns:EndpointInterface_echoResponse'/>
  </operation></portType>
```

Operation parameters are defined by individual message parts.

```
 <message name='EndpointInterface_echo'>
  <part name='String_1' type='xsd:string'/></message><message
 name='EndpointInterface_echoResponse'>
  <part name='result' type='xsd:string'/></message>
```

Note, there is no complex type in XML schema that could validate the entire SOAP message payload.

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
   @WebMethod
   @WebResult(name="result")
   public String echo(@WebParam(name="String_1") String input)
   {
      ...
   }
}
```

The element names of RPC parameters/return values may be defined using the JAX-WS *WebParam* and *WebResult* annotations respectively.

4.1.1.3. RPC/Encoded

SOAP encodeding style is defined by the infamous *chapter 5* [http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383512] of the *SOAP-1.1* [http://www.w3.org/TR/2000/NOTE-SOAP-

20000508] specification. **It has inherent interoperability issues** that cannot be fixed. The *Basic Profile-1.0* [http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html] prohibits this encoding style in *4.1.7 SOAP encodingStyle Attribute* [http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html#refinement16448072].

JBossWS doesn't support rpc/encoded anymore.

4.1.2. Web Service Endpoints

JAX-WS simplifies the development model for a web service endpoint a great deal. In short, an endpoint implementation bean is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes the abstract contract (i.e. wsdl+schema) for client consumption. All marshalling/unmarshalling is delegated to JAXB.

4.1.2.1. Plain old Java Object (POJO)

Let's take a look at simple POJO endpoint implementation. All endpoint associated metadata is provided via JSR-181 annotations

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
   @WebMethod
   public String echo(String input)
   {
      ...
   }
}
```

The endpoint as a web application

A JAX-WS java service endpoint (JSE) is deployed as a web application.

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>

 <servlet-class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</
servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Packaging the endpoint

A JSR-181 java service endpoint (JSE) is packaged as a web application in a *.war file.

```
    <war warfile="${build.dir}/libs/jbossws-samples-jsr181pojo.war"
 webxml="${build.resources.dir}/samples/jsr181pojo/WEB-INF/web.xml">
      <classes dir="${build.dir}/classes">
        <include
 name="org/jboss/test/ws/samples/jsr181pojo/JSEBean01.class"/>
      </classes>
    </war>
```

Note, only the endpoint implementation bean and web.xml are required.

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the service endpoint manager. This is also where you find the links to the generated wsdl.

```
   http://yourhost:8080/jbossws/services
```

Note, it is also possible to generate the abstract contract off line using jbossw tools. For details of that please see *Bottom-Up (Java to WSDL)*

4.1.2.2. EJB3 Stateless Session Bean (SLSB)

The JAX-WS programming model support the same set of annotations on EJB3 stateless session beans as on *Plain old Java Object (POJO)* endpoints. EJB-2.1 endpoints are supported using the JAX-RPC progamming model (with JBossWS-Native only).

```
    @Stateless
    @Remote(EJB3RemoteInterface.class)
    @RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

    @WebService
    @SOAPBinding(style = SOAPBinding.Style.RPC)
    public class EJB3Bean01 implements EJB3RemoteInterface
    {
       @WebMethod
       public String echo(String input)
       {
          ...
       }
    }
```

Above you see an EJB-3.0 stateless session bean that exposes one method both on the remote interface and on and as an endpoint operation.

Packaging the endpoint

A JSR-181 EJB service endpoint is packaged as an ordinary ejb deployment.

```
    <jar jarfile="${build.dir}/libs/jbossws-samples-jsr181ejb.jar">
      <fileset dir="${build.dir}/classes">
        <include
 name="org/jboss/test/ws/samples/jsr181ejb/EJB3Bean01.class"/>
        <include
 name="org/jboss/test/ws/samples/jsr181ejb/EJB3RemoteInterface.class"/>
      </fileset>
    </jar>
```

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the service endpoint manager. This is also where you find the links to the generated wsdl.

```
   http://yourhost:8080/jbossws/services
```

Note, it is also possible to generate the abstract contract off line using jbossw tools. For details of that please see *Bottom-Up (Java to WSDL)*

4.1.2.3. Endpoint Provider

JAX-WS services typically implement a native Java service endpoint interface (SEI), perhaps mapped from a WSDL port type, either directly or via the use of annotations.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The Provider interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML message level.

A Provider based service instances invoke method is called for each message received for the service.

```
   @WebServiceProvider(wsdlLocation = "WEB-INF/wsdl/Provider.wsdl")
   @ServiceMode(value = Service.Mode.PAYLOAD)
   public class ProviderBeanPayload implements Provider<Source>
```

```
    {
       public Source invoke(Source req)
       {
          // Access the entire request PAYLOAD and return the response
 PAYLOAD
       }
    }
```

Note, Service.Mode.PAYLOAD is the default and does not have to be declared explicitly. You can also use Service.Mode.MESSAGE to access the entire SOAP message (i.e. with MESSAGE the Provider can also see SOAP Headers)

The abstract contract for a provider endpoint cannot be derived/generated automatically. Therefore it is necessary to specify the wsdlLocation with the @WebServiceProvider annotation.

4.1.2.4. WebServiceContext

The WebServiceContext is treated as an injectable resource that can be set at the time an endpoint is initialized. The WebServiceContext object will then use thread-local information to return the correct information regardless of how many threads are concurrently being used to serve requests addressed to the same endpoint object.

```
    @WebService
    public class EndpointJSE
    {
       @Resource
       WebServiceContext wsCtx;

       @WebMethod
       public String testGetMessageContext()
       {
          SOAPMessageContext jaxwsContext =
 (SOAPMessageContext)wsCtx.getMessageContext();
          return jaxwsContext != null ? "pass" : "fail";
       }

       @WebMethod
       public String testGetUserPrincipal()
       {
          Principal principal = wsCtx.getUserPrincipal();
          return principal.getName();
       }

       @WebMethod
       public boolean testIsUserInRole(String role)
       {
          return wsCtx.isUserInRole(role);
```

```
      }
   }
```

4.1.3. Web Service Clients

4.1.3.1. Service

`Service` is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at a particular endpoint address.

For most clients, you will start with a set of stubs generated from the WSDL. One of these will be the service, and you will create objects of that class in order to work with the service (see "static case" below).

4.1.3.1.1. Service Usage

Static case

Most clients will start with a WSDL file, and generate some stubs using jbossws tools like *wsconsume*. This usually gives a mass of files, one of which is the top of the tree. This is the service implementation class.

The generated implementation class can be recognised as it will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a java.net.URL) and the service name (a javax.xml.namespace.QName) respectively.

Usually you will use the no-argument constructor. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the WebServiceClient annotation that decorates the generated class.

The following code snippet shows the generated constructors from the generated class:

```
// Generated Service Class

@WebServiceClient(name="StockQuoteService",
 targetNamespace="http://example.com/stocks",
 wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
   public StockQuoteService()
   {
      super(new URL("http://example.com/stocks.wsdl"), new
 QName("http://example.com/stocks", "StockQuoteService"));
   }

   public StockQuoteService(String wsdlLocation, QName serviceName)
   {
      super(wsdlLocation, serviceName);
   }

   ...
}
```

Section *Dynamic Proxy* explains how to obtain a port from the service and how to invoke an operation on the port. If you need to work with the XML payload directly or with the XML representation of the entire SOAP message, have a look at *Dispatch*.

Dynamic case

In the dynamic case, when nothing is generated, a web service client uses `Service.create` to create Service instances, the following code illustrates this process.

```
    URL wsdlLocation = new URL("http://example.org/my.wsdl");
    QName serviceName = new QName("http://example.org/sample", "MyService");
    Service service = Service.create(wsdlLocation, serviceName);
```

4.1.3.1.2. Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-WS runtime system. *Handler Framework* describes the handler framework in detail. A Service instance provides access to a HandlerResolver via a pair of getHandlerResolver/setHandlerResolver methods that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a Service instance is used to create a proxy or a Dispatch instance then the handler resolver currently registered with the service is used to create the required handler chain. Subsequent changes to the handler resolver configured for a Service instance do not affect the handlers on previously created proxies, or Dispatch instances.

4.1.3.1.3. Executor

Service instances can be configured with a java.util.concurrent.Executor. The executor will then be used to invoke any asynchronous callbacks requested by the application. The setExecutor and getExecutor methods of Service can be used to modify and retrieve the executor configured for a service.

4.1.3.2. Dynamic Proxy

You can create an instance of a client proxy using one of getPort methods on the *Service*.

```
    /**
     * The getPort method returns a proxy. A service client
     * uses this proxy to invoke operations on the target
     * service endpoint. The <code>serviceEndpointInterface</code>
     * specifies the service endpoint interface that is supported by
     * the created dynamic proxy instance.
     **/
    public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
    {
       ...
    }

    /**
     * The getPort method returns a proxy. The parameter
     * <code>serviceEndpointInterface</code> specifies the service
     * endpoint interface that is supported by the returned proxy.
     * In the implementation of this method, the JAX-WS
     * runtime system takes the responsibility of selecting a protocol
     * binding (and a port) and configuring the proxy accordingly.
     * The returned proxy should not be reconfigured by the client.
     *
     **/
    public <T> T getPort(Class<T> serviceEndpointInterface)
    {
       ...
    }
```

The service endpoint interface (SEI) is usually generated using tools. For details see *Top Down (WSDL to Java)*

A generated static *Service* [http://community.jboss.org/Service] usually also offers typed methods to get ports. These methods also return dynamic proxies that implement the SEI.

```
@WebServiceClient(name = "TestEndpointService", targetNamespace =
 "http://org.jboss.ws/wsref",
   wsdlLocation =
 "http://localhost.localdomain:8080/jaxws-samples-webserviceref?wsdl")

public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }
    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT,
 TestEndpoint.class);
    }
}
```

4.1.3.3. WebServiceRef

The WebServiceRef annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the javax.annotation.Resource annotation in JSR-250.

There are two uses to the WebServiceRef annotation:

1. To define a reference whose type is a generated service class. In this case, the type and value element will both refer to the generated service class type. Moreover, if the reference type can be inferred by the field/method declaration the annotation is applied to, the type and value elements MAY have the default value (Object.class, that is). If the type cannot be inferred, then at least the type element MUST be present with a non-default value.
2. To define a reference whose type is a SEI. In this case, the type element MAY be present with its default value if the type of the reference can be inferred from the annotated field/method declaration, but the value element MUST always be present and refer to a generated service class type (a subtype of javax.xml.ws.Service). The wsdlLocation element, if present, overrides theWSDL location information specified in the WebService annotation of the referenced generated service class.

```
public class EJB3Client implements EJB3Remote
{
   @WebServiceRef
   public TestEndpointService service4;

   @WebServiceRef
   public TestEndpoint port3;
```

WebServiceRef Customization

Starting from jboss-5.0.x we offer a number of overrides and extensions to the WebServiceRef annotation. These include

- define the port that should be used to resolve a container-managed port

- define default Stub property settings for Stub objects
- define the URL of a final WSDL document to be used

Example:

```
  <service-ref>
   <service-ref-name>OrganizationService</service-ref-name>

 <wsdl-override>file:/wsdlRepository/organization-service.wsdl</wsdl-
override>
  </service-ref>

  <service-ref>
   <service-ref-name>OrganizationService</service-ref-name>
   <config-name>Secure Client Config</config-name>
   <config-file>META-INF/jbossws-client-config.xml</config-file>
   <handler-chain>META-INF/jbossws-client-handlers.xml</handler-chain>
  </service-ref>

  <service-ref>
   <service-ref-name>SecureService</service-ref-name>

 <service-impl-
class>org.jboss.tests.ws.jaxws.webserviceref.SecureEndpointService</service-
impl-class>

 <service-qname>{http://org.jboss.ws/wsref}SecureEndpointService</service-
qname>
    <port-component-ref>

 <service-endpoint-
interface>org.jboss.tests.ws.jaxws.webserviceref.SecureEndpoint</service-
endpoint-interface>
     <port-qname>{http://org.jboss.ws/wsref}SecureEndpointPort</port-qname>
     <stub-property>
      <prop-name>javax.xml.ws.security.auth.username</prop-name>
      <prop-value>kermit</prop-value>
     </stub-property>
     <stub-property>
      <prop-name>javax.xml.ws.security.auth.password</prop-name>
      <prop-value>thefrog</prop-value>
     </stub-property>
   </port-component-ref>
  </service-ref>
```

For details please see **service-ref_5_0.dtd** in the jboss docs directory.

4.1.3.4. Dispatch

XMLWeb Services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The Dispatch interface provides support for this mode of interaction.

Dispatch supports two usage modes, identified by the constants javax.xml.ws.Service.Mode.MESSAGE and javax.xml.ws.Service.Mode.PAYLOAD respectively:

Message In this mode, client applications work directly with protocol-specific message structures. E.g., when used with a SOAP protocol binding, a client application would work directly with a SOAP message.

Message Payload In this mode, client applications work with the payload of messages rather than the messages themselves. E.g., when used with a SOAP protocol binding, a client application would work with the contents of the SOAP Body rather than the SOAP message as a whole.

Dispatch is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. Dispatch is a generic class that supports input and output of messages or message payloads of any type.

```
      Service service = Service.create(wsdlURL, serviceName);
      Dispatch dispatch = service.createDispatch(portName,
 StreamSource.class, Mode.PAYLOAD);

      String payload = "<ns1:ping
 xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
      dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

      payload = "<ns1:feedback
 xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
      Source retObj = (Source)dispatch.invoke(new StreamSource(new
 StringReader(payload)));
```

4.1.3.5. Asynchronous Invocations

The BindingProvider interface represents a component that provides a protocol binding for use by clients, it is implemented by proxies and is extended by the Dispatch interface.

BindingProvider instances may provide asynchronous operation capabilities. When used, asynchronous operation invocations are decoupled from the BindingProvider instance at invocation time such that the response context is not updated when the operation completes. Instead a separate response context is made available using the Response interface.

```
    public void testInvokeAsync() throws Exception
    {
       URL wsdlURL = new URL("http://" + getServerHost() +
  ":8080/jaxws-samples-asynchronous?wsdl");
       QName serviceName = new QName(targetNS, "TestEndpointService");
       Service service = Service.create(wsdlURL, serviceName);
       TestEndpoint port = service.getPort(TestEndpoint.class);

       Response response = port.echoAsync("Async");

       // access future
       String retStr = (String) response.get();
       assertEquals("Async", retStr);
    }
```

4.1.3.6. Oneway Invocations

@Oneway indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method.

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
   private static String feedback;

   @WebMethod
   @Oneway
   public void ping()
   {
      log.info("ping");
      feedback = "ok";
   }

   @WebMethod
   public String feedback()
   {
      log.info("feedback");
      return feedback;
   }
}
```

4.1.3.7. Timeout Configuration

There are two properties to configure the http connection timeout and client receive time out:

```
 public void testConfigureTimeout() throws Exception
  {   //Set timeout until a connection is established
 ((BindingProvider) port).getRequestContext().
```

```
   put("javax.xml.ws.client.connectionTimeout", "6000");
   //Set timeout until the response is received
   ((BindingProvider) port).getRequestContext().
   put("javax.xml.ws.client.receiveTimeout", "1000");
   port.echo("testTimeout");    }
```

4.1.4. Common API

This sections describes concepts that apply equally to *Web Service Endpoints* and *Web Service Clients*.

4.1.4.1. Handler Framework

The handler framework is implemented by a JAX-WS protocol binding in both client and server side runtimes. Proxies, and Dispatch instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

4.1.4.1.1. Logical Handler

Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement javax.xml.ws.handler.LogicalHandler.

4.1.4.1.2. Protocol Handler

Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message. Protocol handlers are handlers that implement any interface derived from javax.xml.ws.handler.Handler except javax.xml.ws.handler.LogicalHandler.

4.1.4.1.3. Service endpoint handlers

On the service endpoint, handlers are defined using the @HandlerChain annotation.

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
```

```
public class SOAPEndpointSourceImpl
{
   ...
}
```

The location of the handler chain file supports 2 formats

1. An absolute java.net.URL in externalForm. (ex: *http://myhandlers.foo.com/handlerfile1.xml*)

2. A relative path from the source file or class file. (ex: bar/handlerfile1.xml)

4.1.4.1.4. Service client handlers

On the client side, handler can be configured using the @HandlerChain annotation on the SEI or dynamically using the API.

```
      Service service = Service.create(wsdlURL, serviceName);
      Endpoint port = (Endpoint)service.getPort(Endpoint.class);

      BindingProvider bindingProvider = (BindingProvider)port;
      List<Handler> handlerChain = new ArrayList<Handler>();
      handlerChain.add(new LogHandler());
      handlerChain.add(new AuthorizationHandler());
      handlerChain.add(new RoutingHandler());
      bindingProvider.getBinding().setHandlerChain(handlerChain); //
 important!
```

4.1.4.2. Message Context

MessageContext is the super interface for all JAX-WS message contexts. It extends Map<String,Object> with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the put method to insert a property in the message context that one or more other handlers in the handler chain may subsequently obtain via the get method.

Properties are scoped as either APPLICATION or HANDLER. All properties are available to all handlers for an instance of an MEP on a particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during the execution of an MEP instance. APPLICATION scoped properties are also made available to client applications (see section 4.2.1) and service endpoint implementations. The defaultscope for a property is HANDLER.

4.1.4.2.1. Logical Message Context

Logical Handlers are passed a message context of type LogicalMessageContext when invoked. LogicalMessageContext extends MessageContext with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of amessage. A protocol binding defines what component of a message are available via a logical message context. The SOAP binding defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers whereas the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

4.1.4.2.2. SOAP Message Context

SOAP handlers are passed a SOAPMessageContext when invoked. SOAPMessageContext extends MessageContext with methods to obtain and modify the SOAP message payload.

4.1.4.3. Fault Handling

An implementation may thow a SOAPFaultException

```
   public void throwSoapFaultException()
   {
      SOAPFactory factory = SOAPFactory.newInstance();
      SOAPFault fault = factory.createFault("this is a fault string!", new
 QName("http://foo", "FooCode"));
      fault.setFaultActor("mr.actor");
      fault.addDetail().addChildElement("test");
      throw new SOAPFaultException(fault);
   }
```

or an application specific user exception

```
   public void throwApplicationException() throws UserException
   {
      throw new UserException("validation", 123, "Some validation error");
   }
```

Note

Note

In case of the latter JBossWS generates the required fault wrapper beans at runtime if they are not part of the deployment

4.1.4.4. JBossWS Extensions

This section describes propriatary JBoss extensions to JAX-WS, that works with all the supported stacks.

4.1.4.4.1. Proprietary Annotations

For the set of standard annotations, please have a look at *JAX-WS_Annotations*.

4.1.4.4.1.1. WebContext

```
/**
 * Provides web context specific meta data to EJB based web service
 endpoints.
 *
 * @author thomas.diesler@jboss.org [mailto:thomas.diesler@jboss.org]
 * @since 26-Apr-2005
 */
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = { ElementType.TYPE })
public @interface WebContext {

   /**
    * The contextRoot element specifies the context root that the web
 service endpoint is deployed to.
    * If it is not specified it will be derived from the deployment short
 name.
    *
    * Applies to server side port components only.
    */
   String contextRoot() default "";

   /**
    * The virtual hosts that the web service endpoint is deployed to.
    *
    * Applies to server side port components only.
    */
   String[] virtualHosts() default {};

   /**
    * Relative path that is appended to the contextRoot to form fully
 qualified
    * endpoint address for the web service endpoint.
    *
    * Applies to server side port components only.
    */
   String urlPattern() default "";

   /**
    * The authMethod is used to configure the authentication mechanism for
 the web service.
    * As a prerequisite to gaining access to any web service which are
 protected by an authorization
```

```
    * constraint, a user must have authenticated using the configured
 mechanism.
    *
    * Legal values for this element are "BASIC", or "CLIENT-CERT".
    */
   String authMethod() default "";

   /**
    * The transportGuarantee specifies that the communication
    * between client and server should be NONE, INTEGRAL, or
    * CONFIDENTIAL. NONE means that the application does not require any
    * transport guarantees. A value of INTEGRAL means that the application
    * requires that the data sent between the client and server be sent in
    * such a way that it can't be changed in transit. CONFIDENTIAL means
    * that the application requires that the data be transmitted in a
    * fashion that prevents other entities from observing the contents of
    * the transmission. In most cases, the presence of the INTEGRAL or
    * CONFIDENTIAL flag will indicate that the use of SSL is required.
    */
   String transportGuarantee() default "";

   /**
    * A secure endpoint does not secure wsdl access by default.
    * Explicitly setting secureWSDLAccess overrides this behaviour.
    *
    * Protect access to WSDL. See
 http://jira.jboss.org/jira/browse/JBWS-723
    */
   boolean secureWSDLAccess() default false;
}
```

4.1.4.4.1.2. SecurityDomain

```
/**
 * Annotation for specifying the JBoss security domain for an EJB
 *
 * @author <a href="mailto:bill@jboss.org [mailto:bill@jboss.org]">Bill
 Burke</a>
 **/
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface SecurityDomain
{
   /**
    * The required name for the security domain.
    *
    * Do not use the JNDI name
    *
    *    Good: "MyDomain"
    *    Bad:  "java:/jaas/MyDomain"
```

```
     */
    String value();

    /**
     * The name for the unauthenticated pricipal
     */
    String unauthenticatedPrincipal() default "";
```

4.1.4.5. JAXB Introductions

As Kohsuke Kawaguchi writes on *his blog* [http://weblogs.java.net/blog/kohsuke/archive/2007/07/binding_3rd_par.html], one common complaint from the JAXB users is the lack of support for binding 3rd party classes. The scenario is this you are trying to annotate your classes with JAXB annotations to make it XML bindable, but some of the classes are coming from libraries and JDK, and thus you cannot put necessary JAXB annotations on it.

To solve this JAXB has been designed to provide hooks for programmatic introduction of annotations to the runtime.

This is currently leveraged by the JBoss JAXB Introductions project, using which users can define annotations in XML and make JAXB see those as if those were in the class files (perhaps coming from 3rd party libraries).

JAXB Introductions are currently supported in JBossWS-Native (server side only, since 3.0.2.GA) and JBossWS-CXF (both server and client side, since 3.2.1.GA).

Take a look at the *JAXB Introductions page* [http://community.jboss.org/docs/DOC-10075] on the wiki and at the examples in the sources.

4.1.5. Tools

The JBossWS Web Service Framework provides unified tooling for all the supported stacks. This currently includes common JAX-WS tools for both contract-first and code-first development and common management tools.

4.1.5.1. JAX-WS tools

Please refer to *JBossWS_JAX-WS_Tools* for details. This covers directions on web service contract generation (bottom-up development) and consumption (top-down and client development).

4.1.5.2. Management tools

JBoss and its web service framework come with some tools allowing WS endpoint management.

Please refer the *Endpoint management* page for an overview of the available tools. In particular the *JBossWS - Records management* gives administrators a means of performing custom analysis of their web service traffic as well as exporting communication logs.

4.1.5.3. Web Service console

All supported stacks provide a web console for getting the list of the endpoints currently deployed on a given host as well as basic metrics regarding invocations to them. The console is available at *http://localhost:8080/jbossws/services* assuming your application server is currently bound to localhost:8080.

4.1.6. Configuration

4.1.6.1. Address rewrite

JBossWS allows users to configure the soap:address attribute in the wsdl contract of deployed services as well as wsdl address in the web service console. [due to a known issue this does not currently work with JBossWS-Metro, see: *JBWS-2462* [https://jira.jboss.org/jira/browse/JBWS-2462]]

Server configuration options

There're few attributes in the jbossws deployers configuration (currently in jbossws.deployer/META-INF/stack-agnostic-jboss-beans.xml) controlling the way the soap:address attribute in the wsdl is rewritten.

```
<bean name="WSServerConfig"
 class="org.jboss.webservices.integration.config.ServerConfigImpl">
    <property name="mbeanServer"><inject bean="WSMBeanServerLocator"
 property="mbeanServer"/></property>

    <property name="webServiceHost">${jboss.bind.address}</property>
    <property name="modifySOAPAddress">true</property>

    <!--
      <property name="webServiceSecurePort">8443</property>
      <property name="webServicePort">8080</property>
    -->
  </bean>
```

If the content of *<soap:address>* in the wsdl is a valid URL, JBossWS will not rewrite it unless *modifySOAPAddress* is true. If the content of *<soap:address>* is not a valid URL instead, JBossWS will always rewrite it using the attribute values given below. Please note that the variable *${jboss.bind.address}* can be used to set the address which the application is bound to at each startup.

The webServiceSecurePort and webServicePort attributes are used to explicitly define the ports to be used for rewriting the SOAP address. If these attributes are not set, the ports will be identified by querying the list of installed connectors. If multiple connectors are found the port of the first connector is used.

Dynamic rewrite

When the application server is bound to multiple addresses or non-trivial real-world network architectures cause request for different external addresses to hit the same endpoint, a static rewrite of the soap:address may not be enough. JBossWS allows for both the soap:address in the wsdl and the wsdl address in the console to be rewritten with the host use in the client request. This way, users always get the right wsdl address assuming they're connecting to an instance having the endpoint they're looking for. To trigger this behaviour, the **jbossws.undefined.host** value has to be specified for the *webServiceHost* attribute.

```
<property name="webServiceHost">jbossws.undefined.host</property>
<property name="modifySOAPAddress">true</property>
```

Of course, when a confidential transport address is required, the addresses are always rewritten using https protocol and the port currently configured for the https/ssl connector.

JBossWS-StackCXFUserGuide

Note

This page covers features available in **JBossWS CXF stack only**. *Please refer to the common user guide* for a basic introduction to JAX-WS programming as well as documentation on all features, tools, etc. the JBossWS Web Service Framework provides for every supported stack (including CXF stack).

Also please note this page does not go through the documentation of every feature, option, etc. provided by Apache CXF; on the countrary the only topics covered here are specific issues regarding integration with JBoss and stack specific features provided by JBossWS Web Service Framework for the CXF stack. A few tutorials are also provided for show how to leverage some WS technologies.

The *official Apache CXF documentation* is available *here* [http://cxf.apache.org].

5.1. JBossWS CXF Integration

5.1.1. Creating a Bus instance

Most of the Apache CXF features are configurable using the *org.apache.cxf.Bus* class. New Bus instances are produced by the currently configured *org.apache.cxf.BusFactory* implementation the following way:

```
Bus bus = BusFactory.newInstance().createBus();
```

The algorithm for selecting the actual implementation of BusFactory to be used leverages the Service API, basically looking for optional configurations in META-INF/services/... location using the current classloader. JBossWS-CXF integration comes with his own implementation of BusFactory, *org.jboss.wsf.stack.cxf.client.configuration.JBossWSBusFactory*, that allows for automatic detection of Spring availability as well as seamless setup of JBossWS customizations on top of Apache CXF. JBossWSBusFactory is *automatically* retrieved by the BusFactory.newInstance() call above.

JBossWS users willing to explicitely use functionalities of *org.apache.cxf.bus.spring.SpringBusFactory* or *org.apache.cxf.bus.CXFBusFactory*, get the same API with JBossWS additions through JBossWSBusFactory:

```
String myConfigFile = ...
Bus bus = new JBossWSBusFactory().createBus(myConfigFile);
```

```
Map<Class, Object> myExtensions = new HashMap<Class, Object>();
myExtensions.put(...);
Bus bus = new JBossWSBusFactory().createBus(myExtensions);
```

5.1.2. Server Side Integration Customization

It is possible to customize the JBossWS and CXF integration by incorporating the CXF configuration file to the endpoint deployment archive. In order for that to be possible, JBossWS-CXF requires Spring to be installed in the application server. The Spring Framework libraries installation can be perfomed using the *JBossWS-CXF installation* [http://community.jboss.org/docs/DOC-13545].

The convention is the following:

- 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 user do not provide its own CXF configuration file, a default one is automatically generated during the deployment.

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 implements them. For example see *CXF WS-RM tutorial* customization file. We are providing custom CXF endpoint configuration there to turn on WS-RM feature for endpoint.

Note

Note

*When user incorporates its own CXF configuration to the endpoint archive he must reference either **org.jboss.wsf.stack.cxf.InvokerJSE** or **org.jboss.wsf.stack.cxf.InvokerEJB3** jaxws invoker bean there for each jaxws endpoint.*

5.2. Extended Features

Here [http://cwiki.apache.org/CXF20DOC/ws-support.html] is the CXF documentation about supported WS-* specifications.

5.2.1. WS-Addressing

Apache CXF has a thorough support for WS-Addressing; details are available at the following pages:

CXF WS-Addressing documentation [http://cwiki.apache.org/CXF20DOC/ws-addressing.html] *CXF WS-Addressing configuration* [http://cwiki.apache.org/CXF20DOC/wsaconfiguration.html]

Given the JAXWS specification currently covers WS-Addressing basic fuctionalities, users simply needing to enable it can make use of the @Addressing annotation and AddressingFeature, as shown in the following JBossWS-CXF tutorial:

JBossWS-CXF WS-Addressing Tutorial

5.2.2. WS-ReliableMessaging

The Apache CXF technical documentation on WS-RealiableMessaging is available as a reference at the following pages:

CXF WS-ReliableMessaging documentation [http://cwiki.apache.org/CXF20DOC/ws-reliablemessaging.html] *CXF WS-ReliableMessaging configuration* [http://cwiki.apache.org/CXF20DOC/wsrmconfiguration.html]

For a complete tutorial on how to enable WS-ReliableMessaging in a user client-server application, please take a look at:

JBossWS-CXF WS-ReliableMessaging Tutorial

5.2.3. WS-Policy

Apache CXF technical documentation on the WS-Policy engine and its configuration is available at:

CXF WS-Policy documentation [http://cwiki.apache.org/CXF20DOC/ws-policy.html] *CXF WS-Policy configuration* [http://cwiki.apache.org/CXF20DOC/wspconfiguration.html]

For a complete sample of WS-Policy usage, please take a look at the JBossWS-CXF WS-ReliableMessaging tutorial below, as WS-RM is implemented leveraging policies there:

JBossWS-CXF WS-Policy & WS-ReliableMessaging Tutorial

5.2.3.1. Note on PolicyEngine setup

When building up the Bus without Spring libraries available on the classpath, JBossWSBusFactory still makes sure the PolicyEngine (as well as the RMManager) is properly setup. This allows users to leverage basic WS-Policy functionalities the same way they'd do with a full Spring-enabled Bus.

5.2.4. WS-Security

Apache CXF [http://cxf.apache.org/] leverages *WSS4J* [http://ws.apache.org/wss4j/] to provide WS-Security functionalities. This means that thanks to the JBossWS-CXF integration, users can create web service applications using CXF - WSS4J implementation of WS-Security and deploy them on JBoss Application Server.

5.2.4.1. WSS4J security on JBoss

The Apache CXF documentation features an brief chapter on *how to use WSS4J security in CXF* [http://cwiki.apache.org/CXF20DOC/ws-security.html]. Here below instead you'll find some explanations on how to create a simple application and what you need to do to leverage WSS4J security on JBoss.

Creating the web service endpoint

First of all you need to create the web service endpoint / 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 (bottom-up approach);
2. run the **wsconsume** JBoss commandline tool to get the client artifacts from the service contract (top-down approach);
3. write your client implementation.

Turn on WS-Security

WSS4J security is triggered through interceptors that are added to the service and/or client. These interceptors allows you to perform the most common WS-Security related process:

- 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 *jboss-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/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.crypto.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's a trivial 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 client side, you can similarly setup the interceptors programmatically; here is an excerpt of the client for the above described endpoint (of course you can also leverage a proper Spring configuration for loading an already configured CXF Bus instance):

```
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());
```

Package and deploy

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

- the jbossws-cxf.xml descriptor
- the properties file
- the keystore file (if required for signature/encryption)

- the keystore password callback handler class

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

```
[alessio@localhost 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.class
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 setup the interceptors programmatically).

Check that JBossWS-CXF is installed on your current JBoss Application Server, deploy and test your WS-Security-enabled application.

5.2.4.2. WS-Security Policies

Starting from JBossWS-CXF 3.1.1, WS-Security Policy implementation is available and can be used to configure WS-Security more easily.

Please refer to the *Apache CXF documentation* [http://cwiki.apache.org/CXF20DOC/ws-securitypolicy.html]; basically instead of manually configuring interceptors in the client or through 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/IncludeToken/
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/IncludeToken/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>
  ...
```

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

```
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER,
 new KeystorePasswordCallback());
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_PROPERTIES,
 Thread.currentThread().getContextClassLoader().getResource("META-INF/
alice.properties"));
((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.wsse
  >
      </jaxws:properties>
    </jaxws:endpoint>
  </beans>
```

5.2.4.3. Authentication and authorization

The Username Token Profile can of course be used to provide client's credentials to the target endpoint. Starting from JBossWS-CXF 3.3.0 (which includes Apache CXF 2.2.8), the username token information can be used for authentication and authorization on JBoss AS (JAAS integration).

On server side, you need to specify what follows (for instance using a *jbossws-cxf.xml* descriptor):

- an interceptor for performing authentication and populating a valid SecurityContext; the provided interceptor should extend org.apache.cxf.ws.security.wss4j.AbstractUsernameTokenAuthenticatingInterceptor, in particular JBossWS integration comes with *org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingInterceptor* for this;
- an interceptor for performing authorization; CXF requires that to extend org.apache.cxf.interceptor.security.AbstractAuthorizingInInterceptor, for instance the *SimpleAuthorizingInterceptor* can be used for simply mapping endpoint operations to allowed roles.

```
<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:util='http://www.springframework.org/schema/util'
  xsi:schemaLocation='http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-2.0.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="SecurityContextIn"

   class="org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingInterceptor">
      <constructor-arg>
        <map>
          <entry key="action" value="UsernameToken"/>
        </map>
      </constructor-arg>
    </bean>

    <util:map id="methodPermissions">
       <entry key="sayHello" value="friend"/>
       <entry key="greetMe" value="snoopies"/>
    </util:map>

    <bean id="AuthorizeIn"

   class="org.apache.cxf.interceptor.security.SimpleAuthorizingInterceptor">
     <property name="methodRolesMap" ref="methodPermissions"/>
    </bean>

    <jaxws:endpoint
      id='ServiceImpl'

   address='http://@jboss.bind.address@:8080/jaxws-samples-wsse-username-
  authorize'
      implementor='org.jboss.test.ws.jaxws.samples.wsse.ServiceImpl'>
      <jaxws:inInterceptors>
          <ref bean="SecurityContextIn"/>
          <ref bean="AuthorizeIn"/>
          <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor"/>
      </jaxws:inInterceptors>
    </jaxws:endpoint>
  </beans>
```

Authentication and authorization will simply be delegated to the security domain configured for the endpoint. Of course you can specify the login module you prefer for that security domain (refer the application server / security documentation for that).

On client side, the username is provided through API (or a custom Spring configuration used to load the Bus):

```
Endpoint cxfEndpoint = client.getEndpoint();
Map<String, Object> outProps = new HashMap<String, Object>();
outProps.put("action", "UsernameToken");
outProps.put("user", username);
outProps.put("passwordType", "PasswordText");
outProps.put("passwordCallbackClass",
 "org.jboss.test.ws.jaxws.samples.wsse.UsernamePasswordCallback");
WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor(outProps); //request
cxfEndpoint.getOutInterceptors().add(wssOut);
cxfEndpoint.getOutInterceptors().add(new SAAJOutInterceptor());
```

The password instead is provided through a password callback handler that needs to implement *javax.security.auth.callback.CallbackHandler*, similarly to the keystore's password callback handler.

If you're running an older JBossWS-CXF version, or you're not interested in the the application server auth integration, you can use a password callback handler on server side too, configured through a WSS4JInInterceptor:

```
<bean id="UsernameToken_Request">
  <constructor-arg>
    <map>
      <entry key="action" value="UsernameToken"/>
      <entry key="passwordCallbackClass" value="org.jboss.test.ws.jaxws.samples.wsse.ServerU
>
    </map>
  </constructor-arg>
</bean>
```

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

import java.io.IOException;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ServerUsernamePasswordCallback implements CallbackHandler
{
   public void handle(Callback[] callbacks) throws IOException,
 UnsupportedCallbackException
   {
      WSPasswordCallback pc = (WSPasswordCallback)callbacks[0];
      if (!("kermit".equals(pc.getIdentifier()) &&
  "thefrog".equals(pc.getPassword())))
         throw new SecurityException("User '" + pc.getIdentifier() + "' with
  password '" + pc.getPassword() + "' not allowed.");
    }
 }
```

5.2.4.4. Further information

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* .

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 need to be configured in your JRE. Please refer the *Native stack user* [http://community.jboss.org/docs/DOC-13532] guide for further information about this.

5.2.5. JMS transport

Here is a tutorial on how to deploy and invoke a JMS endpoint using JBossWS-CXF.

5.3. HTTP server transport setup

Apache CXF comes with pluggable transport layers, allowing different transport modules to be used.

The JBossWS-CXF integration leverages CXF servlet transport for the deployment of endpoints on top of the running JBoss Application Server.

However, when users directly leverage the JAXWS *Endpoint.publish(String s)* [http://download.oracle.com/javase/6/docs/api/javax/xml/ws/Endpoint.html#publish%28java.lang.String%29] API, endpoints are expected to be deployed on a standalone http server started just for serving the specified endpoint. Apache CXF currently defaults to using the *Jetty* [http://jetty.codehaus.org/jetty/] based http transport. Starting *from release 3.4.0*, the JBossWS-CXF integration instead uses a different http transport module based on the *http server* [http://download.oracle.com/javase/6/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/package-summary.html] embedded in JDK6 distributions. Thanks to Apache CXF transport pluggability, users can still change the transport they want to use in this case by simply replacing the *jbossws-cxf-transports-httpserver.jar* library with another http transport one, for instance the *cxf-rt-transports-http-jetty.jar*.

5.4. SOAP Message Logging

In the jbossws-cxf-client.jar[*] file you will find META-INF/cxf/cxf-extension-jbossws.xml, which contains the JBossWS extensions to the Apache CXF stack. In that file 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/zip.

[*] The cxf-extension-jbossws.xml is available from version 3.2.2; if you don't have that file, you can manually add it and link it in cxf.extensions file.

Finally, please note that logging can be enabled in many ways with Apache CXF, see the following documentation pages for instance:

- *http://cxf.apache.org/docs/configuration.html*
- *http://cxf.apache.org/docs/debugging-and-logging.html*

JBossWS-JAX-WSTools

The JAX-WS tools provided by JBossWS can be used in a variety of ways. First we will look at server-side development strategies, and then proceed to the client.

6.1. Server side

When developing a Web Service Endpoint (the server-side) you have the option of starting from Java (bottom-up development), or from the abstact contract (WSDL) that defines your service (top-down development). If this is a new service (no existing contract), the bottom-up approach is the fastest route; you only need to add a few annotations to your classes to get a service up and running. However, if you are developing a service with an already defined contract, it is far simpler to use the top-down approach, since the provided tool will generate the annotated code for you.

Bottom-up use cases:

- Exposing an already existing EJB3 bean as a Web Service
- Providing a new service, and you want the contract to be generated for you

Top-down use cases:

- Replacing the implementation of an existing Web Service, and you can't break compatibility with older clients
- Exposing a service that conforms to a contract specified by a third party (e.g. a vender that calls you back using an already defined protocol).
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front

The following JAX-WS command line tools are included in JBossWS:

Command	Description
JBossWS - wsprovide	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
JBossWS - wsconsume	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development
JBossWS - wsrunclient	Executes a Java client (has a main method) using the JBossWS classpath.

6.1.1. Bottom-Up (Using wsprovide)

The bottom-up strategy involves developing the Java code for your service, and then annotating it using JAX-WS annotations. These annotations can be used to customize the contract that is generated for your service. For example, you can change the operation name to map to anything you like. However, all of the annotations have sensible defaults, so only the @WebService annotation is required.

This can be as simple as creating a single class:

```
package echo;

@javax.jws.WebService
public class Echo
{
   public String echo(String input)
   {
      return input;
   }
}
```

A JSE or EJB3 deployment can be built using this class, and it is the only Java code needed to deploy on JBossWS. The WSDL, and all other Java artifacts called "wrapper classes" will be generated for you at deploy time. This actually goes beyond the JAX-WS specification, which requires that wrapper classes be generated using an offline tool. The reason for this requirement is purely a vender implementation problem, and since we do not believe in burdening a developer with a bunch of additional steps, we generate these as well. However, if you want your deployment to be portable to other application servers, you will unfortunately need to use a tool and add the generated classes to your deployment.

This is the primary purpose of the *JBossWS - wsprovide*tool, to generate portable JAX-WS artifacts. Additionally, it can be used to "provide" the abstract contract (WSDL file) for your service. This can be obtained by invoking *JBossWS - wsprovide* using the "-w" option:

```
$ javac -d . -classpath jboss-jaxws.jar Echo.java
$ wsprovide -w echo.Echo
Generating WSDL:
EchoService.wsdl
Writing Classes:
echo/jaxws/Echo.class
echo/jaxws/EchoResponse.class
```

Inspecting the WSDL reveals a service called EchoService:

```
<service name='EchoService'><port binding='tns:EchoBinding' name='EchoPort'>
  <soap:address location='REPLACE_WITH_ACTUAL_URL'/></port>
</service>
```

As expected, this service defines one operation, "echo":

```
<portType name='Echo'><operation name='echo' parameterOrder='echo'>
  <input message='tns:Echo_echo'/>
  <output message='tns:Echo_echoResponse'/></operation>
</portType>
```

Note

Note

*Remember that **when deploying on JBossWS you do not need to run this tool.** You only need it for generating portable artifacts and/or the abstract contract for your service.*

Let's create a POJO endpoint for deployment on JBoss AS. A simple web.xml needs to be created:

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

  <servlet>
    <servlet-name>Echo</servlet-name>
    <servlet-class>echo.Echo</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Echo</servlet-name>
    <url-pattern>/Echo</url-pattern>
  </servlet-mapping>
</web-app>
```

The web.xml and the single class can now be used to create a war:

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

The war can then be deployed:

```
cp echo.war /usr/local/jboss-4.2.0.GA-ejb3/server/default/deploy
```

This will internally invoke *JBossWS - wsprovide*, which will generate the WSDL. If deployment was successful, and you are using the default settings, it should be available here: *http://localhost:8080/echo/Echo?wsdl*

For a portable JAX-WS deployment, the wrapper classes generated earlier could be added to the deployment.

6.1.2. Top-Down (Using wsconsume)

The top-down development strategy begins with the abstract contract for the service, which includes the WSDL file and zero or more schema files. The *JBossWS - wsconsume* tool is then used to consume this contract, and produce annotated Java classes (and optionally sources) that define it.

Note

Note

wsconsume seems to have a problem with symlinks on unix systems

Using the WSDL file from the bottom-up example, a new Java implementation that adheres to this service can be generated. The "-k" option is passed to *JBossWS - wsconsume* to preserve the Java source files that are generated, instead of providing just classes:

```
$ wsconsume -k EchoService.wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The following table shows the purpose of each generated file:

File	Purpose
Echo.java	Service Endpoint Interface
Echo_Type.java	Wrapper bean for request message
EchoResponse.java	Wrapper bean for response message

File	Purpose
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
EchoService.java	Used only by JAX-WS clients

Examining the Service Endpoint Interface reveals annotations that are more explicit than in the class written by hand in the bottom-up example, however, these evaluate to the same contract:

```
@WebService(name = "Echo", targetNamespace = "http://echo/")
public interface Echo {
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "echo", targetNamespace = "http://echo/",
 className = "echo.Echo_Type")
    @ResponseWrapper(localName = "echoResponse", targetNamespace =
 "http://echo/", className = "echo.EchoResponse")
    public String echo(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);

}
```

The only missing piece (besides the packaging) is the implementation class, which can now be written, using the above interface.

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo
{
   public String echo(String arg0)
   {
      return arg0;
   }
}
```

6.2. Client Side

Before going to detail on the client-side it is important to understand the decoupling concept that is central to Web Services. Web Services are not the best fit for internal RPC, even though they can be used in this way. There are much better technologies for this (CORBA, and RMI for example). Web Services were designed specifically for interoperable coarse-grained correspondence. There is no expectation or guarantee that any party participating in a Web Service interaction will be at any particular location, running on any particular OS, or written in any particular programming language. So because of this, it is important to clearly separate client and server implementations. The only thing they should have in common is the abstract contract definition. If, for whatever reason, your software does not adhere to this principal, then you should not be using Web Services. For the above reasons, the ***recommended methodology for developing a client is*** to follow ***the top-down approach***, even if the client is running on the same server.

Let's repeat the process of the top-down section, although using the deployed WSDL, instead of the one generated offline by *JBossWS - wsprovide*. The reason why we do this is just to get the right value for soap:address. This value must be computed at deploy time, since it is based on container configuration specifics. You could of course edit the WSDL file yourself, although you need to ensure that the path is correct.

Offline version:

```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
   <soap:address location='REPLACE_WITH_ACTUAL_URL'/>
  </port></service>
```

Online version:

```
<service name="EchoService">
  <port binding="tns:EchoBinding" name="EchoPort">
    <soap:address location="http://localhost.localdomain:8080/echo/Echo"/>
  </port>
</service>
```

Using the online deployed version with *JBossWS - wsconsume*:

```
$ wsconsume -k http://localhost:8080/echo/Echo?wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The one class that was not examined in the top-down section, was EchoService.java. Notice how it stores the location the WSDL was obtained from.

```
@WebServiceClient(name = "EchoService", targetNamespace = "http://echo/",
 wsdlLocation = "http://localhost:8080/echo/Echo?wsdl")
public class EchoService extends Service
```

```
{
    private final static URL ECHOSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:8080/echo/Echo?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        ECHOSERVICE_WSDL_LOCATION = url;
    }

    public EchoService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public EchoService() {
        super(ECHOSERVICE_WSDL_LOCATION, new QName("http://echo/",
 "EchoService"));
    }

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort() {
        return (Echo)super.getPort(new QName("http://echo/", "EchoPort"),
 Echo.class);
    }
}
```

As you can see, this generated class extends the main client entry point in JAX-WS, javax.xml.ws.Service. While you can use Service directly, this is far simpler since it provides the configuration info for you. The only method we really care about is the getEchoPort() method, which returns an instance of our Service Endpoint Interface. Any WS operation can then be called by just invoking a method on the returned interface.

Note

Note

It's not recommended to refer to a remote WSDL URL in a production application. This causes network I/O every time you instantiate the Service Object. Instead, use the tool on a saved local copy, or use the URL version of the constructor to provide a new WSDL location.

All that is left to do, is write and compile the client:

```
import echo.*;
```

```
public class EchoClient
{
   public static void main(String args[])
   {
      if (args.length != 1)
      {
         System.err.println("usage: EchoClient <message>");
         System.exit(1);
      }

      EchoService service = new EchoService();
      Echo echo = service.getEchoPort();
      System.out.println("Server said: " + echo.echo(args[0]));
   }}
```

It can then be easily executed using the *JBossWS - wsrunclient* tool. This is just a convenience tool that invokes java with the needed classpath:

```
$ wsrunclient EchoClient 'Hello World!'
Server said: Hello World!
```

It is easy to change the endpoint address of your operation at runtime, setting the ENDPOINT_ADDRESS_PROPERTY as shown below:

```
...
      EchoService service = new EchoService();
      Echo echo = service.getEchoPort();

      /* Set NEW Endpoint Location */
      String endpointURL = "http://NEW_ENDPOINT_URL";
      BindingProvider bp = (BindingProvider)echo;
      bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
 endpointURL);

      System.out.println("Server said: " + echo.echo(args[0]));
...
```

6.3. Command-line, Maven Plugin and Ant Task Reference

- *JBossWS - wsconsume* reference page
- *JBossWS - wsprovide* reference page
- *JBossWS - wsrunclient* reference page

6.4. JAX-WS binding customization

An introduction to binding customizations:

- *http://java.sun.com/webservices/docs/2.0/jaxws/customizations.html*
- *binding schema* [https://jax-ws.dev.java.net/source/browse/jax-ws/guide/docs/wsdl-customization.xsd?rev=1.2&view=log]
- *xnsdoc* [https://jax-ws.dev.java.net/nonav/guide/customizations/]

The schema for the binding customization files can be found here:

- *https://jax-ws.dev.java.net/source/browse/jax-ws/guide/docs/wsdl-customization.xsd?rev=1.2&view=log*

JBossWS-wsconsume

wsconsume is a command line tool and ant task that "consumes" the abstract contract (WSDL file) and produces portable JAX-WS service and client artifacts. For a more detailed overview, see *"Using wsconsume"* [http://community.jboss.org/docs/DOC-13544#TopDown_Using_wsconsume].

7.1. Command Line Tool

The command line tool has the following usage:

```
usage: wsconsume [options] <wsdl-url>
options:
  -h, --help                    Show this help message
  -b, --binding=<file>          One or more JAX-WS or JAXB binding files
  -k, --keep                    Keep/Generate Java source
  -c  --catalog=<file>          Oasis XML Catalog file for entity resolution
  -p  --package=<name>          The target package for generated source
  -w  --wsdlLocation=<loc>      Value to use for
 @WebServiceClient.wsdlLocation
  -o, --output=<directory>      The directory to put generated artifacts
  -s, --source=<directory>      The directory to put Java source
  -t, --target=<2.0|2.1|2.2>    The JAX-WS specification target
  -q, --quiet                   Be somewhat more quiet
  -v, --verbose                 Show full exception stack traces
  -l, --load-consumer           Load the consumer and exit (debug utility)
  -e, --extension               Enable SOAP 1.2 binding extension
  -a, --additionalHeaders       Enables processing of implicit SOAP headers
```

Note : The wsdlLocation is used when creating the Service to be used by clients and will be added to the @WebServiceClient annotation, for an endpoint implementation based on the generated service endpoint interface you will need to manually add the wsdlLocation to the @WebService annotation on your web service implementation and not the service endpoint interface.

7.1.1. Examples

Generate artifacts in Java class form only:

```
wsconsume Example.wsdl
```

Generate source and class files:

```
wsconsume -k Example.wsdl
```

Generate source and class files in a custom directory:

```
wsconsume -k -o custom Example.wsdl
```

Generate source and class files in the org.foo package:

```
wsconsume -k -p org.foo Example.wsdl
```

Generate source and class files using multiple binding files:

```
wsconsume -k -b wsdl-binding.xml -b schema1-binding.xml -b
 schema2-binding.xml
```

7.2. Maven Plugin

The wsconsume tools is included in the **org.jboss.ws.plugins:maven-jaxws-tools-plugin** plugin. The plugin has two goals for running the tool, *wsconsume* and *wsconsume-test*, which basically do the same during different maven build phases (the former triggers the sources generation during *generate-sources* phase, the latter during the *generate-test-sources* one).

The wsconsume plugin has the following parameters:

Attribute	Description	Default
bindingFiles	JAXWS or JAXB binding file	true
classpathElements	Each classpathElement provides a library file to be added to classpath	${project.compileClasspathElements} or ${project.testClasspathElements}
catalog	Oasis XML Catalog file for entity resolution	none
targetPackage	The target Java package for generated code.	generated
bindingFiles	One or more JAX-WS or JAXB binding file	none
wsdlLocation	Value to use for @WebServiceClient.wsdlLocation	generated
outputDirectory	The output directory for generated artifacts.	${project.build.outputDirectory} or ${project.build.testOutputDirectory}
sourceDirectory	The output directory for Java source.	${project.build.directory}/ wsconsume/java

Attribute	Description	Default
verbose	Enables more informational output about command progress.	false
wsdls	The WSDL files or URLs to consume	n/a
extension	Enable SOAP 1.2 binding extension.	false

7.2.1. Examples

You can use wsconsume in your own project build simply referencing the *maven-jaxws-tools-plugin* in the configured plugins in your pom.xml file.

The following example makes the plugin consume the test.wsdl file and generate SEI and wrappers' java sources. The generated sources are then compiled together with the other project classes.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>maven-jaxws-tools-plugin</artifactId>
      <version>1.0.0.GA</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
        </wsdls>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

You can also specify multiple wsdl files, as well as force the target package, enable SOAP 1.2 binding and turn the tool's verbose mode on:

```
<build>
```

```
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>maven-jaxws-tools-plugin</artifactId>
      <version>1.0.0.GA</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
          <wsdl>${basedir}/test2.wsdl</wsdl>
        </wsdls>
        <targetPackage>foo.bar</targetPackage>
        <extension>true</extension>
        <verbose>true</verbose>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Finally, if the wsconsume invocation is required for consuming a wsdl to be used in your testsuite only, you might want to use the *wsconsume-test* goal as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>maven-jaxws-tools-plugin</artifactId>
      <version>1.0.0.GA</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
        </wsdls>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume-test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

7.3. Ant Task

Note

Note

With 2.0.GA the task was renamed to org.jboss.wsf.spi.tools.ant.WSConsumeTask. Also put streamBuffer.jar and stax-ex.jar in the classpath of the ant task to generate the appropriate artefacts. Both jar files are in the jbossws lib directory. For jbossws-native-2.0.3.GA these files are not automatically installed if you run jboss-deployXX.

The wsconsume ant task has the following attributes:

Attribute	Description	Default
fork	Whether or not to run the generation task in a separate VM.	true
keep	Keep/Enable Java source code generation.	false
catalog	Oasis XML Catalog file for entity resolution	none
package	The target Java package for generated code.	generated
binding	A JAX-WS or JAXB binding file	none
wsdlLocation	Value to use for @WebServiceClient.wsdlLocation	generated
destdir	The output directory for generated artifacts.	"output"
sourcedestdir	The output directory for Java source.	value of destdir
target	The JAX-WS specification target. Allowed values are 2.0, 2.1 and 2.2	
verbose	Enables more informational output about command progress.	false
wsdl	The WSDL file or URL	n/a

Attribute	Description	Default
extension	Enable SOAP 1.2 binding extension.	false
additionalHeaders	Enables processing of implicit SOAP headers	false

Note : The wsdlLocation is used when creating the Service to be used by clients and will be added to the @WebServiceClient annotation, for an endpoint implementation based on the generated service endpoint interface you will need to manually add the wsdlLocation to the @WebService annotation on your web service implementation and not the service endpoint interface.

Also, the following nested elements are supported:

Element	Description	Default
binding	A JAXWS or JAXB binding file	none
jvmarg	Allows setting of custom jvm arguments	

7.3.1. Examples

Generate JAX-WS source and classes in a separate JVM with separate directories, a custom wsdl location attribute, and a list of binding files from foo.wsdl:

```
<wsconsume
  fork="true"
  verbose="true"
  destdir="output"
  sourcedestdir="gen-src"
  keep="true"
  wsdllocation="handEdited.wsdl"
  wsdl="foo.wsdl">
  <binding dir="binding-files" includes="*.xml" excludes="bad.xml"/>
</wsconsume>
```

7.4. Related information

- *JAX-WS binding customization* [http://java.sun.com/webservices/docs/2.0/jaxws/customizations.html]

JBossWS-wsprovide

wsprovide is a command line tool and ant task that generates portable JAX-WS artifacts for a service endpoint implementation. It also has the option to "provide" the abstract contract for offline usage. See *"Using wsprovide"* [http://community.jboss.org/docs/DOC-13544#BottomUp_Using_wsprovide] for a detailed walk-through.

8.1. Command Line Tool

The command line tool has the following usage:

```
    usage: wsprovide [options] <endpoint class name>
    options:
    -h, --help                  Show this help message
    -k, --keep                  Keep/Generate Java source
    -w, --wsdl                  Enable WSDL file generation
    -c. --classpath=<path<      The classpath that contains the endpoint
    -o, --output=<directory>    The directory to put generated artifacts
    -r, --resource=<directory>  The directory to put resource artifacts
    -s, --source=<directory>    The directory to put Java source
    -e, --extension             Enable SOAP 1.2 binding extension
    -q, --quiet                 Be somewhat more quiet
    -t, --show-traces           Show full exception stack traces
```

8.1.1. Examples

Generating wrapper classes for portable artifacts in the "generated" directory:

```
wsprovide -o generated foo.Endpoint
```

Generating wrapper classes and WSDL in the "generated" directory

```
wsprovide -o generated -w foo.Endpoint
```

Using an endpoint that references other jars

```
wsprovide -o generated -c application1.jar:application2.jar foo.Endpoint
```

8.2. Maven Plugin

The wsprovide tools is included in the **org.jboss.ws.plugins:maven-jaxws-tools-plugin** plugin. The plugin has two goals for running the tool, *wsprovide* and *wsprovide-test*, which basically do the same during different maven build phases (the former triggers the sources generation during *process-classes* phase, the latter during the *process-test-classes* one).

The wsprovide plugin has the following parameters:

Attribute	Description	Default
testClasspathElements	Each classpathElement provides a library file to be added to classpath	${project.compileClasspathElements} or ${project.testClasspathElements}
outputDirectory	The output directory for generated artifacts.	${project.build.outputDirectory} or ${project.build.testOutputDirectory}
resourceDirectory	The output directory for resource artifacts (WSDL/XSD).	${project.build.directory}/wsprovide/resources
sourceDirectory	The output directory for Java source.	${project.build.directory}/wsprovide/java
extension	Enable SOAP 1.2 binding extension.	false
generateWsdl	Whether or not to generate WSDL.	false
verbose	Enables more informational output about command progress.	false
endpointClass	**Service Endpoint Implementation.**	

8.2.1. Examples

You can use wsprovide in your own project build simply referencing the *maven-jaxws-tools-plugin* in the configured plugins in your pom.xml file.

The following example makes the plugin provide the wsdl file and artifact sources for the specified endpoint class:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>maven-jaxws-tools-plugin</artifactId>
      <version>@pom.version@</version>
      <configuration>
        <verbose>true</verbose>

 <endpointClass>org.jboss.test.ws.plugins.tools.wsprovide.TestEndpoint</
endpointClass>
```

```
        <generateWsdl>true</generateWsdl>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsprovide</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The following example does the same, but is meant for use in your own testsuite:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>maven-jaxws-tools-plugin</artifactId>
      <version>@pom.version@</version>
      <configuration>
        <verbose>true</verbose>

 <endpointClass>org.jboss.test.ws.plugins.tools.wsprovide.TestEndpoint2</
endpointClass>
        <generateWsdl>true</generateWsdl>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsprovide-test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

8.3. Ant Task

Note

Note

With 2.0.GA the task was renamed to org.jboss.wsf.spi.tools.ant.WSProvideTask

The wsprovide ant task has the following attributes:

Attribute	Description	Default
fork	Whether or not to run the generation task in a separate VM.	true
keep	Keep/Enable Java source code generation.	false
destdir	The output directory for generated artifacts.	"output"
resourcedestdir	The output directory for resource artifacts (WSDL/XSD).	value of destdir
sourcedestdir	The output directory for Java source.	value of destdir
extension	Enable SOAP 1.2 binding extension.	false
genwsdl	Whether or not to generate WSDL.	false
verbose	Enables more informational output about command progress.	false
sei	**Service Endpoint Implementation.**	
classpath	The classpath that contains the service endpoint implementation.	"."

8.3.1. Examples

Executing wsprovide in verbose mode with separate output directories for source, resources, and classes:

```
<target name="test-wsproivde" depends="init">
  <taskdef name="wsprovide"
 classname="org.jboss.wsf.spi.tools.ant.WSProvideTask">
    <classpath refid="core.classpath"/>
  </taskdef>
  <wsprovide
    fork="false"
```

```
      keep="true"
      destdir="out"
      resourcedestdir="out-resource"
      sourcedestdir="out-source"
      genwsdl="true"
      verbose="true"
      sei="org.jboss.test.ws.jaxws.jsr181.soapbinding.DocWrappedServiceImpl">
      <classpath>
        <pathelement path="${tests.output.dir}/classes"/>
      </classpath>
    </wsprovide>
  </target>
```

JBossWS-wsrunclient

wsrunclient is a command line tool that invokes a JBossWS JAX-WS Web Service client. It builds the correct classpath and endorsed libs for you. Feel free to use the code for this script to make your own shell scripts. It is open source after all.

9.1. Usage

```
wsrunclient [-classpath <additional class path>] <java-main-class>
[arguments...]
```

9.2. Examples

Invoking a standalone JAX-WS client:

```
wsrunclient echo.EchoClient
```

Invoking a standalone JAX-WS client that uses external jars:

```
wsrunclient -classpath jar1.jar:jar2.jar echo.EchoClient
```

Part III. Additional documentation

This section of the book provides documentation on common additional user requirements, like enabling authentication, securing the transport, etc.

JBossWS-Authentication

This page explains the simplest way to authenticate a web service user with JBossWS.

First we secure the access to the SLSB as we would do for normal (non web service) invocations: this can be easily done through the @RolesAllowed, @PermitAll, @DenyAll annotation. 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
{
  ...
}
```

Similarly POJO endpoints are secured the same way as we do for normal web applications in web.xml:

```
    <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>
```

10.1. Define the security domain

Next, define the security domain for this deployment. This is performed using the *@SecurityDomain* [http://community.jboss.org/docs/DOC-13972#SecurityDomain] annotation for EJB3 endpoints

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

or modifying the jboss-web.xml for POJO endpoints

```
<jboss-web>
  <security-domain>java:/jaas/JBossWS</security-domain>
</jboss-web>
```

The JBossWS security context is configured in login-config.xml and uses the *UsersRolesLoginModule* [http://wiki.jboss.org/wiki/Wiki.jsp?page=UsersRolesLoginModule]. As a matter of fact login-config.xml, that lives in the server config dir, contains this security domain definition:

```
    <!--
      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>
```

Of course you can define and use your own security domain as well as your login module (in order to check for users' identity querying a database for example).

10.2. Use BindingProvider to set principal/credential

A web service client may use the javax.xml.ws.BindingProvider interface to set the username/password combination

```
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, "kermit");
bp.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "thefrog");
```

10.3. Using HTTP Basic Auth for security

To enable HTTP Basic authentication you use the *@WebContext* [http://community.jboss.org/docs/DOC-13972#WebContext] annotation on the bean class

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

For POJO endpoints, we modify the web.xml adding the auth-method element:

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

JBossWS-Securetransport

JBossWS allows you to require that requests to a given endpoint use SSL by specifying the *transportGuarantee* attribute in the *@WebContext* [http://community.jboss.org/docs/DOC-13972#WebContext] annotation.

Here is an example using a SLSB endpoint:

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
@WebContext
(
  contextRoot="/my-cxt",
  urlPattern="/*",
  authMethod="BASIC",
  transportGuarantee="CONFIDENTIAL",
  secureWSDLAccess=false
)
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

Similarly to enforce the same requirement on POJO endpoints, you need to edit web.xml and add a user-data-constraint element to your security-constraint element:

```
  <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>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>

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

If you're manually creating your service contract, make sure that the endpoint address in your WSDL file uses a secure protocol. The easiest way is to add "*https://*" to the SOAP Address entry:

```
    <service name="MyService">
     <port name="BasicSecuredPort" binding="tns:MyBinding">
      <soap:address location="https://localhost:8443/my-ctx/SecureEndpoint"/>
     </port>
    </service>
```

For this to work the Tomcat+SSL connector must be enabled:

```
    <Connector port="8443" address="${jboss.bind.address}"
         maxThreads="100" minSpareThreads="5" maxSpareThreads="15"
         scheme="https" secure="true" clientAuth="want"
         keystoreFile="${jboss.server.home.dir}/conf/keystores/wsse.keystore"

         keystorePass="jbossws"

  truststoreFile="${jboss.server.home.dir}/conf/keystores/wsse.keystore"
         truststorePass="jbossws"
         sslProtocol = "TLS" />
```

Please refer the *Tomcat-5.5 SSL Configuration HOWTO* [http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html] for further details.

11.1. Client side

On the client side the truststore must be installed:

```
      <sysproperty key="javax.net.ssl.keyStore"
  value="${test.resources.dir}/wsse/wsse.keystore"/>
      <sysproperty key="javax.net.ssl.trustStore"
  value="${test.resources.dir}/wsse/wsse.truststore"/>
      <sysproperty key="javax.net.ssl.keyStorePassword" value="jbossws"/>
      <sysproperty key="javax.net.ssl.trustStorePassword" value="jbossws"/>
      <sysproperty key="javax.net.ssl.keyStoreType" value="jks"/>
      <sysproperty key="javax.net.ssl.trustStoreType" value="jks"/>
```

As you can see, this requires you to setup the environment specifying both the location and type of your truststore.

Finally, in case you see the following exception:

```
   java.io.IOException: HTTPS hostname wrong:  should be <localhost>
     at
  sun.net.www.protocol.https.HttpsClient.checkURLSpoofing(HttpsClient.java:493)
     at
  sun.net.www.protocol.https.HttpsClient.afterConnect(HttpsClient.java:418)
```

you should disable URL checking on the client side:

```
<sysproperty key="org.jboss.security.ignoreHttpsHost" value="true"/>
```

JBossWS-Endpointmanagement

JBossWS registers MBeans that users can leverage to manage every webservice endpoint. Apart from the obvious start/stop functionalities, they provide valuable information and statistics about messages processed by the endpoints.

12.1. Getting the information

JBoss ships with a JMX-Console with all the application server MBeans. It is usually available at URL *http://localhost:8080/jmx-console*. For endpoint management you might be interested in the MBeans belonging to the jboss.ws domain.

The application server also has an applet based web-console which basically has the same data as the JMX-Console plus some advanced features including snapshot graphics.

Of course you can access an MBean programmatically too. Please refer to the *JBoss JMX faq* [http://wiki.jboss.org/wiki/Wiki.jsp?page=FAQJBossJMX] for further details; here is a brief code snippet you might want to start from in order to access a ManagedEndpointMBean from the same virtual machine:

```
try
{
  MBeanServer server = MBeanServerLocator.locate();
  ManagedEndpointMBean mep = (ManagedEndpointMBean)MBeanProxyExt.create(
            ManagedEndpointMBean.class,
            "jboss.ws:context=my-ctx,endpoint=MyEndpoit",
            server);
  ...
}
catch (Exception e)
{
  e.printStackTrace();
}
```

12.2. Metrics

For each deployed endpoint you'll find an *org.jboss.wsf.framework.management.ManagedEndpoint* MBean providing basic start/stop functionalities and metrics. Calling a stopped endpoint will always result in a SOAP fault message.

The metrics available for each managed endpoint are:

- Min, max, average and total processing time: processing includes both the WS stack plus application server work and the user business logic

- Last start and stop time
- Request, response and fault count

12.3. Records

JBossWS features a highly configurable records' collection and management system. Each record is basically composed of a message plus additional information (for example the caller address and the called endpoint operation).

Endpoints can be configured with record processors that are invoked whenever a message flow is detected and records are thus created.

Every deployed endpoint is configured with default record processors. However custom processors as well as record filters can be easily plugged in and managed at any time through JMX. This gives users the chance of performing advanced analysis of the webservice traffic according to their business requirements.

Please refer to the *records management page* [http://jbossws.jboss.org/mediawiki/index.php?title=Records_management] for further details.

12.4. Snapshots and threshold monitors

As previously said, the *JBoss Web Console* [http://wiki.jboss.org/wiki/Wiki.jsp?page=WebConsole] has interesting features including *snapshots* [http://wiki.jboss.org/wiki/Wiki.jsp?page=WebConsoleSnapshots] and *threshold monitors* [http://wiki.jboss.org/wiki/Wiki.jsp?page=WebConsoleMonitoring].

Snapshots allow users to record changes of a given MBean attribute within a defined time interval. Data are sampled at a given rate and may be plotted to graphs with a few clicks. Snapshots are listed in the Web console and can be created simply browsing to *http://localhost:8080/web-console/createSnapshot.jsp* .

Threshold monitors allow users to be notified whenever a given MBean attribute exceed a certain range of values. The threshold monitor's creation and management processes are similar to those mentioned above for the snapshots. Simply browse to *http://localhost:8080/web-console/createThresholdMonitor.jsp* .

Speaking of WS availability and SLA, this all becomes interesting because users can monitor and take snapshots of critical attributes like the average/max processing time of a managed endpoint. Moreover, advanced analysis can be performed leveraging ad-hoc attributes of custom *record processors* [http://jbossws.jboss.org/mediawiki/index.php?title=Endpoint_management#Records].

JBossWS-Recordsmanagement

JBossWS records' collection and management system gives administrators a means of performing custom analysis of their webservice traffic as well as exporting communication logs.

13.1. What is recorded

Each record is basically composed of a message plus additional information; here are the current record attributes:

- Creation date
- Source host
- Destination host
- Message type (in/out)
- Invoked endpoint operation
- Message envelope (including both soap:header and soap:body for SOAP messages)
- Http headers
- Record group ID (allowing records belonging to the same message flow to be linked together)

Of course records may also have meaningful values for a subset of the afore mentioned record attributes.

13.2. Use cases

What are records useful for? In spite of *endpoint metrics* that provide response time information and counts of invocations, records provide users with rich data about the content of the exchanged messages and their sender/receiver. The record system allows fine grained management and is customizable according to the users need; some of the use cases supported by the default configuration are:

- Logging request and response messages: being able to record messages received from and sent to a given service consumer without stopping the provider may be really useful. You just need to set the *recording* attribute of their endpoint's LogRecorder to true. The added value of this logging solution comes from the use of filters through which messages coming from a given address and related to a given wsdl operation only can be logged.

- Accountability: service providers may want to know which consumers are actually hitting a given service. This can be done for example using the *getClientHosts* functionality of the MemoryBufferRecorder once it has been switched to recording state.
- Getting statistics, filtering records: service administrators might want to see the last records related to a given endpoint or operation, the last records related to messages coming from a given customer and the response the system gave them, etc. These information can be obtained using the *getRecordsByOperation*, *getRecordsByClientHost* or the more general *getMatchingRecords* functionality of the MemoryBufferRecorder.

13.3. How it works and how to use it

The recording system is composed of

- JAX-WS handlers intercepting inbound and outbound communication
- Record processors plugged into deployed endpoints; handlers collect records and send them to every processors through the current endpoint. Processors may store records, convert them, log them, ...
- MBean views of processors that can be used to configure and fine tune recording at runtime
- Record filters allowing selection of information to be recorded as well as providing means of performing custom queries on the saved records.

13.3.1. Server side

On server side records are collected by JAX-WS handlers and passed to the configured processors. JBossWS comes with two default record processors that are plugged into every endpoint during the deployment:

- LogRecorder: a simple record processor that writes records to the configured log.
- MemoryBufferRecorder: a record processor that keeps the last received records in memory and allows user to search / get statistics on them.

Every processors can be fine tuned to process some record attributes only according to the user and/or performance requirements. Default processors are not in recording mode upon creation, thus you need to switch them to recording mode through their MBean interfaces (see the *Recording* flag in the jmx-console).

Common processor properties and their respective defaults values are:

- processDestinationHost (true)

- processSourceHost (true)
- processHeaders (true)
- processEnvelope (true)
- processMessageType (true)
- processOperation (true)
- processDate (true)
- recording (false)

The recorders can be configured in the stacks bean configuration

```
  <!-- Installed Record Processors-->
  <bean name="WSMemoryBufferRecorder"
 class="org.jboss.wsf.framework.management.recording.MemoryBufferRecorder">
    <property name="recording">false</property>
  </bean>
  <bean name="WSLogRecorder"
 class="org.jboss.wsf.framework.management.recording.LogRecorder">
    <property name="recording">false</property>
  </bean>
```

The recording system is available for all the JBossWS supported stacks. However slightly different procedure is required to enable it depending on the used stack.

Native stack

Native stack comes with *JBossWS - JAX-WS Endpoint Configuration* [http://community.jboss.org/docs/DOC-13512]. The default standard endpoint already has the server side recording handler:

```
  <endpoint-config>
    <config-name>Standard Endpoint</config-name>
    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:protocol-bindings>##SOAP11_HTTP</javaee:protocol-bindings>
        <javaee:handler>
          <javaee:handler-name>Recording Handler</javaee:handler-name>

 <javaee:handler-
class>org.jboss.wsf.framework.invocation.RecordingServerHandler</
javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>
    </pre-handler-chains>
  </endpoint-config>
```

thus nothing is required to use it since it is automatically installed in the pre-handler-chain. Of course you might want to add it to other endpoint configurations you're using.

Metro and CXF stacks

Other stacks require users to manually add the *org.jboss.wsf.framework.invocation.RecordingServerHandler* to their endpoint handler chain. This can be done *the same way common user handlers are added*.

Once the handler is properly added to the chain, log recording configuration is agnostic to the used stack. Users just need to tune the processors parameters though their MBean interfaces.

13.3.2. Client side

JMX management of processors is of course available on server side only. However users might also be interested in collecting and processing records on client side. Since handlers can be set on client side too, customer handlers could be configured to capture messages almost like the *RecordingServerHandler* does. This is left to the users since it is directly linked to their custom needs. For instance a common use could be to pass client side collected records to the LogRecorder.

13.4. Advanced hints

13.4.1. Adding custom recorders

As previously said, the recording system is extensible: JBossWS users can write their own processors and plug them at runtime into their deployed endpoints through the *addRecordProcessor* functionality of the ManagedEndpoint MBean. Every processor needs to implement the *org.jboss.wsf.spi.management.recording.RecordProcessor* interface. Then you can choose one of the two following options:

- Give you record processor an MBean interface declaring the manageable attributes: the recording system will plug your processor to the endpoint and register a management MBean for it using your interface. This allows you to create highly configurable custom processors. For an example of this development option, take a look at the *org.jboss.wsf.framework.management.recording.MemoryBufferRecorder*.
- Add your record processor to the managed endpoint as is: the recording system will plug it to the endpoint and register a standard management MBean for its basic processing configuration. The *org.jboss.wsf.framework.management.recording.LogRecorder* is an example of this development option.

A code snippet showing how to get the MBeanProxy instance which you can invoke MBean with can be found *here*.

13.4.2. Handler's position

Of course the recording handler's position in the handler chain influences the collected records. As a matter of fact some information may or may not be available at a given point of the handler chain execution. The standard endpoint configuration declares the RecordingServerHandler into the pre-handler-chain. Speaking of the native stack, this means for example that you'll get the invoked operation data and that decrypted messages will be recorded if using WS-Security, since the WS-Security handler runs in the post-handler-chain. Users might want to change the recording handler's position in the chain according to their requirements.

13.4.3. Multiple handlers

Records attributes include a record group ID that is meant to link records whose messages belong to the same message flow (a request-response for example). In order to set the right group ID to the records, the current ID is associated to the thread that is processing the endpoint invocation. This means that multiple related records can be linked together and extracted together from a processor.

For this reason, you might want to install multiple recording handlers into different points of the handler chain. For instance, it could make sense to record messages both before and after encryption/decryption when using WS-Security.

13.5. Future extensions

This paragraph covers eventual future extensions and/or idea JBossWS users may want to leverage for their own business.

13.5.1. Database recorder

The MemoryBufferRecorder provides interesting functionalities to query the collected records set. For obvious reasons, records are discarded once a given size of the buffer is reached.

A DB based recorder could be developed; it should work the same way the MemoryBufferRecorder does, except for records that should be saved through a given datasource. This will provide persistence of data even in case of application server reboot and webservice application redeploy. It will also allow records coming from different node of a cluster to be stored together. Finally this would allow administrators to directly query the database, which might be far more efficient.

13.5.2. Custom log writer

The idea of getting statistics from collected records could be further exploited getting custom logs from the records. These logs could be outputted by a custom processor in standard or proprietary formats allowing them to be imported into eventual third-party log processing tools which might offer complex/funky graphic or statistic functionalities and so on.

13.6. References

You might want to take a look at the *org.jboss.wsf.framework.management.recording* and *org.jboss.wsf.spi.management.recording* packages in the source code to better understand how all this works and can be used.

Part IV. Samples & Tutorials

Below you find few tutorials on WS-* technologies usage as well as a brief list of reference links and the list of supported JAX-WS annotations.

JBossWS-CXFWS-Addressingtutorial

Apache CXF [http://incubator.apache.org/cxf/] comes with support for *WS-Addressing 1.0* [http://www.w3.org/TR/ws-addr-core/]. In this sample we will show how to create client and endpoint communicating each other using this feature.

Creating WS-Addressing based service and client is very simple. User needs to create regular JAX-WS service and client first. The last step is to configure the addressing on both sides.

14.1. The Service

We will start with the following endpoint implementation (bottom-up approach):

```
@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 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**. Our next step is to generate JAX-WS artifacts and WSDL that will be part of endpoint archive.

14.2. Generating WSDL and JAX-WS Endpoint Artifacts

We will use **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
ls
 /home/username/wsa/cxf/wsprovide/generated/classes/org/jboss/test/ws/jaxws/
samples/wsa/jaxws
SayHello.class  SayHelloResponse.class

# java sources
ls
 /home/username/wsa/cxf/wsprovide/generated/src/org/jboss/test/ws/jaxws/
samples/wsa/jaxws
SayHello.java  SayHelloResponse.java

# contract artifacts
ls /home/username/wsa/cxf/wsprovide/generated/wsdl/
AddressingService.wsdl
```

All aforementioned generated artifacts will be part of endpoint archive. But before we will create the endpoint archive we need to reference generated WSDL from endpoint. To achieve that we will use **wsdlLocation** annotation attribute. Here's the updated endpoint implementation before packaging it 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!";
   }
}
```

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.class
     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 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>
```

14.3. 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;

```
import java.net.URL;import javax.xml.namespace.QName;import
 javax.xml.ws.Service;public final class SimpleServiceTestCase{
 private final String serviceURL = "http://" + getServerHost()
 + ":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();   }   }
```

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

14.4. Turning on WS-Addressing 1.0

In order to turn on WS-Addressing in JBossWS-CXF integration the last two steps are remaining:

- annotate service endpoint with @Addressing annotation
- modify client to configure WS-Addressing using JAX-WS webservice feature

14.4.1. 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!";
   }
}
```

As users can see we added JAX-WS 2.1 **Addressing** annotation to configure WS-Addressing. The next step is to repackage the endpoint archive to apply this change.

14.4.2. Updating Client Code to Configure WS-Addressing

Now we need to update client implementation as well 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://" + getServerHost() +
 ":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
      assertEquals("Hello World!", proxy.sayHello());
   }

}
```

And that's all. Now we have both JAX-WS client and endpoint communicating each other using WS-Addressing feature.

14.4.3. Leveraging WS-Addressing Policy

An option you can also evaluate to simplify both client and server deployment, is to let the server engine generate and publish the wsdl contract instead of using the one mentioned above: (please note the removal of wsdlLocation attribute in the @WebService annotation)

```
@WebService
(
   portName = "AddressingServicePort",
   serviceName = "AddressingService",
   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
{
     ...
}
```

This way the endpoint is published with a contract containing a WS-Addressing Policy that tells clients addressing needs to be on.

```
<wsp:Policy wsu:Id="AddressingServiceSoapBinding_WSAM_Addressing_Policy">
  <wsam:Addressing>
    <wsp:Policy/>
  </wsam:Addressing>
</wsp:Policy>
```

The client can then simply do as follows:

```
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);
// invoke method
```

No need for setting the AddressingFeature, the policy engine takes care of enabling WS-Addressing to match the policy advertised by the server.

14.5. Sample Sources

All sources from this tutorial are part of JBossWS-CXF testsuite.

JBossWS-CXFWS-ReliableMessagingtutorial

Apache CXF [http://incubator.apache.org/cxf/] comes with support for *WS-RM 1.0* [http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf]. In this sample we will show how to create client and endpoint communicating each other using WS-RM 1.0. The sample uses *WS-Policy* [http://www.w3.org/2006/07/ws-policy/] specification to configure WS-RM.

Creating the WS-RM based service and client is very simple. User needs to create regular JAX-WS service and client first. The last step is to configure WSRM.

15.1. The service

We will start with the following endpoint implementation (bottom-up approach):

```
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 WSDL.

15.2. Generating WSDL and JAX-WS Endpoint Artifacts

We will use **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
ls
 /home/username/wsrm/cxf/wsprovide/generated/classes/org/jboss/test/ws/
jaxws/samples/wsrm/service/jaxws/
Echo.class  EchoResponse.class  Ping.class

# java sources
ls
 /home/username/wsrm/cxf/wsprovide/generated/src/org/jboss/test/ws/jaxws/
samples/wsrm/service/jaxws/
Echo.java  EchoResponse.java  Ping.java

# contract artifacts
ls /home/username/wsrm/cxf/wsprovide/generated/wsdl/
SimpleService.wsdl
```

All aforementioned generated artifacts will be part of endpoint archive. But before we will create the endpoint archive we need to reference generated WSDL from endpoint. To achieve that we will use **wsdlLocation** annotation attribute. Here's the updated endpoint implementation before packaging it 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;
   }
}
```

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/
EchoResponse.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</
servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>SimpleService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

15.3. Generating JAX-WS Client Artifacts

Before we will write 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 above command generates the following artifacts:

```
# compiled classes
ls
 /home/username/wsrm/cxf/wsconsume/generated/classes/org/jboss/test/ws/
jaxws/samples/wsrm/generated/
Echo.class          ObjectFactory.class  Ping.class
 SimpleService_Service.class
EchoResponse.class  package-info.class   SimpleService.class
 SimpleService_SimpleServicePort_Client.class

# java sources
ls
 /home/username/wsrm/cxf/wsconsume/generated/src/org/jboss/test/ws/jaxws/
samples/wsrm/generated/
Echo.java          ObjectFactory.java  Ping.java
 SimpleService_Service.java
EchoResponse.java  package-info.java   SimpleService.java
 SimpleService_SimpleServicePort_Client.java
```

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

15.4. 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 client implementation but without WSRM in place. Our next goal is to turn on the WS-RM feature.

15.5. Turning on WS-RM 1.0

15.5.1. Extending WSDL Using WS-Policy

To activate WSRM on server side we need to extend the WSDL with WSRM and addressing policies. Here is how it looks like:

```
<?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">
<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">

      <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - -->
      <!-- Created WS-Policy with WSRM addressing assertions -->
      <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - --><wsp:Policy>
        <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>

      <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>
```

15.5.2. Basic WS-RM configuration

Once the endpoint wsdl is properly updated with the policies elements, the JBossWS-CXF stack is *automatically* able to detect the need for the WS-Policy engine to be used, both on client and server side, for enabling WS-Reliable Messaging.

The endpoint advertises RM capabilities through the published wsdl and the client is required to also enable WS-RM for successfully exchanging messages with the server.

The regular jaxws client above is enough if the user does not need to tune any specific detail of the RM subsystem (acknowledgment / retransmission intervals, thresholds, ...)

15.5.3. Advanced WS-RM configuration

When users want to have full control over the way WS-RM communication is established, the current CXF Bus needs to be properly configured. This can be done through a CXF Spring configuration.

15.5.3.1. Providing Client CXF Configuration

Next step is to create the client CXF configuration file that will be used by client. The following file was copied/pasted from CXF 2.0.5 ws_rm sample. It simply activates the WSRM protocol for CXF client. We will name this file **cxf.xml** in our sample. Here's the content of this file:

```
  <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>
```

And that's almost all. The client configuration needs to picked up by the client classloader; in order to achieve that the cxf.xml file has to be put in the META-INF directory of client jar. That jar should then be provided when setting the class loader.

Alternatively the bus configuration can also be read programmatically as follows:

15.5.3.2. Updating Client Code to Read Bus Configuration File

And here's the last piece 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);
   }

}
```

15.6. Sample Sources

All sources from this tutorial are part of JBossWS-CXF distribution.

JBossWS-CXFJMStransporttutorial

JBossWS-CXF supports JMS Transport to transfer SOAP messages. There is a testcase in the codebase to demonstrate this ability, available *here* [http://anonsvn.jboss.org/repos/jbossws/stack/cxf/tags/jbossws-cxf-3.4.0.CR2/modules/testsuite/cxf-spring-tests/src/test/java/org/jboss/test/ws/jaxws/samples/jmstransport/]. In this tutorial, we will use a wsdl first web service example to show you how to enable this feature in JBossWS.

16.1. WSDL

```
<wsdl:definitions name="OrganizationJMSEndpointService"
 targetNamespace="http://org.jboss.ws/samples/jmstransport"
 xmlns:jms="http://cxf.apache.org/transports/jms"
 xmlns:ns1="http://schemas.xmlsoap.org/wsdl/soap/http"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://org.jboss.ws/samples/jmstransport"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:message name="getContactInfoResponse">
    <wsdl:part name="return" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getContactInfo">
    <wsdl:part name="arg0" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="Organization">
    <wsdl:operation name="getContactInfo">
      <wsdl:input message="tns:getContactInfo" name="getContactInfo">
    </wsdl:input>
      <wsdl:output message="tns:getContactInfoResponse"
 name="getContactInfoResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HTTPSoapBinding" type="tns:Organization">
    <soap:binding style="rpc"
 transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getContactInfo">
      <soap:operation soapAction="" style="rpc"/>
      <wsdl:input name="getContactInfo">
        <soap:body namespace="http://org.jboss.ws/samples/jmstransport"
 use="literal"/>
```

```
        </wsdl:input>
        <wsdl:output name="getContactInfoResponse">
          <soap:body namespace="http://org.jboss.ws/samples/jmstransport"
   use="literal"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>

    <wsdl:binding name="JMSSoapBinding" type="tns:Organization">
      <soap:binding style="rpc"
   transport="http://cxf.apache.org/transports/jms"/>
      <wsdl:operation name="getContactInfo">
        <soap:operation soapAction="" style="rpc"/>
        <wsdl:input name="getContactInfo">
          <soap:body namespace="http://org.jboss.ws/samples/jmstransport"
   use="literal"/>
        </wsdl:input>
        <wsdl:output name="getContactInfoResponse">
          <soap:body namespace="http://org.jboss.ws/samples/jmstransport"
   use="literal"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>


    <wsdl:service name="OrganizationService">
      <wsdl:port binding='tns:HTTPSoapBinding' name='HttpEndpointPort'>
        <soap:address
   location='http://@jboss.bind.address@:8080/jaxws-samples-jmstransport'/>
      </wsdl:port>
      <wsdl:port binding="tns:JMSSoapBinding" name="JmsEndpointPort">
             <jms:address
                     destinationStyle="queue"
                     jndiConnectionFactoryName="ConnectionFactory"
                     jndiDestinationName="queue/RequestQueue"
                     jndiReplyDestinationName="queue/ResponseQueue">
                     >
                 </jms:address>
          </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

Apache CXF defines the jms wsdl extension, so the jms queue name or other information about jms in wsdl port can be parsed to send or receive jms message. Check this wiki page to see what jms attributes you can defined in WSDL. In this wsdl, we define two queues to send and receive the soap message. CXF uses JNDI to look up the jms ConnectionFactory, so we may also need to provide the JNDI properties as the following example :

```
<jms:address
    destinationStyle="queue"
    jndiConnectionFactoryName="ConnectionFactory"
    jndiDestinationName="queue/RequestQueue"
    jndiReplyDestinationName="queue/ResponseQueue"
    >
    <jms:JMSNamingProperty name="java.naming.factory.initial"
                           value="org.jnp.interfaces.NamingContextFactory"/>
    <jms:JMSNamingProperty name="java.naming.provider.url"
                           value="jnp://localhost:1099"/>
</jms:address>
```

16.2. Service Implementation

After generated code from this wsdl , we wrote two class to implement this interface for this two ports . We annotate the portName in annotation to tell web service stack which transport this service uses :

```
@WebService (serviceName="OrganizationService",
 portName="HttpEndpointPort",wsdlLocation =
 "WEB-INF/wsdl/jmstransport.wsdl",targetNamespace =
 "http://org.jboss.ws/samples/jmstransport", endpointInterface="org.jboss.test.ws.jaxws.samp
@SOAPBinding(style = SOAPBinding.Style.RPC)

public class OrganizationHttpEndpoint implements Organization
{
   @WebMethod
   public String getContactInfo(String organization)
   {
      return "The '" + organization + "' boss is currently out of office,
 please call again.";
   }
}
```

```
@WebService (serviceName="OrganizationService",portName="JmsEndpointPort",
 wsdlLocation = "WEB-INF/wsdl/jmstransport.wsdl", targetNamespace =
 "http://org.jboss.ws/samples/jmstransport", endpointInterface="org.jboss.test.ws.jaxws.samp
@SOAPBinding(style = SOAPBinding.Style.RPC)

public class OrganizationJmsEndpoint implements Organization
{
   @WebMethod
   public String getContactInfo(String organization)
   {
      return "The '" + organization + "' boss is currently out of office,
 please call again.";
   }
}
```

16.3. web.xml

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

  <servlet>
    <servlet-name>OrganizationService</servlet-name>

 <servlet-
class>org.jboss.test.ws.jaxws.samples.jmstransport.OrganizationHttpEndpoint</
servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

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

It is almost the same as the usual web.xml to deploy a web service except the <load-on-startup> servlet initializeparameter. This is for jms service start ready when deployment, no need to wait until the first servlet request to start the jms endpoint.

16.4. jbossws-cxf.xml

In addition to web.xml, the jbossws-cxf.xml is needed to actually pass in cxf to start this two port.

```
<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:jms="http://cxf.apache.org/transports/jms"
    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://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
  http://cxf.apache.org/transports/jms
 http://cxf.apache.org/schemas/configuration/jms.xsd'>

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

    <jaxws:endpoint id='SOAPQueryService'

 implementor='org.jboss.test.ws.jaxws.samples.jmstransport.OrganizationHttpEndpoint'
               >
        <jaxws:invoker>
            <bean class='org.jboss.wsf.stack.cxf.InvokerJSE'/>
        </jaxws:invoker>
    </jaxws:endpoint>

    <jaxws:endpoint id='JMSQueryService'

 implementor='org.jboss.test.ws.jaxws.samples.jmstransport.OrganizationJmsEndpoint'
               transportId="http://cxf.apache.org/transports/jms">
    </jaxws:endpoint>
</beans>
```

Note: the import resource is the JmsTransportFactory configuration . It is required to jms transport enablement .

Below gives the war file directory structure to make it more clear what inside :

```
|-- jmstransport-sample.war
`-- WEB-INF
    |-- classes
    |   `-- org
    |       `-- jboss
    |           `-- test
    |               `-- ws
    |                   `-- jaxws
    |                       `-- samples
    |                           `-- jmstransport
    |                               |--
 JMSTransportTestCase$ResponseListener.class
    |                               |-- JMSTransportTestCase.class
    |                               |-- Organization.class
    |                               |-- OrganizationHttpEndpoint.class
    |                               `-- OrganizationJmsEndpoint.class
    |-- jboss-web.xml
    |-- jbossws-cxf.xml
    |-- web.xml
    `-- wsdl
        `-- jmstransport.wsdl
```

JBossWS-JAX-WSAnnotations

17.1. JAX-WS Annotations

For details, see *JSR-224 - Java API for XML-Based Web Services (JAX-WS) 2.0* [http://www.jcp.org/en/jsr/detail?id=224]

17.1.1. javax.xml.ws.ServiceMode

The ServiceMode annotation is used to specify the mode for a provider class, i.e. whether a provider wants to have access to protocol message payloads (e.g. a SOAP body) or the entire protocol messages (e.g. a SOAP envelope).

17.1.2. javax.xml.ws.WebFault

The WebFault annotation is used when mapping WSDL faults to Java exceptions, see section 2.5. It is used to capture the name of the fault element used when marshalling the JAXB type generated from the global element referenced by the WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults.

17.1.3. javax.xml.ws.RequestWrapper

The RequestWrapper annotation is applied to the methods of an SEI. It is used to capture the JAXB generated request wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of localName element is the operationName as defined in WebMethod annotation and the default value for the targetNamespace element is the target namespace of the SEI.When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the className element is required in this case.

17.1.4. javax.xml.ws.ResponseWrapper

The ResponseWrapper annotation is applied to the methods of an SEI. It is used to capture the JAXB generated response wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of the localName element is the operationName as defined in the WebMethod appended with ”Response” and the default value of the targetNamespace element is the target namespace of the SEI. When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the className element is required in this case.

17.1.5. javax.xml.ws.WebServiceClient

The WebServiceClient annotation is specified on a generated service class (see 2.7). It is used to associate a class with a specific Web service, identify by a URL to a WSDL document and the qualified name of a wsdl:service element.

17.1.6. javax.xml.ws.WebEndpoint

The WebEndpoint annotation is specified on the getPortName() methods of a generated service class (see 2.7). It is used to associate a get method with a specific wsdl:port, identified by its local name (a NCName).

17.1.7. javax.xml.ws.WebServiceProvider

The WebServiceProvider annotation is specified on classes that implement a strongly typed javax-.xml.ws.Provider. It is used to declare that a class that satisfies the requirements for a provider (see 5.1) does indeed define a Web service endpoint, much like the WebService annotation does for SEI-based endpoints.

The WebServiceProvider and WebService annotations are mutually exclusive.

17.1.8. javax.xml.ws.BindingType

The BindingType annotation is applied to an endpoint implementation class. It specifies the binding to use when publishing an endpoint of this type.

The default binding for an endpoint is the SOAP 1.1/HTTP one

17.1.9. javax.xml.ws.WebServiceRef

The WebServiceRef annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the javax.annotation.Resource annotation in JSR-250 [32]. The WebServiceRef annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification [33].

17.1.10. javax.xml.ws.WebServiceRefs

The WebServiceRefs annotation is used to declare multiple references to Web services on a single class. It is necessary to work around the limition against specifying repeated annotations of the same type on any given class, which prevents listing multiple javax.ws.WebServiceRef annotations one after the other. This annotation follows the resource pattern exemplified by the javax.annotation.Resources annotation in JSR-250.

Since no name and type can be inferred in this case, each WebServiceRef annotation inside a WebServiceRefs MUST contain name and type elements with non-default values. The WebServiceRef annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification.

17.1.11. javax.xml.ws.Action

The Action annotation is applied to the methods of a SEI. It used to generate the wsa:Action on wsdl:input and wsdl:output of each wsdl:operation mapped from the annotated methods.

17.1.12. javax.xml.ws.FaultAction

The FaultAction annotation is used within the Action annotation to generate the wsa:Action element on the wsdl:fault element of each wsdl:operation mapped from the annotated methods.

17.1.13. Annotations Defined by JSR-181

JSR-181 defines the syntax and semantics of Java Web Service (JWS) metadata and default values.

For details, see *JSR 181 - Web Services Metadata for the Java Platform* [http://jcp.org/en/jsr/detail?id=181]

17.1.13.1. javax.jws.WebService

Marks a Java class as implementing a Web Service, or a Java interface as defining a Web Service interface.

17.1.13.2. javax.jws.WebMethod

Customizes a method that is exposed as a Web Service operation.

17.1.13.3. javax.jws.OneWay

Indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method. A JSR-181 processor is REQUIRED to report an error if an operation marked @Oneway has a return value, declares any checked exceptions or has any INOUT or OUT parameters.

17.1.13.4. javax.jws.WebParam

Customizes the mapping of an individual parameter to a Web Service message part and XML element.

17.1.13.5. javax.jws.WebResult

Customizes the mapping of the return value to a WSDL part and XML element.

17.1.13.6. javax.jws.SOAPBinding

Specifies the mapping of the Web Service onto the SOAP message protocol.

The SOAPBinding annotation has a target of TYPE and METHOD. The annotation may be placed on a method if and only if the SOAPBinding.style is DOCUMENT. Implementations MUST report an error if the SOAPBinding annotation is placed on a method with a SOAPBinding.style of RPC. Methods that do not have a SOAPBinding annotation accept the SOAPBinding behavior defined on the type.

17.1.13.7. javax.jws.HandlerChain

The @HandlerChain annotation associates the Web Service with an externally defined handler chain.

It is an error to combine this annotation with the @SOAPMessageHandlers annotation.

The @HandlerChain annotation MAY be present on the endpoint interface and service implementation bean. The service implementation bean's @HandlerChain is used if @HandlerChain is present on both.

The @HandlerChain annotation MAY be specified on the type only. The annotation target includes METHOD and FIELD for use by JAX-WS-2.0.