PROGRESS SOFTWARE

Artix® ESB

Developing Artix® Applications with JAX-WS

Version 5.5
December 2008

Developing Artix® Applications with JAX-WS

Version 5.5

Published 19 Feb 2009

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

What is Covered in This Book

This book describes how to use the JAX-WS 2.1 APIs to develop applications with Artix ESB. It also describes how to develop RESTful services as POJOs.

Who Should Read This Book

This book is intended for developers using Artix ESB. It assumes that you have a good understanding of the following:

- general programming concepts.
- general SOA concepts.
- Java 5.
- the runtime environment into which you are deploying services.

How to Use This Book

This book is organized into the following chapters:

- Part I on page 23 describes how to develop SOA applications with out using WSDL documents.
- Part II on page 59 describes how to develop SOA applications using a WSDL document as a starting point.
- Part V on page 131 describes how XML Schema data definitions are mapped into Java for use in developing services.
- *Publishing a Service* on page 117 describes how to publish a service using a stand alone Java application.
- *Developing Asynchronous Applications* on page 273 describes how to develop service consumers that can interact with service providers asynchronously.
- *Using Raw XML Messages* on page 289 describes how to use the `Dispatch` and `Provider` interfaces to develop applications that work with raw XML instead of JAXB object.
- *Working with Contexts* on page 313 describes how to manipulate message and transport properties programaticaly.
- Part III on page 83 describes how to use the Artix ESB API's annotations to create RESTful services.

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see Using the Artix ESB Library[1].

[1] http://www.iona.com/support/docs/artix/5.5/library_intro/index.htm

Part I. Starting from Java Code

One of the advantages of JAX-WS is that it does not require you to start with a WSDL document that defines their service. You can start with Java code that defines the features you want to expose as services. The code may be a class, or classes, from a legacy application that is being upgraded. It may also be a class that is currently being used as part of a non-distributed application and implements features that you want to use in a distributed manner. You annotate the Java code and generate a WSDL document from the annotated code. If you do not wish to work with WSDL at all, you can create the entire application without ever generating WSDL.

Bottom-Up Service Development

There are many instances where you have Java code that already implements a set of functionality that you want to expose as part of a service oriented application. You may also simply want to avoid using WSDL to define your interface. Using JAX-WS annotations, you can add the information required to service enable a Java class. You can also create a Service Endpoint Interface (SEI) that can be used in place of a WSDL contract. If you want a WSDL contract, Artix ESB provides tools to generate a contract from annotated Java code.

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

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

Tip

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

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

Tip

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

4. Publish the service as a service provider.

Creating the SEI

Overview

The *service endpoint interface* (SEI) is the piece of Java code that is shared between a service implementation and the consumers that make requests on that service. The SEI defines the methods implemented by the service and provides details about how the service will be exposed as an endpoint. When starting with a WSDL contract, the SEI is generated by the code generators. However, when starting from Java, it is the developer's responsibility to create the SEI.

There are two basic patterns for creating an SEI:

- Green field development — In this pattern, you are developing a new service without any existing Java code or WSDL. It is best to start by creating the SEI. You can then distribute the SEI to any developers that are responsible for implementing the service providers and consumers that use the SEI.

Note

The recommended way to do green field service development is to start by creating a WSDL contract that defines the service and its interfaces. See Part II: Starting from a WSDL Contract on page 59.

- Service enablement — In this pattern, you typically have an existing set of functionality that is implemented as a Java class, and you want to service enable it. This means that you must do two things:

  1. Create an SEI that contains **only** the operations that are going to be exposed as part of the service.
  2. Modify the existing Java class so that it implements the SEI.

Note

Although you can add the JAX-WS annotations to a Java class, it is not recommended.

Writing the interface

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

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

Tip

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

Example 1 on page 29 shows a simple SEI for a stock updating service.

Example 1. Simple SEI

```
package com.iona.demo;

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

Implementing the interface

Because the SEI is a standard Java interface, the class that implements it is a standard Java class. If you start with a Java class you must modify it to implement the interface. If you start with the SEI, the implementation class implements the SEI.

Example 2 on page 30 shows a class for implementing the interface in Example 1 on page 29.

Example 2. Simple Implementation Class

```
package com.iona.demo;

import java.util.*;

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

[1] `Board` is an assumed class whose implementation is left to the reader.

Annotating the Code

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

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

Tip

Most of the annotations have sensible defaults and it is not necessary to provide values for them. However, the more information you provide in the annotations, the better your service definition is specified. A well-specified service definition increases the likelihood that all parts of a distributed application will work together.

Required Annotations

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

The @WebService annotation

The `@WebService` annotation is defined by the `javax.jws.WebService` interface and it is placed on an interface or a class that is intended to be used as a service. `@WebService` has the properties described in Table 1 on page 32

Table 1. `@WebService` Properties

Property	Description
name	Specifies the name of the service interface. This property is mapped to the `name` attribute of the `wsdl:portType` element that defines the service's interface in a WSDL contract. The default is to append `PortType` to the name of the implementation class. [a]
targetNamespace	Specifies the target namespace where the service is defined. If this property is not specified, the target namespace is derived from the package name.
serviceName	Specifies the name of the published service. This property is mapped to the `name` attribute of the `wsdl:service` element that defines the published service. The default is to use the name of the service's implementation class. [a]
wsdlLocation	Specifies the URL where the service's WSDL contract is stored. This must be specified using a relative URL. The default is the URL where the service is deployed.
endpointInterface	Specifies the full name of the SEI that the implementation class implements. This property is only specified when the attribute is used on a service implementation class.
portName	Specifies the name of the endpoint at which the service is published. This property is mapped to the `name` attribute of the `wsdl:port` element that specifies the endpoint details for a published service. The default is the append `Port` to the name of the service's implementation class.[a]

[a]When you generate WSDL from an SEI the interface's name is used in place of the implementation class' name.

Tip

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

Annotating the SEI

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

Example 3 on page 33 shows the interface defined in Example 1 on page 29 with the `@WebService` annotation.

Example 3. Interface with the `@WebService` Annotation

```
package com.iona.demo;

import javax.jws.*;

@WebService(name="quoteUpdater", ❶
            targetNamespace="http:\\demos.iona.com", ❷
         serviceName="updateQuoteService", ❸
            wsdlLocation="http:\\demos.iona.com\quoteExampleService?wsdl", ❹
            portName="updateQuotePort") ❺
public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

The `@WebService` annotation in Example 3 on page 33 does the following:

❶ Specifies that the value of the `name` attribute of the `wsdl:portType` element defining the service interface is `quoteUpdater`.

❷ Specifies that the target namespace of the service is `http:\\demos.iona.com`.

❸ Specifies that the value of the `name` of the `wsdl:service` element defining the published service is `updateQuoteService`.

❹ Specifies that the service will publish its WSDL contract at `http:\\demos.iona.com\quoteExampleService?wsdl`.

❺ Specifies that the value of the `name` attribute of the `wsdl:port` element defining the endpoint exposing the service is `updateQuotePort`.

Annotating the service implementation

In addition to annotating the SEI with the `@WebService` annotation, you also must annotate the service implementation class with the `@WebService` annotation. When adding the annotation to the service implementation class you only need to specify the endpointInterface property. As shown in Example 4 on page 34 the property must be set to the full name of the SEI.

Example 4. Annotated Service Implementation Class

```
package org.eric.demo;

import javax.jws.*;

@WebService(endpointInterface="com.iona.demo.quoteReporter")
public class stockQuoteReporter implements quoteReporter
{
public Quote getQuote(String ticker)
  {
  ...
  }
}
```

Optional Annotations

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

Tip

The more details you provide in the SEI the easier it is for developers to implement applications that can use the functionality it defines. It also makes the WSDL documents generated by the tools more specific.

Defining the Binding Properties with Annotations

Overview

If you are using a SOAP binding for your service, you can use JAX-WS annotations to specify a number of the bindings properties. These properties correspond directly to the properties you can specify in a service's WSDL contract. Some of the settings, such as the parameter style, can restrict how you implement a method. These settings can also effect which annotations can be used when annotating method parameters.

The @SOAPBinding annotation

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

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

Table 2 on page 36 shows the properties for the `@SOAPBinding` annotation.

Table 2. `@SOAPBinding` Properties

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

[a] `Use.ENCODED` is not currently supported.

[b] If you set the style to `RPC` you must use the `WRAPPED` parameter style.

Document bare style parameters

Document bare style is the most direct mapping between Java code and the resulting XML representation of the service. When using this style, the schema types are generated directly from the input and output parameters defined in the operation's parameter list.

You specify you want to use bare document\literal style by using the `@SOAPBinding` annotation with its style property set to `Style.DOCUMENT`, and its parameterStyle property set to `ParameterStyle.BARE`.

To ensure that an operation does not violate the restrictions of using document style when using bare parameters, your operations must adhere to the following conditions:

- The operation must have no more than one input or input/output parameter.

- If the operation has a return type other than void, it must not have any output or input/output parameters.
- If the operation has a return type of void, it must have no more than one output or input/output parameter.

Note

Any parameters that are placed in the SOAP header using the `@WebParam` annotation or the `@WebResult` annotation are not counted against the number of allowed parameters.

Document wrapped parameters

Document wrapped style allows a more RPC like mapping between the Java code and the resulting XML representation of the service. When using this style, the parameters in the method's parameter list are wrapped into a single element by the binding. The disadvantage of this is that it introduces an extra-layer of indirection between the Java implementation and how the messages are placed on the wire.

To specify that you want to use wrapped document\literal style use the `@SOAPBinding` annotation with its style property set to `Style.DOCUMENT`, and its parameterStyle property set to `ParameterStyle.WRAPPED`.

You have some control over how the wrappers are generated by using the `@RequestWrapper` annotation and the `@ResponseWrapper` annotation.

Example

Example 5 on page 37 shows an SEI that uses document bare SOAP messages.

Example 5. Specifying a Document Bare SOAP Binding with the SOAP Binding Annotation

```
package org.eric.demo;

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

@WebService(name="quoteReporter")
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
public interface quoteReporter
{
```

```
  ...
}
```

Defining Operation Properties with Annotations

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

- What the exchanged messages look like in XML
- If the message can be optimized as a one way message
- The namespaces where the messages are defined

The @WebMethod annotation

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

Table 3 on page 38 describes the properties of the `@WebMethod` annotation.

Table 3. `@WebMethod` Properties

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

The @RequestWrapper annotation

The `@RequestWrapper` annotation is defined by the `javax.xml.ws.RequestWrapper` interface. It is placed on the methods in the SEI. The `@RequestWrapper` annotation specifies the Java class implementing the wrapper bean for the method parameters of the request

message starting a message exchange. It also specifies the element names, and namespaces, used by the runtime when marshalling and unmarshalling the request messages.

Table 4 on page 39 describes the properties of the `@RequestWrapper` annotation.

Table 4. `@RequestWrapper` Properties

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

Tip

Only the className property is required.

Important

If the method is also annotated with the `@SOAPBinding` annotation, and its parameterStyle property is set to `ParameterStyle.BARE`, this annotation is ignored.

The @ResponseWrapper annotation

The `@ResponseWrapper` annotation is defined by the `javax.xml.ws.ResponseWrapper` interface. It is placed on the methods in the SEI. The `@ResponseWrapper` specifies the Java class implementing the wrapper bean for the method parameters in the response message in the message exchange. It also specifies the element names, and namespaces, used by the runtime when marshaling and unmarshalling the response messages.

Table 5 on page 40 describes the properties of the `@ResponseWrapper` annotation.

Table 5. `@ResponseWrapper` Properties

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

Tip

Only the className property is required.

Important

If the method is also annotated with the `@SOAPBinding` annotation and its parameterStyle property is set to `ParameterStyle.BARE`, this annotation is ignored.

The @WebFault annotation

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

Table 6 on page 41 describes the properties of the `@WebFault` annotation.

Table 6. `@WebFault` Properties

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

Important

The name property is required.

The @Oneway annotation

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

This annotation can only be used on methods that meet the following criteria:

- They return void
- They have no parameters that implement the `Holder` interface
- They do not throw any exceptions that can be passed back to a consumer

Example

Example 6 on page 41 shows an SEI with its methods annotated.

Example 6. SEI with Annotated Methods

```
package com.iona.demo;

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

@WebService(name="quoteReporter")
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
```

```
  @RequestWrapper(targetNamespace="http://demo.iona.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.iona.com/types",
                   className="org.eric.demo.Quote")
  public Quote getQuote(String ticker);
}
```

Defining Parameter Properties with Annotations

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

The @WebParam annotation

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

Table 7 on page 42 describes the properties of the `@WebParam` annotation.

Table 7. `@WebParam` Properties

Property	Values	Description
name		Specifies the name of the parameter as it appears in the generated WSDL document. For RPC bindings, this is the name of the `wsdl:part` representing the parameter. For document bindings, this is the local name of the XML element representing the parameter. Per the JAX-WS specification, the default is `argN`, where *N* is replaced with the zero-based argument index (i.e., arg0, arg1, etc.).
targetNamespace		Specifies the namespace for the parameter. It is only used with document bindings where the parameter maps to an XML element. The default is to use the service's namespace.

mode	`Mode.IN` (default)[a]
	`Mode.OUT`
	`Mode.INOUT`
		Specifies the direction of the parameter.

Property	Values	Description
header	`false` (default) `true`	Specifies if the parameter is passed as part of the SOAP header.
partName		Specifies the value of the `name` attribute of the `wsdl:part` element for the parameter. This property is used for document style SOAP bindings.

[a]Any parameter that implements the `Holder` interface is mapped to `Mode.INOUT` by default.

The @WebResult annotation

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

Table 8 on page 43 describes the properties of the `@WebResult` annotation.

Table 8. `@WebResult` Properties

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

Example

Example 7 on page 44 shows an SEI that is fully annotated.

Example 7. Fully Annotated SEI

```
package com.iona.demo;

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

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

Generating WSDL

Using the command line tool

Once your code is annotated, you can generate a WSDL contract for your service using the **artix java2ws** command's `-wsdl` flag. For a detailed listing of options for the **artix java2ws** command see artix java2ws in the *Artix® ESB Command Reference*.

Using Ant

To call the WSDL generator from Ant use the **java** task to execute the `org.apache.cxf.tools.java2ws.JavaToWS` class and pass `-wsdl` as one of its arguments. Example 8 on page 45 shows a sample Ant target that calls the WSDL generator.

Example 8. Calling the WSDL Generator from Ant

```
<project name="java2ws" basedir=".">
   <property name="fsf.home" location ="/usr/myapps/fsf-trunk"/>
   <property name="build.classes.dir" location ="${basedir}/build/classes"/>

   <path id="fsf.classpath">
      <pathelement location="${build.classes.dir}"/>
      <fileset dir="${fsf.home}/lib">
         <include name="*.jar"/>
      </fileset>
   </path>

   <target name="WSDLGen">
      <java classname="org.apache.cxf.tools.java2ws.JavaToWS" fork="true">
         <arg value="-wsdl"/>
         <arg value="service.Greeter"/>
         <classpath>
            <path refid="fsf.classpath"/>
         </classpath>
      </java>
   </target>
</project>
```

Important

You must set the Java task's fork to true.

Example

Example 9 on page 46 shows the WSDL contract that is generated for the SEI shown in Example 7 on page 44.

Example 9. Generated WSDL from an SEI

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

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

Developing a Consumer Without a WSDL Contract

You do not need a WSDL contract to develop a service consumer. You can create a service consumer from an annotated SEI. Along with the SEI you need to know the address at which the endpoint exposing the service is published, the QName of the service element that defines the endpoint exposing the service, and the QName of the port element defining the endpoint on which your consumer makes requests. This information can be specified in the SEI's annotations or provided separately.

To create a consumer without a WSDL contract you must do the following:

1. Create a `Service` object for the service on which the consumer will invoke operations.
2. Add a port to the `Service` object.
3. Get a proxy for the service using the `Service` object's `getPort()` method.
4. Implement the consumer's business logic.

Creating a Service Object

Overview

The `javax.xml.ws.Service` class represents the `wsdl:service` element which contains the definition of all of the endpoints that expose a service. As such, it provides methods that allow you to get endpoints, defined by `wsdl:port` elements, that are proxies for making remote invocations on a service.

Note

The `Service` class provides the abstractions that allow the client code to work with Java types as opposed to working with XML documents.

The create() methods

The `Service` class has two static `create()` methods that can be used to create a new `Service` object. As shown in Example 10 on page 50, both of the `create()` methods take the QName of the `wsdl:service` element the `Service` object will represent, and one takes a URI specifying the location of the WSDL contract.

Tip

All services publish their WSDL contracts. For SOAP/HTTP services the URI is usually the URI for the service appended with `?wsdl`.

Example 10. `Service create()` Methods

```
public static Service create(URL wsdlLocation,
                             QName serviceName)
  throws WebServiceException;

public static Service create(QName serviceName)
  throws WebServiceException;
```

The value of the *`serviceName`* parameter is a QName. The value of its namespace part is the target namespace of the service. The service's target namespace is specified in the targetNamespace property of the `@WebService` annotation. The value of the QName's local part is the value of `wsdl:service`

element's `name` attribute. You can determine this value in one of the following ways:

1. It is specified in the serviceName property of the `@WebService` annotation.
2. You append `Service` to the value of the name property of the `@WebService` annotation.
3. You append `Service` to the name of the SEI.

Example

Example 11 on page 51 shows code for creating a `Service` object for the SEI shown in Example 7 on page 44.

Example 11. Creating a `Service` Object

```
package com.iona.demo;

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

public class Client
{
public static void main(String args[])
  {
❶    QName serviceName = new QName("http://demo.iona.com", "stockQuoteReporter");
❷    Service s = Service.create(serviceName);
   ...
  }
}
```

The code in Example 11 on page 51 does the following:

❶ Builds the QName for the service using the targetNamespace property and the name property of the `@WebService` annotation.

❷ Calls the single parameter `create()` method to create a new `Service` object.

Note

Using the single parameter `create()` frees you from having any dependencies on accessing a WSDL contract.

Adding a Port to a Service

The endpoint information for a service is defined in a `wsdl:port` element, and the `Service` object creates a proxy instance for each of the endpoints defined in a WSDL contract, if one is specified. If you do not specify a WSDL contract when you create your `Service` object, the `Service` object has no information about the endpoints that implement your service, and therefore cannot create any proxy instances. In this case, you must provide the `Service` object with the information needed to represent a `wsdl:port` element using the `addPort()` method.

The addPort() method

The `Service` class defines an `addPort()` method, shown in Example 12 on page 52, that is used in cases where there is no WSDL contract available to the consumer implementation. The `addPort()` method allows you to give a `Service` object the information, which is typically stored in a `wsdl:port` element, necessary to create a proxy for a service implementation.

Example 12. The `addPort()` Method

```
void addPort(QName portName,
             String bindingId,
             String endpointAddress)
  throws WebServiceException;
```

The value of the *`portName`* is a QName. The value of its namespace part is the target namespace of the service. The service's target namespace is specified in the targetNamespace property of the `@WebService` annotation. The value of the QName's local part is the value of `wsdl:port` element's `name` attribute. You can determine this value in one of the following ways:

1. Specify it in the portName property of the `@WebService` annotation.
2. Append `Port` to the value of the name property of the `@WebService` annotation.
3. Append `Port` to the name of the SEI.

The value of the *bindingId* parameter is a string that uniquely identifies the type of binding used by the endpoint. For a SOAP binding you use the standard SOAP namespace: http://schemas.xmlsoap.org/soap/. If the endpoint is not using a SOAP binding, the value of the *bindingId* parameter is determined by the binding developer.

The value of the *endpointAddress* parameter is the address where the endpoint is published. For a SOAP/HTTP endpoint, the address is an HTTP address. Transports other than HTTP use different address schemes.

Example

Example 13 on page 53 shows code for adding a port to the `Service` object created in Example 11 on page 51.

Example 13. Adding a Port to a `Service` Object

```
package com.fusesource.demo;

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

public class Client
{
public static void main(String args[])
  {
    ...
❶    QName portName = new QName("http://demo.fusesource.com", "stockQuoteReporterPort");
❷    s.addPort(portName,
❸             "http://schemas.xmlsoap.org/soap/",
❹             "http://localhost:9000/StockQuote");
    ...
  }
}
```

The code in Example 13 on page 53 does the following:

❶ Creates the QName for the *portName* parameter.

❷ Calls the `addPort()` method.

❸ Specifies that the endpoint uses a SOAP binding.

❹ Specifies the address where the endpoint is published.

Getting a Proxy for an Endpoint

A service proxy is an object that provides all of the methods exposed by a remote service and handles all of the details required to make the remote invocations. The `Service` object provides service proxies for all of the endpoints it is aware of through the `getPort()` method. Once you have a service proxy, you can invoke its methods. The proxy forwards the invocation to the remote service endpoint using the connection details specified in the service's contract.

The getPort() method

The `getPort()` method, shown in Example 14 on page 54, returns a service proxy for the specified endpoint. The returned proxy is of the same class as the SEI.

Example 14. The `getPort()` Method

```
public <T> T getPort(QName portName,
                     Class<T> serviceEndpointInterface)
  throws WebServiceException;
```

The value of the *`portName`* parameter is a QName that identifies the `wsdl:port` element that defines the endpoint for which the proxy is created. The value of the *`serviceEndpointInterface`* parameter is the fully qualified name of the SEI.

Tip

When you are working without a WSDL contract the value of the *`portName`* parameter is typically the same as the value used for the *`portName`* parameter when calling `addPort()`.

Example

Example 15 on page 54 shows code for getting a service proxy for the endpoint added in Example 13 on page 53.

Example 15. Getting a Service Proxy

```
package com.fusesource.demo;

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

```
public class Client
{
public static void main(String args[])
  {
    ...
    quoteReporter proxy = s.getPort(portName, quoteReporter.class);
    ...
  }
}
```

Implementing the Consumer's Business Logic

Overview

Once you instantiate a service proxy for a remote endpoint, you can invoke its methods as if it were a local object. The calls block until the remote method completes.

Note

If a method is annotated with the `@OneWay` annotation, the call returns immediately.

Example

Example 16 on page 56 shows a consumer for the service defined in Example 7 on page 44.

Example 16. Consumer Implemented without a WSDL Contract

```
package com.fusesource.demo;

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

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

    QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
❷  s.addPort(portName, "http://schemas.xmlsoap.org/soap/", "http://localhost:9000/EricStock
Quote");

❸  quoteReporter proxy = s.getPort(portName, quoteReporter.class);

❹  Quote quote = proxy.getQuote("ALPHA");
    System.out.println("Stock "+quote.getID()+" is worth "+quote.getVal()+" as of
"+quote.getTime());
  }
}
```

The code in Example 16 on page 56 does the following:

❶ Creates a `Service` object.

❷ Adds an endpoint definition to the `Service` object.

❸ Gets a service proxy from the `Service` object.

❹ Invokes an operation on the service proxy.

Part II. Starting from a WSDL Contract

The recommended way to develop service-oriented applications is to start from a WSDL contract. The WSDL contract provides an implementation neutral way of defining the operations a service exposes and the data that is exchanged with the service. Artix ESB provides tools to generate JAX-WS annotated starting point code from a WSDL contract. The code generators create all of the classes needed to implement any abstract data types defined in the contract. This approach simplifies the development of widely distributed applications.

A Starting Point WSDL Contract

Example 17 on page 63 shows the HelloWorld WSDL contract. This contract defines a single interface, `Greeter`, in the `wsdl:portType` element. The contract also defines the endpoint which will implement the service in the `wsdl:port` element.

Example 17. HelloWorld WSDL Contract

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"
                  xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"><element name="sayHi">
      <element name="sayHi">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="sayHiResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMe">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeResponse">
        <complexType>
          <sequence>
```

```
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeOneWay">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="pingMe">
        <complexType/>
      </element>
      <element name="pingMeResponse">
        <complexType/>
      </element>
      <element name="faultDetail">
        <complexType>
          <sequence>
            <element name="minor" type="short"/>
            <element name="major" type="short"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>

  <wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
  </wsdl:message>
  <wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
  </wsdl:message>
  <wsdl:message name="greetMeRequest">
    <wsdl:part element="x1:greetMe" name="in"/>
  </wsdl:message>
  <wsdl:message name="greetMeResponse">
    <wsdl:part element="x1:greetMeResponse" name="out"/>
  </wsdl:message>
  <wsdl:message name="greetMeOneWayRequest">
    <wsdl:part element="x1:greetMeOneWay" name="in"/>
  </wsdl:message>
  <wsdl:message name="pingMeRequest">
    <wsdl:part name="in" element="x1:pingMe"/>
  </wsdl:message>
  <wsdl:message name="pingMeResponse">
    <wsdl:part name="out" element="x1:pingMeResponse"/>
  </wsdl:message>
```

```
  <wsdl:message name="pingMeFault">
    <wsdl:part name="faultDetail" element="x1:faultDetail"/>
  </wsdl:message>

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

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

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

❹     <wsdl:operation name="pingMe">
       <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
       <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
       <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
     </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
    ...
  </wsdl:binding>

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

The `Greeter` interface defined in Example 17 on page 63 defines the following operations:

❶ sayHi — Has a single output parameter, of xsd:string.

❷ greetMe — Has an input parameter, of xsd:string, and an output parameter, of xsd:string.

❸ greetMeOneWay — Has a single input parameter, of xsd:string. Because this operation has no output parameters, it is optimized to be a oneway invocation (that is, the consumer does not wait for a response from the server).

❹ pingMe — Has no input parameters and no output parameters, but it can raise a fault exception.

Top-Down Service Development

In the top-down method of developing a service provider you start from a WSDL document that defines the operations and methods the service provider will implement. Using the WSDL document, you generate starting point code for the service provider. Adding the business logic to the generated code is done using normal Java programming APIs.

Once you have a WSDL document, the process for developing a JAX-WS service provider is as follows:

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

Generating the Starting Point Code

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

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

Running the code generator

You can generate the code needed to develop your service provider using the following command:

```
artix wsdl2java -ant -impl -server -d outputDir myService.wsdl
```

This command does the following:

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

For a complete list of the arguments for **artix wsdl2java** see artix wsdl2java in the *Artix® ESB Command Reference*.

Generating code from Ant

If you are using Apache Ant as your build system, you can call the code generator using Ant's **java** task as shown in Example 18 on page 69.

Example 18. Generating Service Starting Point Code from Ant

```
<project name="myProject" basedir=".">
  <property name="fsf.home" location ="InstallDir"/>

   <path id="fsf.classpath">
     <fileset dir="${fsf.home}/lib">
         <include name="*.jar"/>
      </fileset>
   </path>

  <target  name="ServiceGen">
    <java classname="org.apache.cxf.tools.wsdlto.WSDLToJava" fork="true">
      <arg value="-ant"/>
      <arg value="-impl"/>
      <arg value="-server"/>
      <arg value="-d"/>
      <arg value="outputDir"/>
      <arg value="myService.wsdl"/>
      <classpath>
        <path refid="fsf.classpath"/>
      </classpath>
    </java>
    ...
  </target>
  ...
</project>
```

The command line options are passed to the code generator using the task's `arg` element. Arguments that require two strings, such as `-d`, must be split into two `arg` elements.

Generated code

Table 9 on page 70 describes the files generated for creating a service provider.

Table 9. Generated Classes for a Service Provider

File	Description
portTypeName.java	The SEI. This file contains the interface your service provider implements. You should not edit this file.
serviceName.java	The endpoint. This file contains the Java class consumers use to make requests on the service.
*portTypeName*Impl.java	The skeleton implementation class. Modify this file to build your service provider.
*portTypeName*Server.java	A basic server mainline that allows you to deploy your service provider as a stand alone process. For more information see *Publishing a Service* on page 117.

In addition, **artix wsdl2java** will generate Java classes for all of the types defined in the WSDL contract.

Generated packages

The generated code is placed into packages based on the namespaces used in the WSDL contract. The classes generated to support the service (based on the `wsdl:portType` element, the `wsdl:service` element, and the `wsdl:port` element) are placed in a package based on the target namespace of the WSDL contract. The classes generated to implement the types defined in the `types` element of the contract are placed in a package based on the `targetNamespace` attribute of the `types` element.

The mapping algorithm is as follows:

1. The leading `http://` or `urn://` are stripped off the namespace.
2. If the first string in the namespace is a valid Internet domain, for example it ends in `.com` or `.gov`, then the leading `www.` is stripped off the string, and the two remaining components are flipped.
3. If the final string in the namespace ends with a file extension of the pattern `.xxx` or `.xx`, then the extension is stripped.
4. The remaining strings in the namespace are appended to the resulting string and separated by dots.
5. All letters are made lowercase.

Implementing the Service Provider

Generating the implementation code

You generate the implementation class used to build your service provider with **artix wsdl2java**'s `-impl` flag.

Tip

If your service's contract includes any custom types defined in XML Schema, you must ensure that the classes for the types are generated and available.

For more information on using **artix wsdl2java** see artix wsdl2java in the *Artix® ESB Command Reference*.

Generated code

The implementation code consists of two files:

- *`portTypeName`*`.java` — The service interface(SEI) for the service.
- *`portTypeName`*`Impl.java` — The class you will use to implement the operations defined by the service.

Implement the operation's logic

To provide the business logic for your service's operations complete the stub methods in *`portTypeName`*`Impl.java`. You usually use standard Java to implement the business logic. If your service uses custom XML Schema types, you must use the generated classes for each type to manipulate them. There are also some Artix ESB specific APIs that can be used to access some advanced features.

Example

For example, an implementation class for the service defined in Example 17 on page 63 may look like Example 19 on page 71. Only the code portions highlighted in bold must be inserted by the programmer.

Example 19. Implementation of the Greeter Service

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

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

```
public class GreeterImpl implements Greeter {

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

    public void greetMeOneWay(String me) {
       System.out.println("Executing operation greetMeOneWay\n");
       System.out.println("Hello there " + me);
    }

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

    public void pingMe() throws PingMeFault {
       FaultDetail faultDetail = new FaultDetail();
       faultDetail.setMajor((short)2);
       faultDetail.setMinor((short)1);
       System.out.println("Executing operation pingMe, throwing PingMeFault exception\n");

       throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
}
```

Developing a Consumer From a WSDL Contract

One way method of creating a consumer is to start from a WSDL contract. The contract defines the operations, messages, and transport details of the service on which a consumer makes requests. The starting point code for the consumer is generated from the WSDL contract. The functionality required by the consumer is added to the generated code.

Generating the Stub Code

Overview

The **artix wsdl2java** tool generates the stub code from the WSDL contract. The stub code provides the supporting code that is required to invoke operations on the remote service.

For consumers, the **artix wsdl2java** tool generates the following types of code:

- Stub code — Supporting files for implementing a consumer.
- Starting point code — Sample code that connects to the remote service and invokes every operation on the remote service.
- Ant build file — A `build.xml` file intended for use with the Ant build utility. It has targets for building and for running the sample consumer.

Generating the consumer code

To generate consumer code use the **artix wsdl2java** tool. Enter the following command at a command-line prompt:

```
artix wsdl2java -ant -client -d outputDir hello_world.wsdl
```

Where *`outputDir`* is the location of a directory where the generated files are placed and `hello_world.wsdl` is a file containing the contract shown in Example 17 on page 63. The `-ant` option generates an ant `build.xml` file, for use with the ant build utility. The `-client` option generates starting point code for the consumer's `main()` method.

For a complete list of the arguments available for the **artix wsdl2java** tool see artix wsdl2java in the *Artix® ESB Command Reference*.

Generating code from Ant

If you are using Apache Ant as your build system, you can call the code generator using Ant's **java** task, as shown in Example 20 on page 74.

Example 20. Generating Service Starting Point Code from Ant

```
<project name="myProject" basedir=".">
  <property name="fsf.home" location ="InstallDir"/>

   <path id="fsf.classpath">
     <fileset dir="${fsf.home}/lib">
         <include name="*.jar"/>
```

```
      </fileset>
   </path>

  <target  name="ServiceGen">
    <java classname="org.apache.cxf.tools.wsdlto.WSDLToJava" fork="true">
      <arg value="-ant"/>
      <arg value="-client"/>
      <arg value="-d"/>
      <arg value="outputDir"/>
      <arg value="myService.wsdl"/>
      <classpath>
        <path refid="fsf.classpath"/>
      </classpath>
    </java>
    ...
  </target>
  ...
</project>
```

The command line options are passed to the code generator using the task's `arg` element. Arguments that require two strings, such as `-d`, must be split into two `arg` elements.

Generated code

The preceding command generates the following Java packages:

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

The stub files generated by the **artix wsdl2java** tool fall into the following categories:

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

- `Greeter` — A Java interface that represents the Greeter `wsdl:portType` element. In JAX-WS terminology, this Java interface is the service endpoint interface (SEI).
- `SOAPService` — A Java service class (extending `javax.xml.ws.Service`) that represents the SOAPService `wsdl:service` element.
- `PingMeFault` — A Java exception class (extending `java.lang.Exception`) that represents the pingMeFault `wsdl:fault` element.

- Classes representing XML types in the `org.objectweb.hello_world_soap_http.types` package. In the HelloWorld example, the only generated types are the various wrappers for the request and reply messages. Some of these data types are useful for the asynchronous invocation model.

Implementing a Consumer

Overview

To implement a consumer when starting from a WSDL contract, you must use the following stubs:

- Service class
- SEI

Using these stubs, the consumer code instantiates a service proxy to make requests on the remote service. It also implements the consumer's business logic.

Generated service class

Example 21 on page 77 shows the typical outline of a generated service class, *ServiceName*_Service[1], which extends the `javax.xml.ws.Service` base class.

Example 21. Outline of a Generated Service Class

```
@WebServiceClient(name="..." targetNamespace="..."
                  wsdlLocation="...")
public class ServiceName extends javax.xml.ws.Service
{
  ...
  public ServiceName(URL wsdlLocation, QName serviceName) { }

  public ServiceName() { }

  @WebEndpoint(name="...")
  public SEI getPortName() { }
  .
  .
  .
}
```

The *ServiceName* class in Example 21 on page 77 defines the following methods:

- *ServiceName*(URL wsdlLocation, QName serviceName) — Constructs a service object based on the data in the `wsdl:service` element with the

[1] If the `name` attribute of the `wsdl:service` element ends in `Service` the `_Service` is not used.

QName *ServiceName* service in the WSDL contract that is obtainable from *wsdlLocation*.

- *ServiceName*() — The default constructor. It constructs a service object based on the service name and the WSDL contract that were provided at the time the stub code was generated (for example, when running the **artix wsdl2java** tool). Using this constructor presupposes that the WSDL contract remains available at a specified location.
- get*PortName*() — Returns a proxy for the endpoint defined by the `wsdl:port` element with the `name` attribute equal to *PortName*. A getter method is generated for every `wsdl:port` element defined by the *ServiceName* service. A `wsdl:service` element that contains multiple endpoint definitions results in a generated service class with multiple get*PortName*() methods.

Service endpoint interface

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

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

For example, Example 22 on page 78 shows the Greeter SEI, which is generated from the `wsdl:portType` element defined in Example 17 on page 63. For simplicity, Example 22 on page 78 omits the standard JAXB and JAX-WS annotations.

Example 22. The Greeter Service Endpoint Interface

```
package org.apache.hello_world_soap_http;
  ...
public interface Greeter
{
```

```
  public String sayHi();
  public String greetMe(String requestType);
  public void greetMeOneWay(String requestType);
  public void pingMe() throws PingMeFault;
}
```

Consumer main function

Example 23 on page 79 shows the code that implements the HelloWorld consumer. The consumer connects to the SoapPort port on the SOAPService service and then proceeds to invoke each of the operations supported by the Greeter port type.

Example 23. Consumer Implementation Code

```
package demo.hw.client;

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

public final class Client {

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

  private Client()
  {
  }

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

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

```
    {
      wsdlURL = new URL(args[0]);
    }

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

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

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

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

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

The `Client.main()` method from Example 23 on page 79 proceeds as follows:

❶ Provided that the Artix ESB runtime classes are on your classpath, the runtime is implicitly initialized. There is no need to call a special function to initialize Artix ESB.

❷ The consumer expects a single string argument that gives the location of the WSDL contract for HelloWorld. The WSDL contract's location is stored in `wsdlURL`.

❸ You create a service object using the constructor that requires the WSDL contract's location and service name.

❹ Call the appropriate `get`*`PortName`*`()` method to obtain an instance of the required port. In this case, the SOAPService service supports only the SoapPort port, which implements the `Greeter` service endpoint interface.

❺ The consumer invokes each of the methods supported by the `Greeter` service endpoint interface.

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

Part III. Developing RESTful Services

RESTful services take the concepts of lose coupling and coarse grained interfaces one step farther than standard Web services. Built using the REST architectural style, they rely solely on the four HTTP verbs to access the operations provided by a service. Artix ESB provides a robust mechanism for building RESTful services using straightforward Java classes and annotations.

Introduction to RESTful Services

Representational State Transfer (REST)is a software architecture style that centers around the transmission of data over HTTP, using only the four basic HTTP verbs. It also eschews the use of any additional wrappers such as a SOAP envelope and the use of any state data.

Overview

Representational State Transfer (REST) is an architectural style first described in a doctoral dissertation by a researcher named Roy Fielding. In REST, servers expose resources using a URI, and clients access these resources using the four HTTP verbs. As clients receive representations of a resource they are placed in a state. When they access a new resource, typically by following a link, they change, or transition, their state. In order to work, REST assumes that resources are capable of being represented using a pervasive standard grammar.

The World Wide Web is the most ubiquitous example of a system designed on REST principles. Web browsers act as clients accessing resources hosted on Web servers. The resources are represented using HTML or XML grammars that all Web browsers can consume. The browsers can also easily follow the links to new resources.

The advantages of REST style systems is that they are highly scalable and highly flexible. Because the resources are accessed and manipulated using the four HTTP verbs, the resources are exposed using a URI, and the resources are represented using standard grammars, clients are not as affected by changes to the servers. Also, REST style systems can take full advantage of the scalability features of HTTP such as caching and proxies.

Basic REST principles

RESTful architectures adhere to the following basic principles:

- Application state and functionality are divided into resources.
- Resources are addressable using standard URIs that can be used as hypermedia links.
- All resources use only the four HTTP verbs.
  - `DELETE`
  - `GET`
  - `POST`

  - `PUT`

- All resources provide information using the MIME types supported by HTTP.
- The protocol is stateless.
- The protocol is cacheable.
- The protocol is layered.

Resources

Resources are central to REST. A resource is a source of information that can be addressed using a URI. In the early days of the Web, resources were largely static documents. In the modern Web, a resource can be any source of information. For example a Web service can be a resource if it can be accessed using a URI.

RESTful endpoints exchange *representations* of the resources they address. A representation is a document containing the data provided by the resource. For example, the method of a Web service that provides access to a customer record wourld be a resource, the copy of the customer record exchanged between the service and the consumer is a representation of the resource.

REST best practices

When designing RESTful services it is helpful to keep in mind the following:

- Provide a distinct URI for each resource you wish to expose.

  For example, if you are building a system that deals with driving records, each record should have a unique URI. If the system also provides information on parking violations and speeding fines, each type of resource should also have a unique base. For example, speeding fines could be accessed through `/speeding/`*`driverID`* and parking violations could be accessed through `/parking/`*`driverID`*.

- Use nouns in your URIs.

  Using nouns highlights the fact that resources are things and not actions. URIs such as `/ordering` imply an actions, whereas `/orders` implies a thing.

- Methods that map to `GET` should not change any data.
- Use links in your responses.

Putting links to other resources in your responses makes it easier for clients to follow a chain of data. For example, if your service returns a collection of resources, it would be easier for a client to access each of the individual resources using the provided links. If links are not included, a client needs to have additional logic to follow the chain to a specific node.

- Make your service stateless.

  Requiring the client or the service to maintain state information forces a tight coupling between the two. Tight couplings make upgrading and migrating more difficult. Maintaining state can also make recovery from communication errors more difficult.

Wrapped mode vs. unwrapped mode

RESTful services can only send or receive one XML element. To enable the mapping of methods that use more than one parameter, Artix ESB can use *wrapped mode*. In wrapped mode, Artix ESB wraps the parameters with a root element derived from the operation name. For example, the operation `Car findCar(String make, String model)` could not be mapped to an XML `POST` request like the one shown in Example 24 on page 89.

Example 24. Invalid REST Request

```
<name>Dodge</name>
<model>Daytona</company>
```

Example 24 on page 89 is invalid because it has two root XML elements, which is not allowed. Instead, the parameters would have to be wrapped with the operation name to make the `POST` valid. The resulting request is shown in Example 25 on page 89.

Example 25. Wrapped REST Request

```
<findCar>
  <make>Dodge</make>
  <model>Daytona</model>
</findCar>
```

By default, Artix ESB uses unwrapped mode, because, for cases where operations use a single parameter, it creates prettier XML. Using unwrapped mode, however, requires that you constrain your service interfaces to sending and receiving single elements. If your operation needs to take multiple parameters, you must combine them in an object. With the `findCar()`

example above, you would want to create a `FindCar` class that holds the make and model data.

Implementing REST with Artix ESB

Artix ESB uses an HTTP binding to map Java interfaces into RESTful services. There are two ways to map the methods of the Java interface into resources:

- Convention based mapping (see *Using Automatic REST Mappings* on page 91)
- Java REST annotations (see *Using Java REST Annotations* on page 95)

Using Automatic REST Mappings

To simplify the creation of RESTful services, Artix ESB can automatically map a Java service to a RESTful interface. The mapping requires that the Java service is defined as a CRUD based class.

Overview

To simplify the creation of RESTful service endpoints, Artix ESB can map the methods of a CRUD (Create, Read, Update, and Destroy) based Java bean class to URIs automatically. The mapping looks for keywords in the method names of the bean, such as get, add, update, or remove, and maps them onto HTTP verbs. It then uses the remainder of the method name to create a URI by pluralizing the field name and appending it to the base URI at which the endpoint is published.

Note

For more information about publishing RESTful endpoints, see *Publishing a RESTful Service* on page 99.

Typical CRUD class

Example 26 on page 91 shows a CRUD based class for updating a catalog of widgets.

Example 26. Widget Catalog CRUD Class

```
import javax.jws.WebService;

@WebService
public interface WidgetCatalog
{
  Collection<Widget> getWidgets();
  Widget getWidget(long id);
  void addWidget(Widget widget);
  void updateWidget(Widget widget);
  void removeWidget(String type, long num);
  void deleteWidget(Widget widget);
}
```

Important

You must use the `@WebService` annotation on any class or interface that you wish to expose as a RESTful endpoint.

The class has six operations that are mapped to a URI/verb pair:

- `getWidgets()` is mapped to a `GET` at *`baseURI`*`/widgets`.
- `getWidget()` is mapped to a `GET` at *`baseURI`*`/widgets/`*`id`*.
- `addWidget()` is mapped to a `POST` at *`baseURI`*`/widgets`.
- `updateWidget()` is mapped to a `PUT` at *`baseURI`*`/widgets`.
- `removeWidget()` is mapped to a `DELETE` at *`baseURI`*`/widgets/`*`type`*`/`*`num`*.
- `deleteWidget()` is mapped to a `DELETE` at *`baseURI`*`/widgets`.

Mapping to GET

When Artix ESB sees a method name in the form of `get`*`Resource`*`()`, it maps the method to a `GET`. The URI is generated by appending the plural form of *`Resource`* to the base URI at which the endpoint is published. If *`Resource`* is already plural, it is not pluralized. For example, `getCustomer()` is mapped to a `GET` on `/customers`. The method `getCustomers()` would result in the same mapping.

Any method parameters are appended to the URI. For example, `getWidget(long id)` is mapped to `/widgets/`*`id`* and `getCar(String make, String model)` would be mapped to `/cars/`*`make`*`/`*`model`*. A call to `getCar(plymouth, roadrunner)` would be executed by a `GET` to `/cars/plymouth/roadrunner`.

Important

Artix ESB only supports get methods that use XML primitives in their parameter list.

Mapping to POST

Methods of the form `add`*`Resource`*`()` or `create`*`Resource`*`()` are mapped to `POST`. The URI is generated by pluralizing *`Resource`*. For example `createCar(Car car)` would be mapped to a `POST` at `/cars`.

Mapping to PUT

Methods of the form `update`*`Resource()`* are mapped to `PUT`. The URI is generated by pluralizing *`Resource`* and appending any parameters except the resource to be updated. For example `updateHitter(long number, long rotation, Hitter hitter)` would be mapped to a `PUT` at `/hitters/`*`number`*`/`*`rotation`*.

Important

Artix ESB only supports get methods that use XML primitives in their parameter list.

Mapping to DELETE

Methods of the form `delete`*`Resource`*`()` or `remove`*`Resource`*`()` are mapped to `DELETE`. The URI is generated by pluralizing *`Resource`* and appending any parameters. For example `removeCar(String make, long num)` would be mapped to a `DELETE` at `/cars/`*`make`*`/`*`num`*.

Important

Artix ESB only supports get methods that use XML primitives in their parameter list.

Using Java REST Annotations

Artix ESB recognizes a set of annotations that allow you to dictate the mappings of Java operations to a RESTful interface.

Overview

While the convention-based REST mappings provide an easy way to create a service that maintains a collection of data, or looks like it does, it does not provide the flexibility to create a full range of RESTful services that require operations whose names don't fit into the CRUD format. Artix ESB provides a collection of annotations that allows you to define the mapping of an operation to an HTTP verb/URI combination. The REST annotations allow you to specify which verb to use for an operation and to specify a template for creating a URI for the exposed resource.

Specifying the HTTP verb

Artix ESB uses four annotations for specifying the HTTP verb that will be used for a method:

- `org.codehaus.jra.Delete` specifies that the method maps to a `DELETE`.
- `org.codehaus.jra.Get` specifies that the method maps to a `GET`.
- `org.codehaus.jra.Post` specifies that the method maps to a `POST`.
- `org.codehaus.jra.Put` specifies that the method maps to a `PUT`.

When you map your methods to HTTP verbs, you must ensure that the mapping makes sense. For example, if you map a method that is intended to submit a purchase order, you would map it to a `PUT` or a `POST`. Mapping it to a `GET` or a `DELETE` would result in unpredictable behavior.

Specifying the URI

You specify the URI of the resource using the `org.codehaus.jra.HttpResource` annotation. `HttpResource` has one required attribute, *`location`*, that specifies the location of the resource in relationship to the base URI specified when publishing the service (see *Publishing a RESTful Service* on page 99. For example, if you specify `carts` as the location of the resource and the base URI is

`http://myexample.iona.org`, the full URI for the resource will be `http://myexample.iona.org/carts`.

Using URI templates

In addition to specifying hard coded resource locations, Artix ESB provides a facility for creating URIs on the fly using either the method's parameters or a field from the JAXB bean in the parameter list. When providing a value for the `HttpResource` annotation's *`location`* parameter you provide a URI template using the syntax in Example 27 on page 96.

Example 27. URI Template Syntax

```
@HttpResource(location="resourceName/{param1}/../{paramN}")
```

`resourceName` can be any valid string, and forms the base of the location. Each *`param`* is the name of either a method parameter or a field in the JAXB bean in the parameter list. To create the URI, Artix ESB replaces *`param`* with the value of the associated parameter. For example, if you have the method shown in Example 28 on page 96 and wanted to access the record at id 42, you would perform a `GET` at `http://myexample.iona.com/records/42`.

Example 28. Using a URI Template

```
@Get
@HttpResource(location="\records\{id}")
Record fetchRecord(long id);
```

Important

Artix ESB only supports XML primitives in URI templates.

Example

If you wanted to implement a system for ordering widgets out of the catalog defined by Example 26 on page 91 you may use an SEI like the one shown in Example 29 on page 96.

Example 29. SEI for a Widget Ordering Service

```
@WebService
public interface WidgetOrdering
{
  void placeOrder(WidgetOrder order);
  OrderStatus checkOrder(long orderNum);
  void changeOrder(WidgetOrder order, long orderNum);
```

```
  void cancelOrder(long orderNum);
}
```

`WidgetOrdering` does not match any of the naming conventions outlined in *Using Automatic REST Mappings* on page 91 so the RESTful binding cannot automatically map the methods to verb/URI combinations. You will need to provide the mappings using the Java REST annotations. To do this, you need to consider what each method in the interface does and how it correlates to one of the HTTP verbs:

- `placeOrder()` creates a new order on the system. Resource creation correlates with `POST`.
- `checkOrder()` looks up an order's status and returns it to the user. Returning resources correlates with `GET`.
- `changeOrder()` updates an order that has already been placed. Updating an existing record correlates with `PUT`.
- `cancelOrder()` removes an order from the system. Removing a resource correlates with `DELETE`.

For the URI, you would use a resource name that hinted at the purpose of the resource. For this example, the resource name used is `orders` because it is assumed that the base URI at which the endpoint is published provides information about what is being ordered. For the methods that use *`orderNum`* to identify a particular order, URI templating is used to append the value of the parameter to the end of the URI.

Example 30 on page 97 shows `WidgetOrdering` with the required annotations.

Example 30. `WidgetOrdering` with REST Annotations

```
import org.codehause.jra.*;

@WebService
public interface WidgetOrdering
{
  @Post
  @HttpResource(location="\orders")
  void placeOrder(WidgetOrder order);
```

```
  @Get
  @HttpResource(location="\orders\{orderNum}")
  OrderStatus checkOrder(long orderNum);

  @Put
  @HttpResource(location="\orders\{orderNum}")
  void changeOrder(WidgetOrder order, long orderNum);

  @Delete
  @HttpResource(location="\orders\{orderNum}")
  void cancelOrder(long orderNum);
}
```

To check the status of order number `236`, you would perform a `GET` at *`baseURI`*`/orders/236`.

Publishing a RESTful Service

The Artix ESB APIs provide a simple means of publishing a RESTful service using the `JaxWsServiceFactoryBean`.

Overview

You publish RESTful services using the `JaxWsServerFactoryBean` object. Using the `JaxWsServerFactoryBean` object, you specify the base URI for the resources implemented by the service and whether the resources use wrapped messages. You can then create a `Server` object to start listening for requests to access the service's resources.

Procedure

To publish your RESTful service, do the following:

1. Create a new `JaxWsServerFactoryBean`.
2. Set the server factory's service class to the class of your RESTful service's SEI using the factory's `setServiceClass()` method as shown in Example 31 on page 99.

   Example 31. Setting a Server Factory's Service Class

   ```
   // Service factory sf obtained previously
   sf.setServiceClass(widgetService.class);
   ```

3. If you want to use wrapped mode, set the factory's wrapped property to `true` using the `setWrapped()` method as shown in Example 32 on page 99.

   Example 32. Setting Wrapped Mode

   ```
   sf.getServiceFactory().setWrapped(true);
   ```

   Note

   For more information about using wrapped mode or unwrapped mode, see Wrapped mode vs. unwrapped mode on page 89.

4. Set the server factory's binding to the REST binding using the `setBindingId()` method.

The REST binding is selected using the constant `HttpBindingFactory.HTTP_BINDING_ID` as shown in Example 33 on page 100.

Example 33. Selecting the REST Binding

```
// Server factory sf obtained previously
sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
```

5. Set the base URI for the service's resources using the `setAddress()` method as shown in Example 34 on page 100.

   Example 34. Setting the Base URI

   ```
   sf.setAddress("http://localhost:9000");
   ```

6. Set server factory's service invoker to an instance of your service's implementation class as shown in Example 35 on page 100.

   Example 35. Setting the Service Invoker

   ```
   widgetService service = new widgetServiceImpl();
   sf.getServiceFactory().setInvoker(new BeanInvoker(ser
   vice));
   ```

7. Create a new `Server` object from the server factory using the factory's `create()` method.

Example

Example 36 on page 100 shows the code for publishing a RESTful service at `http://jfu:9000`. All of the resources implemented by the service will use the published URI as the base address.

Example 36. Publishing the WidgetCatalog Service as a RESTful Endpoint

```
JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
sf.setServiceClass(WidgetCatalog.class);

sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
sf.setAddress("http://jfu:9000");

widgetService service = new WidgetCatalogImpl();
sf.setServiceFactory.setInvoker(new BeanInvoker(service));
```

```
Server svr = sf.create();
```

If you used Example 36 on page 100 to publish the service defined by Example 26 on page 91, you would:

- Retrieve a list of all widgets in the catalog using a `GET` at `http://jfu:9000/widgets`.
- Retrieve information about widget 34 using a `GET` at `http://jfu:9000/widgets/34`.
- Modify a widget using a `PUT` at `http://jfu:9000/widgets` with an XML document describing the widget to modify.
- Delete 15 round widgets from the catalog using a `DELETE` at `http://jfu:9000/widgets/round/15`.

Part IV. Common Development Tasks

Aside from basic service provider and consumer implementation, there are a number of tasks that developers will commonly need to perform.

Finding WSDL at Runtime

Hard coding the location of WSDL documents into an application is not scalable. In real deployment environments, you will want to allow the WSDL document's location be resolved at runtime. Artix ESB provides a number of tools to make this possible.

When developing consumers using the JAX-WS APIs you are must provide a hard coded path to the WSDL document that defines your service. While this is OK in a small environment, using hard coded paths does not translate to enterprise deployments.

To address this issue, Artix ESB provides three mechanisms for removing the requirement of using hard coded paths:

- inject a configured proxy object
- a JAX-WS catalog
- the `ServiceContractResolver` interface

Tip

Injecting the proxy into your implementation code is generally the best option.

Instantiating a Proxy by Injection

Overview

Artix ESB's use of the Spring Framework allows you to avoid the hassle of using the JAX-WS APIs to create service proxies. It allows you to define a client endpoint in a configuration file and then inject a proxy directly into the implementation code. When the runtime instantiates the implementation object, it will also instantiate a proxy for the external service based on the configuration. The implementation is handed a reference to the instantiated proxy.

Because the proxy is instantiated using information in the configuration file, the WSDL location does not need to be hard coded. It can be changed at deployment time. You can also specify that the runtime should search the application's classpath for the WSDL.

Procedure

To inject a proxy for an external service into a service provider's implementation do the following:

1. Deploy the required WSDL documents in a well known location that all parts of the application can access.

   Tip

   If you are deploying the application as a WAR file, it is recommended that you place all of the WSDL documents and XML Schema documents in the `WEB-INF/wsdl` folder of the WAR.

   Tip

   If you are deploying the application as a JAR file, it is recommended that you place all of the WSDL documents and XML Schema documents in the `META-INF/wsdl` folder of the JAR.

2. Configure a JAX-WS client endpoint for the proxy that is being injected.

3. Inject the proxy into your service provide using the `@Resource` annotation.

Configuring the proxy

You configure a JAX-WS client endpoint using the `jaxws:client` element in you application's configuration file. This tells the runtime to instantiate a `org.apache.cxf.jaxws.JaxWsClientProxy` object with the specified properties. This object is the proxy that will be injected into the service provider.

At a minimum you need to provide values for the following attributes:

- `id`—Specifies the ID used to identify the client to be injected.
- `serviceClass`—Specifies the SEI of the service on which the proxy makes requests.

Example 37 on page 109 shows the configuration for a JAX-WS client endpoint.

Example 37. Configuration for a Proxy to be Injected into a Service Implementation

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:client id="bookClient"
                serviceClass="org.apache.cxf.demo.BookService"
                wsdlLocation="classpath:books.wsdl"/>
  ...
</beans>
```

Note

In Example 37 on page 109 the `wsdlLocation` attribute instructs the runtime to load the WSDL from the classpath. If `books.wsdl` in on the classpath, the runtime will be able to find it.

For more information on configuring a JAX-WS client see Configuring Consumer Endpoints in the *Artix® ESB Deployment Guide*.

Coding the provider implementation

You inject the configured proxy into a service implementation as a resource using the `@Resource` as shown in Example 38 on page 110.

Example 38. Injecting a Proxy into a Service Implementation

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName =
"SOAPService",
                      targetNamespace = "ht
tp://apache.org/hello_world_soap_http",
                      endpointInterface =
"org.apache.hello_world_soap_http.Greeter")
public class StoreImpl implements Store {

@Resource(name="bookClient")
private BookService proxy;

}
```

The annotation's name property corresponds to the value of the JAX-WS client's `id` attribute. The configured proxy is injected into the `BookService` object declared immediately after the annotation. You can use this object to make invocations on the proxy's external service.

Using a JAX-WS Catalog

Overview

The JAX-WS specification mandates the all implementations support:

> support for a standard catalog facility to be used when resolving any Web service document that is part of the description of a Web service, specifically WSDL and XML Schema documents.

This catalog facility uses the XML catalog facility specified by OASIS. All of the JAX-WS APIs and annotation that take a WSDL URI use the catalog to resolve the WSDL document's location.

This means that you can provide an XML catalog file that rewrites the locations of your WSDL documents to suite specific deployment environments.

Writing the catalog

JAX-WS catalogs are standard XML catalogs as defined by the OASIS XML Catalogs 1.1[1] specification. They allow you to specify mapping:

- a document's public identifier and/or a system identifier to a URI.
- the URI of a resource to another URI.

Table 10 on page 111 lists some common elements used for WSDL location resolution.

Table 10. Common JAX-WS Catalog Elements

Element	Description
`uri`	Maps a URI to an alternate URI.
`rewriteURI`	Rewrites the beginning of a URI. For example, this element allows you to map all URIs that start with `http://cxf.apache.org` to URIs that start with `classpath:`.
`uriSuffix`	Maps a URI to an alternate URI based on the suffix of the original URI. For example you could map all URIs that end in `foo.xsd` to `classpath:foo.xsd`.

Packaging the catalog

The JAX-WS specification mandates that the catalog used to resolve WSDL and XML Schema documents is assembled using all available resources named

[1] http://www.oasis-open.org/committees/download.php/14041/xml-catalogs.html

`META-INF/jax-ws-catalog.xml`. If your application is packaged into a single JAR, or WAR, you can place the catalog into a single file.

If your application is packaged as multiple JARs, you can split the catalog into a number of files. Each catalog file could be modularized to only deal with WSDLs accessed by the code in the specific JARs.

Using a `ServiceContractResolver` Object

Overview

The most involved mechanism for resolving WSDL document locations at runtime is to implement your own custom contract resolver. This requires that you provide an implementation of the Artix ESB specific `ServiceContractResolver` interface. You also need to register your custom resolver with the bus.

Once properly registered, the custom contract resolver will be used to resolve the location of any required WSDL and schema documents.

Implementing the contract resolver

A contract resolver is an implementation of the `org.apache.cxf.endpoint.ServiceContractResolver` interface. As shown in Example 39 on page 113, this interface has a single method, `getContractLocation()`, that needs to be implemented.

`getContractLocation()` takes the QName of a service and returns the URI for the service's WSDL contract.

Example 39. `ServiceContractResolver` Interface

```
public interface ServiceContractResolver
{
   URI getContractLocation(QName qname);
}
```

The logic used to resolve the WSDL contract's location is application specific. You can add logic that to resolve contract locations from a UDDI registry, a database, a custom location on a file system, or any other mechanism you choose.

Registering the contract resolver programmatically

Before the Artix ESB runtime will use your contract resolver, you must register it with a contract resolver registry. Contract resolver registries implement the `org.apache.cxf.endpoint.ServiceContractResolverRegistry` interface. However, you do not need to implement your own registry. Artix ESB provides a default implementation in the `org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl` class.

To register a contract resolver with the default registry you do the following:

1. Get a reference to the default bus object.
2. Get the service contract registry from the bus using the bus' `getExtension()` method.
3. Create an instance of your contract resolver.
4. Register your contract resolver with the registry using the registry's `register()` method.

Example 40 on page 114 shows the code for registering a contract resolver with the default registry.

Example 40. Registering a Contract Resolver

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

ServiceContractResolverRegistry registry = bus.getExtension(ServiceContractResolverRegistry);
 ❷

JarServiceContractResolver resolver = new JarServiceContractResolver(); ❸

registry.register(resolver); ❹
```

The code in Example 40 on page 114 does the following:

❶ Gets a bus instance.

❷ Gets the bus' contract resolver registry.

❸ Creates an instance of a contract resolver.

❹ Registers the contract resolver with the registry.

Registering a contract resolver using configuration

You can also implement a contract resolver so that it can be added to a client through configuration. The contract resolver is implemented in such a way that when the runtime reads the configuration and instantiates the resolver, the resolver registers itself. Because the runtime handles the initialization, you can decide at runtime if a client needs to use the contract resolver.

To implement a contract resolver so that it can be added to a client through configuration do the following:

1. Add an `init()` method to your contract resolver implementation.

2. Add logic to your `init()` method that registers the contract resolver with the contract resolver registry as shown in Example 40 on page 114.
3. Decorate the `init()` method with the `@PostConstruct` annotation.

Example 41 on page 115 shows a contract resolver implementation that can be added to a client using configuration.

Example 41. Service Contract Resolver that can be Registered Using Configuration

```
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.xml.namespace.QName;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;

public class UddiResolver implements ServiceContractResolver
{
  private Bus bus;
  ...

  @PostConstruct
  public void init()
  {
    BusFactory bf=BusFactory.newInstance();
    Bus bus=bf.createBus();
    if (null != bus)
    {
      ServiceContractResolverRegistry resolverRegistry = bus.getExtension(ServiceContract
ResolverRegistry.class);
      if (resolverRegistry != null)
      {
        resolverRegistry.register(this);
      }
    }
  }

  public URI getContractLocation(QName serviceName)
  {
    ...
  }
}
```

To register the contract resolver with a client you need to add a `bean` element to the client's configuration. The `bean` element's `class` attribute is the name of the class implementing the contract resolver.

Example 42 on page 116 shows a bean for adding a configuration resolver implemented by the `org.apache.cxf.demos.myContractResolver` class.

Example 42. Bean Configuring a Contract Resolver

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.spring
framework.org/schema/beans/spring-beans.xsd">
  ...
  <bean id="myResolver" class="org.apache.cxf.demos.myContract
Resolver" />
  ...
</beans>
```

Contract resolution order

When a new proxy is created, the runtime uses the contract registry resolver to locate the remote service's WSDL contract. The contract resolver registry calls each contract resolver's `getContractLocation()` method in the order in which the resolvers were registered. It returns the first URI returned from one of the registered contract resolvers.

If you registered a contract resolver that attempted to resolve the WSDL contract at a well known shared file system, it would be the only contract resolver used. However, if you subsequently registered a contract resolver that resolved WSDL locations using a UDDI registry, the registry could use both resolvers to locate a service's WSDL contract. The registry would first attempt to locate the contract using the shared file system contract resolver. If that contract resolver failed, the registry would then attempt to locate it using the UDDI contract resolver.

Publishing a Service

When you want to deploy a JAX-WS service as a standalone Java application or in an OSGi container without Spring-DM, you must to implement the code that publishes the service provider.

Artix ESB provides a number of ways to publish a service as a service provider. How you publish a service depends on the deployment environment you are using. Many of the containers supported by Artix ESB do not require writing logic for publishing endpoints. There are two exceptions:

- deploying a server as a standalone Java application
- deploying a server into an OSGi container without Spring-DM

For detailed information in deploying applications into the supported containers see Artix® ESB Deployment Guide.

APIs Used to Publish a Service

Overview

The `javax.xml.ws.Enddpoint` class does the work of publishing a JAX-WS service provider. To publishing an endpoint do the following:

1. Create an `Endpoint` object for your service provider.
2. Publish the endpoint.
3. Stop the endpoint when application shuts down.

The `Endpoint` class provides methods for creating and publishing service providers. It also provides a method that can create and publish a service provider in a single method call.

Instantiating an service provider

A service provider is instantiated using an `Endpoint` object. You instantiate an `Endpoint` object for your service provider using one of the following methods:

- ```
  static Endpoint create(Object implementor);
  ```

  This `create()` method returns an `Endpoint` for the specified service implementation. The `Endpoint` object is created using the information provided by the implementation class' `javax.xml.ws.BindingType` annotation, if it is present. If the annotation is not present, the `Endpoint` uses a default SOAP 1.1/HTTP binding.

- ```
  static Endpoint create(URI bindingID,
                         Object implementor);
  ```

  This `create()` method returns an `Endpoint` object for the specified implementation object using the specified binding. This method overrides the binding information provided by the `javax.xml.ws.BindingType` annotation, if it is present. If the *`bindingID`* cannot be resolved, or it is `null`, the binding specified in the `javax.xml.ws.BindingType` is used to create the `Endpoint`. If neither the *`bindingID`* or the `javax.xml.ws.BindingType` can be used, the `Endpoint` is created using a default SOAP 1.1/HTTP binding.

- `static Endpoint publish(String address, Object implementor);`

  The `publish()` method creates an `Endpoint` object for the specified implementation, and publishes it. The binding used for the `Endpoint` object is determined by the URL scheme of the provided *`address`*. The list of bindings available to the implementation are scanned for a binding that supports the URL scheme. If one is found the `Endpoint` object is created and published. If one is not found, the method fails.

Tip

Using `publish()` is the same as invoking one of the `create()` methods, and then invoking the `publish()` method used in publish to an address.

Important

The implementation object passed to any of the `Endpoint` creation methods must either be an instance of a class annotated with `javax.jws.WebService` and meeting the requirements for being an SEI implementation or it must be an instance of a class annotated with `javax.xml.ws.WebServiceProvider` and implementing the `Provider` interface.

Publishing a service provider

You can publish a service provider using either of the following `Endpoint` methods:

- `void publish(String address);`

  This `publish()` method publishes the service provider at the address specified.

Important

The *`address`*'s URL scheme must be compatible with one of the service provider's bindings.

- `void publish(Object serverContext);`

  This `publish()` method publishes the service provider based on the information provided in the specified server context. The server context must define an address for the endpoint, and the context must also be compatible with one of the service provider's available bindings.

Stopping a published service provider

When the service provider is no longer needed you should stop it using its `stop()` method. The `stop()` method, shown in Example 43 on page 120, shuts down the endpoint and cleans up any resources it is using.

Example 43. Method for Stopping a Published Endpoint

```
void stop();
```

Important

Once the endpoint is stopped it cannot be republished.

Publishing a Service in a Plain Java Application

Overview

When you want to deploy your application as a plain java application you need to implement the logic for publishing your endpoints in the application's `main()` method. Artix ESB provides you two options for writing your application's `main()` method.

- use the `main()` method generated by the **artix wsdl2java** tool
- write a custom `main()` method that publishes the endpoints

Generating a Server Mainline

The **artix wsdl2java** tool's `-server` flag makes the tool generate a simple server mainline. The generated server mainline, as shown in Example 44 on page 121, publishes one service provider for each `port` element in the specified WSDL contract.

For more information see artix wsdl2java in the *Artix® ESB Command Reference*.

Example 44 on page 121 shows a generated server mainline.

Example 44. Generated Server Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer {

    protected GreeterServer() throws Exception {
        System.out.println("Starting Server");
❶         Object implementor = new GreeterImpl();
❷         String address = "http://localhost:9000/SoapContext/SoapPort";
❸         Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new GreeterServer();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
```

```
        System.exit(0);
    }
}
```

The code in Example 44 on page 121 does the following:

❶ Instantiates a copy of the service implementation object.

❷ Creates the address for the endpoint based on the contents of the `address` child of the `wsdl:port` element in the endpoint's contract.

❸ Publishes the endpoint.

Writing a Server Mainline

If you used the Java first development model or you do not want to use the generated server mainline you can write your own. To write your server mainline you must do the following:

1. Instantiate an `javax.xml.ws.Endpoint` object for the service provider.
2. Create an optional server context to use when publishing the service provider.
3. Publish the service provider using one of the `publish()` methods.
4. Stop the service provider when the application is ready to exit.

Example 45 on page 122 shows the code for publishing a service provider.

Example 45. Custom Server Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer
{
  protected GreeterServer() throws Exception
  {
  }

  public static void main(String args[]) throws Exception
  {
❶    GreeterImpl impl = new GreeterImpl();
❷    Endpoint endpt.create(impl);
❸    endpt.publish("http://localhost:9000/SoapContext/SoapPort");

    boolean done = false;
```

```
❹    while(!done)
     {
       ...
     }

❺    endpt.stop();
     System.exit(0);
   }
}
```

The code in Example 45 on page 122 does the following:

❶ Instantiates a copy of the service's implementation object.

❷ Creates an unpublished `Endpoint` for the service implementation.

❸ Publishes the service provider at `http://localhost:9000/SoapContext/SoapPort`.

❹ Loops until the server should be shutdown.

❺ Stops the published endpoint.

Publishing a Service in an OSGi Container

Overview

When you develop an application that will be deployed into an OSGi container, you need to coordinate the publishing and stopping of your endpoints with the life-cycle of the bundle in which it is packaged. You want your endpoints published when the bundle is started and you want the endpoints stopped when the bundle is stopped.

You tie your endpoints life-cycle to the bundle's life-cycle by implementing an OSGi bundle activator. A bundle activator is used by the OSGi container to create the resource for a bundle when it is started. The container also uses the bundle activator to clean up the bundles resources when it is stopped.

The bundle activator interface

You create a bundle activator for your application by implementing the `org.osgi.framework.BundleActivator` interface. The `BundleActivator` interface, shown in Example 46 on page 124, it has two methods that need to be implemented.

Example 46. Bundle Activator Interface

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

The `start()` method is called by the container when it starts the bundle. This is where you instantiate and publish the endpoints.

The `stop()` method is called by the container when it stops the bundle. This is where you would stop the endpoints.

Implementing the start method

The bundle activator's start method is where you publish your endpoints. To publish your endpoints the start method must do the following:

1. Instantiate an `javax.xml.ws.Endpoint` object for the service provider.
2. Create an optional server context to use when publishing the service provider.

3. Publish the service provider using one of the `publish()` methods.

Example 47 on page 125 shows code for publishing a service provider.

Example 47. Bundle Activator Start Method for Publishing an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void start(BundleContext context)
  {
❶    WidgetOrderImpl impl = new WidgetOrderImpl();
❷    endpt = Endpoint.create(impl);
❸    endpt.publish("http://localhost:9000/SoapContext/SoapPort");
  }

  ...

}
```

The code in Example 47 on page 125 does the following:

❶ Instantiates a copy of the service's implementation object.

❷ Creates an unpublished `Endpoint` for the service implementation.

❸ Publish the service provider at `http://localhost:9000/SoapContext/SoapPort`.

Implementing the stop method

The bundle activator's stop method is where you clean up the resources used by your application. Its implementation should include logic for stopping all of the endpoint's published by the application.

Example 48 on page 125 shows a stop method for stopping a published endpoint.

Example 48. Bundle Activator Stop Method for Stopping an Endpoint

```
package com.widgetvendor.osgi;
```

```
import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void stop(BundleContext context)
  {
    endpt.stop();
  }

  ...

}
```

Informing the container

You must add inform the container that the application's bundle includes a bundle activator. You do this by adding the Bundle-Activator property to the bundle's manifest. This property tells the container which class in the bundle to use when activating the bundle. Its value is the fully qualified name of the class implementing the bundle activator.

Example 49 on page 126 shows a manifest entry for a bundle whose activator is implemented by the class `com.widgetvendor.osgi.widgetActivator`.

Example 49. Bundle Activator Manifest Entry

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

Generic Fault Handling

The JAX-WS specification defines two type of faults. One is a generic JAX-WS runtime exception. The other is a protocol specific class of exceptions that is thrown during message processing.

Runtime Faults

Overview

Most of the JAX-WS APIs throw a generic `javax.xml.ws.WebServiceException` exception.

APIs that throw WebServiceException

Table 11 on page 128 lists some of the JAX-WS APIs that can throw the generic `WebServiceException` exception.

Table 11. APIs that Throw `WebServiceException`

API	Reason
`Binding.setHandlerChain()`	There is an error in the handler chain configuration.
`BindingProvider.getEndpointReference()`	The specified class is not assigned from a `W3CEndpointReference`.
`Dispatch.invoke()`	There is an error in the `Dispatch` instance's configuration or an error occurred while communicating with the service.
`Dispatch.invokeAsync()`	There is an error in the `Dispatch` instance's configuration.
`Dispatch.invokeOneWay()`	There is an error in the `Dispatch` instance's configuration or an error occurred while communicating with the service.
`LogicalMessage.getPayload()`	An error occurred when using a supplied `JAXBContext` to unmarshall the payload. The `cause` field of the `WebServiceException` contains the original `JAXBException`.
`LogicalMessage.setPayload()`	An error occurred when setting the payload of the message. If the exception is thrown when using a `JAXBContext`, the `cause` field of the `WebServiceException` contains the original `JAXBException`.
`WebServiceContext.getEndpointReference()`	The specified class is not assigned from a `W3CEndpointReference`.

Protocol Faults

Overview

Protocol exceptions are thrown when an error occurs during the processing of a request. All synchronous remote invocations can throw a protocol exception. The underlying cause occurs either in the consumer's message handling chain or in the service provider.

The JAX-WS specification defines a generic protocol exception. It also specifies a SOAP-specific protocol exception and an HTTP-specific protocol exception.

Types of protocol exceptions

The JAX-WS specification defines three types of protocol exception. Which exception you catch depends on the transport and binding used by your application.

Table 12 on page 129 describes the three types of protocol exception and when they are thrown.

Table 12. Types of Generic Protocol Exceptions

Exception Class	When Thrown
`javax.xml.ws.ProtocolException`	This exception is the generic protocol exception. It can be caught regardless of the protocol in use. It can be cast into a specific fault type if you are using the SOAP binding or the HTTP binding. When using the XML binding in combination with the HTTP or JMS transports, the generic protocol exception cannot be cast into a more specific fault type.
`javax.xml.ws.soap.SOAPFaultException`	This exception is thrown by remote invocations when using the SOAP binding. For more information see Using the SOAP protocol exception on page 129.
javax.xml.ws.http.HTTPException	This exception is thrown when using the Artix ESB HTTP binding to develop RESTful services. For more information see ????.

Using the SOAP protocol exception

The `SOAPFaultException` exception wraps a SOAP fault. The underlying SOAP fault is stored in the `fault` field as a `javax.xml.soap.SOAPFault` object.

If a service implementation needs to throw an exception that does not fit any of the custom exceptions created for the application, it can wrap the fault in a `SOAPFaultException` using the exceptions creator and throw it back to

the consumer. Example 50 on page 130 shows code for creating and throwing a `SOAPFaultException` if the method is passed an invalid parameter.

Example 50. Throwing a SOAP Protocol Exception

```
public Quote getQuote(String ticker)
{
  ...
  if(tickers.length()<3)
  {
    SOAPFault fault = SOAPFactory.newInstance().createFault();

    fault.setFaultString("Ticker too short");
    throw new SOAPFaultException(fault);
  }
  ...
}
```

When a consumer catches a `SOAPFaultException` exception they can retrieve the underlying cause of the exception by examining the wrapped `SOAPFault` exception. As shown in Example 51 on page 130, the `SOAPFault` exception is retrieved using the `SOAPFaultException` exception's `getFault()` method.

Example 51. Getting the Fault from a SOAP Protocol Exception

```
...
try
{
  proxy.getQuote(ticker);
}
catch (SOAPFaultException sfe)
{
  SOAPFault fault = sfe.getFault();
  ...
}
```

Part V. Working with Data Types

Service-oriented design abstracts data into a common exchange format. Typically, this format is an XML grammar defined in XML Schema. To save the developer from working directly with XML documents, the JAX-WS specification calls for XML Schema types to be marshaled into Java objects. This marshaling is done in accordance with the Java Architecture for XML Binding (JAXB) specification. JAXB defines bindings for mapping between XML Schema constructs and Java objects and rules for how to marshal the data. It also defines an extensive customization framework for controlling how data is handled.

Basic Data Binding Concepts

There are a number of general topics that apply to how Artix ESB handles type mapping.

Including and Importing Schema Definitions

Overview

Artix ESB supports the including and importing of schema definitions, using the `<include/>` and `<import/>` schema tags. These tags enable you to insert definitions from external files or resources into the scope of a schema element. The essential difference between including and importing is:

- Including brings in definitions that belong to the same target namespace as the enclosing schema element.
- Importing brings in definitions that belong to a different target namespace from the enclosing schema element.

xsd:include syntax

The include directive has the following syntax:

```
<include schemaLocation="anyURI" />
```

The referenced schema, given by *anyURI*, must either belong to the same target namespace as the enclosing schema, or not belong to any target namespace at all. If the referenced schema does not belong to any target namespace, it is automatically adopted into the enclosing schema's namespace when it is included.

Example 52 on page 136 shows an example of an XML Schema document that includes another XML Schema document.

Example 52. Example of a Schema that Includes Another Schema

```
<definitions targetNamespace="http://schemas.iona.com/tests/schema_parser"
             xmlns:tns="http://schemas.iona.com/tests/schema_parser"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://schemas.iona.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <include schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="tns:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
  </types>
```

```
  ...
</definitions>
```

Example 53 on page 137 shows the contents of the included schema file.

Example 53. Example of an Included Schema

```
<schema targetNamespace="http://schemas.iona.com/tests/schema_parser"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

xsd:import syntax

The import directive has the following syntax:

```
<import namespace="namespaceAnyURI"
        schemaLocation="schemaAnyURI" />
```

The imported definitions must belong to the *namespaceAnyURI target* namespace. If *namespaceAnyURI* is blank or remains unspecified, the imported schema definitions are unqualified.

Example 54 on page 137 shows an example of an XML Schema that imports another XML Schema.

Example 54. Example of a Schema that Includes Another Schema

```
<definitions targetNamespace="http://schemas.iona.com/tests/schema_parser"
             xmlns:tns="http://schemas.iona.com/tests/schema_parser"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://schemas.iona.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://schemas.iona.com/tests/imported_types"
              schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="tns:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
```

```
  </types>
  ...
</definitions>
```

Example 55 on page 138 shows the contents of the imported schema file.

Example 55. Example of an Included Schema

```
<schema targetNamespace="http://schemas.iona.com/tests/imported_types"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

Using non-referenced schema documents

Using types defined in a schema document that is not referenced in the service's WSDL document is a three step process:

1. Convert the schema document to a WSDL document using the **xsd2wsdl** tool.
2. Generate Java for the types using the **artix wsdl2java** tool on the generated WSDL document.

   Important

   You will get a warning from the **artix wsdl2java** tool stating that the WSDL document does not define any services. You can ignore this warning.

3. Add the generated classes to your classpath.

XML Namespace Mapping

Overview

XML Schema type, group, and element definitions are scoped using namespaces. The namespaces prevent possible naming clashes between entities that use the same name. Java packages serve a similar purpose. Therefore, Artix ESB maps the target namespace of a schema document into a package containing the classes necessary to implement the structures defined in the schema document.

Package naming

The name of the generated package is derived from a schema's target namespace using the following algorithm:

1. The URI scheme, if present, is stripped.

   Note

   Artix ESB will only strip the `http:`, `https:`, and `urn:` schemes.

   For example, the namespace http:\\www.widgetvendor.com\types\widgetTypes.xsd becomes `\\widgetvendor.com\types\widgetTypes.xsd`.

2. The trailing file type identifier, if present, is stripped.

   For example, `\\www.widgetvendor.com\types\widgetTypes.xsd` becomes `\\widgetvendor.com\types\widgetTypes`.

3. The resulting string is broken into a list of strings using `/` and `:` as separators.

   So, `\\www.widgetvendor.com\types\widgetTypes` becomes the list `{"www.widegetvendor.com", "types", "widgetTypes"}`.

4. If the first string in the list is an internet domain name, it is decomposed as follows:

   a. The leading `www.` is stripped.

   b. The remaining string is split into its component parts using the `.` as the separator.

c. The order of the list is reversed.

So, `{"www.widegetvendor.com", "types", "widgetTypes"}` becomes `{"com", "widegetvendor", "types", "widgetTypes"}`

Note

Internet domain names end in one of the following: `.com`, `.net`, `.edu`, `.org`, `.gov`, or in one of the two-letter country codes.

5. The strings are converted into all lower case.

   So, `{"com", "widegetvendor", "types", "widgetTypes"}` becomes `{"com", "widegetvendor", "types", "widgettypes"}`.

6. The strings are normalized into valid Java package name components as follows:

   a. If the strings contain any special characters, the special characters are converted to an underscore(_).

   b. If any of the strings are a Java keyword, the keyword is prefixed with an underscore(_).

   c. If any of the strings begin with a numeral, the string is prefixed with an underscore(_).

7. The strings are concatenated using `.` as a separator.

   So, `{"com", "widegetvendor", "types", "widgettypes"}` becomes the package name `com.widgetvendor.types.widgettypes`.

The XML Schema constructs defined in the namespace http:\\www.widgetvendor.com\types\widgetTypes.xsd are mapped to the Java package `com.widgetvendor.types.widgettypes`.

Package contents

A JAXB generated package contains the following:

- A class implementing each complex type defined in the schema

For more information on complex type mapping see Using Complex Types on page 171.

- An enum type for any simple types defined using the `enumeration` facet

  For more information on how enumerations are mapped see Enumerations on page 160.

- A public `ObjectFactory` class that contains methods for instantiating objects from the schema

  For more information on the `ObjectFactory` class see The Object Factory on page 142.

- A `package-info.java` file that provides metadata about the classes in the package

The Object Factory

Overview

JAXB uses an object factory to provide a mechanism for instantiating instances of JAXB generated constructs. The object factory contains methods for instantiating all of the XML schema defined constructs in the package's scope. The only exception is that enumerations do not get a creation method in the object factory.

Complex type factory methods

For each Java class generated to implement an XML schema complex type, the object factory contains a method for creating an instance of the class. This method takes the form:

```
typeName createtypeName();
```

For example, if your schema contained a complex type named `widgetType`, Artix ESB generates a class called `WidgetType` to implement it.

Example 56 on page 142 shows the generated creation method in the object factory.

Example 56. Complex Type Object Factory Entry

```
public class ObjectFactory
{
  ...
  WidgetType createWidgetType()
  {
    return new WidgetType();
  }
  ...
}
```

Element factory methods

For elements that are declared in the schema's global scope, Artix ESB inserts a factory method into the object factory. As discussed in Using XML Elements on page 147, XML Schema elements are mapped to `JAXBElement<T>` objects. The creation method takes the form:

```
public JAXBElement<elementType> createelementName(elementType
value);
```

For example if you have an element named `comment` of type xsd:string, Artix ESB generates the object factory method shown in Example 57 on page 143

Example 57. Element Object Factory Entry

```
public class ObjectFactory
{
  ...
    @XmlElementDecl(namespace = "...", name = "comment")
    public JAXBElement<String> createComment(String value) {
        return new JAXBElement<String>(_Comment_QNAME, String.class, null, value);
    }
  ...
}
```

Adding Classes to the Runtime Marshaller

Overview

When the Artix ESB runtime reads and writes XML data it uses a map that associates the XML Schema types with their representative Java types. By default, the map contains all of the types defined in the target namespace of the WSDL contract's `schema` element. It also contains any types that are generated from the namespaces of any schemas that are imported into the WSDL contract.

The addition of types from namespaces other than the schema namespace used by an application's `schema` element is accomplished using the `@XmlSeeAlso` annotation. If your application needs to work with types that are generated outside the scope of your application's WSDL document, you can edit the `@XmlSeeAlso` annotation to add them to the JAXB map.

Using the @XmlSeeAlso annotation

The `@XmlSeeAlso` annotation can be added to the SEI of your service. It contains a comma separated list of classes to include in the JAXB context. Example 58 on page 144 shows the syntax for using the `@XmlSeeAlso` annotation.

Example 58. Syntax for Adding Classes to the JAXB Context

```
import javax.xml.bind.annotation.XmlSeeAlso;
@WebService()
@XmlSeeAlso({Class1.class,
             Class2.class,
             ...,
             ClassN.class})
   public class GeneratedSEI {
       ...
    }
```

Tip

In cases where you have access to the JAXB generated classes, it is more efficient to use the `ObjectFactory` classes generated to

support the needed types. Including the `ObjectFactory` class includes all of the classes that are known to the object factory.

Example

Example 59 on page 145 shows an SEI annotated with `@XmlSeeAlso`.

Example 59. Adding Classes to the JAXB Context

```
...
import javax.xml.bind.annotation.XmlSeeAlso;
...
@WebService()
@XmlSeeAlso({org.apache.schemas.types.test.ObjectFactory.class,org.apache.schem
as.tests.group_test.ObjectFactory.class})
   public interface Foo {
       ...
   }
```

Using XML Elements

XML Schema elements are used to define an instance of an element in an XML document. Elements are defined either in the global scope of an XML Schema document, or they are defined as a member of a complex type. When they are defined in the global scope, Artix ESB maps them to a JAXB element class that makes manipulating them easier.

Overview

An element instance in an XML document is defined by an XML Schema `element` element in the global scope of an XML Schema document To make it easier for Java developers to work with elements, Artix ESB maps globally scoped elements to either a special JAXB element class or to a Java class that is generated to match its content type.

How the element is mapped depends on if the element is defined using a named type referenced by the `type` attribute or if the element is defined using an in-line type definition. Elements defined with in-line type definitions are mapped to Java classes.

Tip

It is recommended that elements are defined using a named type because in-line types are not reusable by other elements in the schema.

XML Schema mapping

In XML Schema elements are defined using `element` elements. `element` elements has one required attribute. The `name` specifies the name of the element as it appears in an XML document.

In addition to the `name` attribute `element` elements have the optional attributes listed in Table 13 on page 147.

Table 13. Attributes Used to Define an Element

Attribute	Description
`type`	Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. If this attribute is not specified, you will need to include an in-line type definition.

Attribute	Description
nillable	Specifies if an element can be left out of a document entirely. If nillable is set to true, the element can be omitted from any document generated using the schema.
abstract	Specifies if an element can be used in an instance document. true indicates that the element cannot appear in the instance document. Instead, another element whose substitutionGroup attribute contains the QName of this element must appear in this element's place. For information on how this attribute effects code generation see Java mapping of abstract elements on page 151.
substitutionGroup	Specifies the name of an element that can be substituted with this element. For more information on using type substitution see Element Substitution on page 217.
default	Specifies a default value for an element. For information on how this attribute effects code generation see Java mapping of elements with a default value on page 152.
fixed	Specifies a fixed value for the element.

Example 60 on page 148 shows a simple element definition.

Example 60. Simple XML Schema Element Definition

```
<element name="joeFred" type="xsd:string" />
```

An element can also define its own type using an in-line type definition. In-line types are specified using either a complexType element or a simpleType element. Once you specify whether the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data.

Example 61 on page 148 shows an element definition with an in-line type definition.

Example 61. XML Schema Element Definition with an In-Line Type

```
<element name="skate">
  <complexType>
    <sequence>
      <element name="numWheels" type="xsd:int" />
      <element name="brand" type="xsd:string" />
    </sequence>
```

```
  </complexType>
</element>
```

Java mapping of elements with a named type

By default, globally defined elements are mapped to `JAXBElement<T>` objects where the template class is determined by the value of the `element` element's `type` attribute. For primitive types, the template class is derived using the wrapper class mapping described in Wrapper classes on page 155. For complex types, the Java class generated to support the complex type is used as the template class.

To support the mapping and to relieve the developer of unnecessary worry about an element's QName, an object factory method is generated for each globally defined element, as shown in Example 62 on page 149.

Example 62. Object Factory Method for a Globally Scoped Element

```
public class ObjectFactory {

    private final static QName _name_QNAME = new QName("targetNamespace", "localName");

  ...

   @XmlElementDecl(namespace = "targetNamespace", name = "localName")
    public JAXBElement<type> createname(type value);

}
```

For example, the element defined in Example 60 on page 148 results in the object factory method shown in Example 63 on page 149.

Example 63. Object Factory for a Simple Element

```
public class ObjectFactory {

    private final static QName _JoeFred_QNAME = new QName("...", "joeFred");

  ...

   @XmlElementDecl(namespace = "...", name = "joeFred")
    public JAXBElement<String> createJoeFred(String value);

}
```

Example 64 on page 150 shows an example of using a globally scoped element in Java.

Example 64. Using a Globally Scoped Element

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

Using elements with named types in WSDL

If a globally scoped element is used to define a message part, the generated Java parameter is not an instance of `JAXBElement<T>`. Instead it is mapped to a regular Java type or class.

Given the WSDL fragment shown in Example 65 on page 150, the resulting method has a parameter of type String.

Example 65. WSDL Using an Element as a Message Part

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"
                  xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"><element name="sayHi">
      <element name="sayHi" type="string"/>
      <element name="sayHiResponse" type="string"/>
    </schema>
  </wsdl:types>

  <wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
  </wsdl:message>
  <wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
  </wsdl:message>

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

Example 66 on page 151 shows the generated method signature for the `sayHi` operation.

Example 66. Java Method Using a Global Element as a Part

```
String sayHi(String in);
```

Java mapping of elements with an in-line type

When an element is defined using an in-line type, it is mapped to Java following the same rules used for mapping other types to Java. The rules for simple types are described in Using Simple Types on page 153. The rules for complex types are described in Using Complex Types on page 171.

When a Java class is generated for an element with an in-line type definition, the generated class is decorated with the `@XmlRootElement` annotation. The `@XmlRootElement` annotation has two useful properties: name and namespace. These attributes are described in Table 14 on page 151.

Table 14. Properties for the @XmlRootElement Annotation

Property	Description
name	Specifies the value of the XML Schema `element` element's `name` attribute.
namespace	Specifies the namespace in which the element is defined. If this element is defined in the target namespace, the property is not specified.

The `@XmlRootElement` annotation is not used if the element meets one or more of the following conditions:

- The element's `nillable` attribute is set to `true`
- The element is the head element of a substitution group

  For more information on substitution groups see *Element Substitution* on page 217.

Java mapping of abstract elements

When the element's `abstract` attribute is set to `true` the object factory method for instantiating instances of the type is not generated. If the element

is defined using an in-line type, the Java class supporting the in-line type is generated.

Java mapping of elements with a default value

When the element's `default` attribute is used the defaultValue property is added to the generated `@XmlElementDecl` annotation. For example, the element defined in Example 67 on page 152 results in the object factory method shown in Example 68 on page 152.

Example 67. XML Schema Element with a Default Value

```
<element name="size" type="xsd:int" default="7"/>
```

Example 68. Object Factory Method for an Element with a Default Value

```
@XmlElementDecl(namespace = "...", name = "size", defaultValue = "7")
    public JAXBElement<Integer> createUnionJoe(Integer value) {
        return new JAXBElement<Integer>(_Size_QNAME, Integer.class, null, value);
    }
```

Using Simple Types

XML Schema simple types are either XML Schema primitive types like xsd:int, or are defined using the `simpleType` *element. They are used to specify elements that do not contain any children or attributes. They are generally mapped to native Java constructs and do not require the generation of special classes to implement them. Enumerated simple types do not result in generated code because they are mapped to Java enum types.*

Primitive Types

Overview

When a message part is defined using one of the XML Schema primitive types, the generated parameter's type is mapped to a corresponding Java native type. The same pattern is used when mapping elements that are defined within the scope of a complex type. The resulting field is of the corresponding Java native type.

Mappings

Table 15 on page 154 lists the mapping between XML Schema primitive types and Java native types.

Table 15. XML Schema Primitive Type to Java Native Type Mapping

XML Schema Type	Java Type
xsd:string	`String`
xsd:integer	`BigInteger`
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	`BigDecimal`
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	`QName`
xsd:dateTime	`XMLGregorianCalendar`
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	`XMLGregorianCalendar`

XML Schema Type	Java Type
xsd:date	`XMLGregorianCalendar`
xsd:g	`XMLGregorianCalendar`
xsd:anySimpleType [a]	`Object`
xsd:anySimpleType [b]	`String`
xsd:duration	`Duration`
xsd:NOTATION	`QName`

[a]For elements of this type.

[b]For attributes of this type.

Wrapper classes

Mapping XML Schema primitive types to Java primitive types does not work for all possible XML Schema constructs. Several cases require that an XML Schema primitive type is mapped to the Java primitive type's corresponding wrapper type. These cases include:

- An `element` element with its `nillable` attribute set to `true` as shown:

```
<element name="finned" type="xsd:boolean"
         nillable="true" />
```

- An `element` element with its `minOccurs` attribute set to `0` and its `maxOccurs` attribute set to `1`, or its `maxOccurs` attribute not specified, as shown :

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- An `attribute` element with its `use` attribute set to `optional`, or not specified, and having neither its `default` attribute nor its `fixed` attribute specified, as shown:

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
               use="optional" />
```

```
  </complexType>
</element>
```

Table 16 on page 156 shows how XML Schema primitive types are mapped into Java wrapper classes in these cases.

Table 16. Primitive Schema Type to Java Wrapper Class Mapping

Schema Type	Java Type
xsd:int	`java.lang.Integer`
xsd:long	`java.lang.Long`
xsd:short	`java.lang.Short`
xsd:float	`java.lang.Float`
xsd:double	`java.lang.Double`
xsd:boolean	`java.lang.Boolean`
xsd:byte	`java.lang.Byte`
xsd:unsignedByte	`java.lang.Short`
xsd:unsignedShort	`java.lang.Integer`
xsd:unsignedInt	`java.lang.Long`
xsd:unsignedLong	`java.math.BigInteger`
xsd:duration	`java.lang.String`

Simple Types Defined by Restriction

Overview

XML Schema allows you to create simple types by deriving a new type from another primitive type or simple type. Simple types are described using a `simpleType` element.

The new types are described by restricting the *base type* with one or more facets. These facets limit the possible valid values that can be stored in the new type. For example, you could define a simple type, SSN, which is a string of exactly 9 characters.

Each of the primitive XML Schema types has their own set of optional facets.

Procedure

To define your own simple type do the following:

1. Determine the base type for your new simple type.
2. Determine what restrictions define the new type based on the available facets for the chosen base type.
3. Using the syntax shown in this section, enter the appropriate `simpleType` element into the types section of your contract.

Defining a simple type in XML Schema

Example 69 on page 157 shows the syntax for describing a simple type.

Example 69. Simple Type Syntax

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value" />
    <facet value="value" />
    ...
  </restriction>
</simpleType>
```

The type description is enclosed in a `simpleType` element and identified by the value of the `name` attribute. The base type from which the new simple type is being defined is specified by the `base` attribute of the `xsd:restriction` element. Each facet element is specified within the `restriction` element. The available facets and their valid settings depend on the base type. For example, xsd:string has a number of facets including:

- length
- minLength
- maxLength
- pattern
- whitespace

Example 70 on page 158 shows the definition for a simple type that represents the two-letter postal code used for US states. It can only contain two, uppercase letters. TX is a valid value, but tx or tX are not valid values.

Example 70. Postal Code Simple Type

```
<xsd:simpleType name="postalCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

Mapping to Java

Artix ESB maps user-defined simple types to the Java type of the simple type's base type. So, any message using the simple type postalCode, shown in Example 70 on page 158, is mapped to a `String` because the base type of postalCode is xsd:string. For example, the WSDL fragment shown in Example 71 on page 158 results in a Java method, `state()`, that takes a parameter, *`postalCode`*, of `String`.

Example 71. Credit Request with Simple Types

```
<message name="stateRequest">
  <part name="postalCode" type="postalCode" />
</message>
...
<portType name="postalSupport">
  <operation name="state">
    <input message="tns:stateRequest" name="stateRec" />
    <output message="tns:stateResponse" name="credResp" />
```

```
  </operation>
</portType>
```

Enforcing facets

By default, Artix ESB does not enforce any of the facets that are used to restrict a simple type. However, you can configure Artix ESB endpoints to enforce the facets by enabling schema validation.

To configure Artix ESB endpoints to use schema validation set the schema-validation-enabled property to `true`. Example 72 on page 159 shows the configuration for a service provider that uses schema validation

Example 72. Service Provider Configured to Use Schema Validation

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

For more information on configuring Artix ESB see Artix® ESB Deployment Guide.

Enumerations

Overview

In XML Schema, enumerated types are simple types that are defined using the `xsd:enumeration` facet. Unlike atomic simple types, they are mapped to Java enums.

Defining an enumerated type in XML Schema

Enumerations are a simple type using the `xsd:enumeration` facet. Each `xsd:enumeration` facet defines one possible value for the enumerated type.

Example 73 on page 160 shows the definition for an enumerated type. It has the following possible values:

- `big`
- `large`
- `mungo`
- `gargantuan`

Example 73. XML Schema Defined Enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
    <enumeration value="gargantuan"/>
  </restriction>
```

Mapping to Java

XML Schema enumerations where the base type is xsd:string are automatically mapped to Java enum type. You can instruct the code generator to map enumerations with other base types to Java enum types by using the customizations described in Customizing Enumeration Mapping on page 253.

The enum type is created as follows:

1. The name of the type is taken from the `name` attribute of the simple type definition and converted to a Java identifier.

In general, this means converting the first character of the XML Schema's name to an uppercase letter. If the first character of the XML Schema's name is an invalid character, an underscrore (_) is prepended to the name.

2. For each `enumeration` facet, an enum constant is generated based on the value of the `value` attribute.

   The constant's name is derived by converting all of the lowercase letters in the value to their uppercase equivalent.

3. A constructor is generated that takes the Java type mapped from the enumeration's base type.

4. A public method called `value()` is generated to access the facet value that is represented by an instance of the type.

   The return type of the `value()` method is the base type of the XML Schema type.

5. A public method called `fromValue()` is generated to create an instance of the enum type based on a facet value.

   The parameter type of the `value()` method is the base type of the XML Schema type.

6. The class is decorated with the `@XmlEnum` annotation.

The enumerated type defined in Example 73 on page 160 is mapped to the enum type shown in Example 74 on page 161.

Example 74. Generated Enumerated Type for a String Bases XML Schema Enumeration

```
@XmlType(name = "widgetSize")
@XmlEnum
public enum WidgetSize {

    @XmlEnumValue("big")
    BIG("big"),
    @XmlEnumValue("large")
    LARGE("large"),
    @XmlEnumValue("mungo")
    MUNGO("mungo"),
```

```
    @XmlEnumValue("gargantuan")
    GARGANTUAN("gargantuan");
    private final String value;

    WidgetSize(String v) {
        value = v;
    }

    public String value() {
        return value;
    }

    public static WidgetSize fromValue(String v) {
        for (WidgetSize c: WidgetSize.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }

}
```

Lists

Overview

XML Schema supports a mechanism for defining data types that are a list of space separated simple types. An example of an element, `primeList`, using a list type is shown in Example 75 on page 163.

Example 75. List Type Example

```
<primeList>1 3 5 7 9 11 13<\primeList>
```

XML Schema list types are generally mapped to Java `List<T>` objects. The only variation to this pattern is when a message part is mapped directly to an instance of an XML Schema list type.

Defining list types in XML Schema

XML Schema list types are simple types and as such are defined using a `simpleType` element. The most common syntax used to define a list type is shown in Example 76 on page 163.

Example 76. Syntax for XML Schema List Types

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

The value given for *`atomicType`* defines the type of the elements in the list. It can only be one of the built in XML Schema atomic types, like xsd:int or xsd:string, or a user-defined simple type that is not a list.

In addition to defining the type of elements listed in the list type, you can also use facets to further constrain the properties of the list type. Table 17 on page 163 shows the facets used by list types.

Table 17. List Type Facets

Facet	Effect
`length`	Defines the number of elements in an instance of the list type.
`minLength`	Defines the minimum number of elements allowed in an instance of the list type.

Facet	Effect
maxLength	Defines the maximum number of elements allowed in an instance of the list type.
enumeration	Defines the allowable values for elements in an instance of the list type.
pattern	Defines the lexical form of the elements in an instance of the list type. Patterns are defined using regular expressions.

For example, the definition for the simpleList element shown in Example 75 on page 163, is shown in Example 77 on page 164.

Example 77. Definition of a List Type

```
<simpleType name="primeListType">
  <list itemType="int"/>
</simpleType>
<element name="primeList" type="primeListType"/>
```

In addition to the syntax shown in Example 76 on page 163 you can also define a list type using the less common syntax shown in Example 78 on page 164.

Example 78. Alternate Syntax for List Types

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
    </simpleType>
  </list>
  </simpleType>
```

Mapping list type elements to Java

When an element is defined a list type, the list type is mapped to a collection property. A collection property is a Java List<T> object. The template class used by the List<T> is the wrapper class mapped from the list's base type. For example, the list type defined in Example 77 on page 164 is mapped to a List<Integer>.

For more information on wrapper type mapping see Wrapper classes on page 155.

Mapping list type parameters to Java

When a message part is defined as a list type, or is mapped to an element of a list type, the resulting method parameter is mapped to an array instead of a `List<T>` object. The base type of the array is the wrapper class of the list type's base class.

For example, the WSDL fragment in Example 79 on page 165 results in the method signature shown in Example 80 on page 165.

Example 79. WSDL with a List Type Message Part

```
<definitions ...>
  ...
  <types ...>
    <schema ... >
      <simpleType name="primeListType">
        <list itemType="int"/>
      </simpleType>
      <element name="primeList" type="primeListType"/>
    </schemas>
  </types>
  <message name="numRequest">
    <part name="inputData" element="xsd1:primeList" />
  </message>
  <message name="numResponse">;
    <part name="outputData" type="xsd:int">
  ...
  <portType name="numberService">
    <operation name="primeProcessor">
      <input name="numRequest" message="tns:numRequest" />
      <output name="numResponse" message="tns:numResponse" />
    </operation>
    ...
  </portType>
  ...
</definitions>
```

Example 80. Java Method with a List Type Parameter

```
public interface NumberService {

    @XmlList
    @WebResult(name = "outputData", targetNamespace = "", partName = "outputData")
    @WebMethod
    public int primeProcessor(
```

```
        @WebParam(partName = "inputData", name = "primeList", targetNamespace = "...")
        java.lang.Integer[] inputData
    );
}
```

Unions

Overview

In XML Schema, a union is a construct that allows you to describe a type whose data can be one of a number of simple types. For example, you can define a type whose value is either the integer `1` or the string `first`. Unions are mapped to Java `String`s.

Defining in XML Schema

XML Schema unions are defined using a `simpleType` element. They contain at least one `union` element that defines the member types of the union. The member types of the union are the valid types of data that can be stored in an instance of the union. They are defined using the `union` element's `memberTypes` attribute. The value of the `memberTypes` attribute contains a list of one or more defined simple type names. Example 81 on page 167 shows the definition of a union that can store either an integer or a string.

Example 81. Simple Union Type

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

In addition to specifying named types as a member type of a union, you can also define an anonymous simple type as a member type of a union. This is done by adding the anonymous type definition inside of the `union` element. Example 82 on page 167 shows an example of a union containing an anonymous member type that restricts the possible values of a valid integer to the range 1 through 10.

Example 82. Union with an Anonymous Member Type

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
        <maxInclusive value="10" />
      </restriction>
    </simpleType>
```

```
  </union>
</simpleType>
```

Mapping to Java

XML Schema union types are mapped to Java `String` objects. By default, Artix ESB does not validate the contents of the generated object. To have Artix ESB validate the contents you will must configure the runtime to use schema validation as described in Enforcing facets on page 159.

Simple Type Substitution

Overview

XML allows for simple type substitution between compatible types using the `xsi:type` attribute. The default mapping of simple types to Java primitive types, however, does not fully support simple type substitution. The runtime can handle basic simple type substitution, but information is lost. The code generators can be customized to generate Java classes that facilitate lossless simple type substitution.

Default mapping and marshaling

Because Java primitive types do not support type substitution, the default mapping of simple types to Java primitive types presents problems for supporting simple type substitution. The Java virtual machine will balk if an attempt is made to pass a short into a variable that expects an int even though the schema defining the types allows it.

To get around the limitations imposed by the Java type system, Artix ESB allows for simple type substitution when the value of the element's `xsi:type` attribute meets one of the following conditions:

- It specifies a primitive type that is compatible with the element's schema type.
- It specifies a type that derives by restriction from the element's schema type.
- It specifies a complex type that derives by extension from the element's schema type.

When the runtime does the type substitution it does not retain any knowledge of the type specified in the element's `xsi:type` attribute. If the type substitution is from a complex type to a simple type, only the value directly related to the simple type is preserved. Any other elements and attributes added by extension are lost.

Supporting lossless type substitution

You can customize the generation of simple types to facilitate lossless support of simple type substitution in the following ways:

- Set the `globalBindings` customization element's `mapSimpleTypeDef` to `true`.

This instructs the code generator to create Java value classes for all named simple types defined in the global scope.

For more information see Generating Java Classes for Simple Types on page 251.

- Add a `javaType` element to the `globalBindings` customization element.

  This instructs the code generators to map all instances of an XML Schema primitive type to s specific class of object.

  For more information see Specifying the Java Class of an XML Schema Primitive on page 243.

- Add a `baseType` customization element to the specific elements you want to customize.

  The `baseType` customization element allows you to specify the Java type generated to represent a property. To ensure the best compatibility for simple type substitution, use `java.lang.Object` as the base type.

  For more information see Specifying the Base Type of an Element or an Attribute on page 262.

Using Complex Types

Complex types can contain multiple elements and they can have attributes. They are mapped into Java classes that can hold the data represented by the type definition. Typically, the mapping is to a bean with a set of properties representing the elements and the attributes of the content model..

Basic Complex Type Mapping

Overview

XML Schema complex types define constructs containing more complex information than a simple type. The most simple complex types define an empty element with an attribute. More intricate complex types are made up of a collection of elements.

By default, an XML Schema complex type is mapped to a Java class, with a member variable to represent each element and attribute listed in the XML Schema definition. The class has setters and getters for each member variable.

Defining in XML Schema

XML Schema complex types are defined using the `complexType` element. The `complexType` element wraps the rest of elements used to define the structure of the data. It can appear either as the parent element of a named type definition, or as the child of an `element` element anonymously defining the structure of the information stored in the element. When the `complexType` element is used to define a named type, it requires the use of the `name` attribute. The `name` attribute specifies a unique identifier for referencing the type.

Complex type definitions that contain one or more elements have one of the child elements described in Table 18 on page 172. These elements determine how the specified elements appear in an instance of the type.

Table 18. Elements for Defining How Elements Appear in a Complex Type

Element	Description
`all`	All of the elements defined as part of the complex type must appear in an instance of the type. However, they can appear in any order.
`choice`	Only one of the elements defined as part of the complex type can appear in an instance of the type.
`sequence`	All of the elements defined as part of the complex type must appear in an instance of the type, and they must also appear in the order specified in the type definition.

Note

If a complex type definition only uses attributes, you do not need one of the elements described in Table 18 on page 172.

After deciding how the elements will appear, you define the elements by adding one or more `element` element children to the definition.

Example 83 on page 173 shows a complex type definition in XML Schema.

Example 83. XML Schema Complex Type

```
<complexType name="sequence">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="street" type="xsd:short" />
    <element name="city" type="xsd:string" />
    <element name="state" type="xsd:string" />
    <element name="zipCode" type="xsd:string" />
  </sequence>
</complexType>
```

Mapping to Java

XML Schema complex types are mapped to Java classes. Each element in the complex type definition is mapped to a member variable in the Java class. Getter and setter methods are also generated for each element in the complex type.

All generated Java classes are decorated with the `@XmlType` annotation. If the mapping is for a named complex type, the annotations name is set to the value of the `complexType` element's `name` attribute. If the complex type is defined as part of an element definition, the value of the `@XmlType` annotation's name property is the value of the `element` element's `name` attribute.

Note

As described in Java mapping of elements with an in-line type on page 151, the generated class is decorated with the `@XmlRootElement` annotation if it is generated for a complex type defined as part of an element definition.

To provide the runtime with guidelines indicating how the elements of the XML Schema complex type should be handled, the code generators alter the annotations used to decorate the class and its member variables.

All Complex Type

All complex types are defined using the `all` element. They are annotated as follows:

- The `@XmlType` annotation's propOrder property is empty.
- Each element is decorated with the `@XmlElement` annotation.
- The `@XmlElement` annotation's required property is set to `true`.

Example 84 on page 174 shows the mapping for an all complex type with two elements.

Example 84. Mapping of an All Complex Type

```
@XmlType(name = "all", propOrder = {

})
public class All {
    @XmlElement(required = true)
    protected BigDecimal amount;
    @XmlElement(required = true)
    protected String type;

    public BigDecimal getAmount() {
        return amount;
    }

    public void setAmount(BigDecimal value) {
        this.amount = value;
    }

    public String getType() {
        return type;
    }

    public void setType(String value) {
        this.type = value;
    }
}
```

Choice Complex Type

Choice complex types are defined using the `choice` element. They are annotated as follows:

- The `@XmlType` annotation's propOrder property lists the names of the elements in the order they appear in the XML Schema definition.
- None of the member variables are annotated.

Example 85 on page 175 shows the mapping for a choice complex type with two elements.

Example 85. Mapping of a Choice Complex Type

```
@XmlType(name = "choice", propOrder = {
    "address",
    "floater"
})
public class Choice {

    protected Sequence address;
    protected Float floater;

    public Sequence getAddress() {
        return address;
    }

    public void setAddress(Sequence value) {
        this.address = value;
    }

    public Float getFloater() {
        return floater;
    }

    public void setFloater(Float value) {
        this.floater = value;
    }

}
```

Sequence Complex Type

A sequence complex type is defined using the `sequence` element. It is annotated as follows:

- The `@XmlType` annotation's propOrder property lists the names of the elements in the order they appear in the XML Schema definition.
- Each element is decorated with the `@XmlElement` annotation.
- The `@XmlElement` annotation's required property is set to `true`.

Example 86 on page 176 shows the mapping for the complex type defined in Example 83 on page 173.

Example 86. Mapping of a Sequence Complex Type

```
@XmlType(name = "sequence", propOrder = {
    "name",
    "street",
    "city",
    "state",
    "zipCode"
})
public class Sequence {

    @XmlElement(required = true)
    protected String name;
    protected short street;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String state;
    @XmlElement(required = true)
    protected String zipCode;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public short getStreet() {
        return street;
    }

    public void setStreet(short value) {
        this.street = value;
    }

    public String getCity() {
```

```
        return city;
    }

    public void setCity(String value) {
        this.city = value;
    }

    public String getState() {
        return state;
    }

    public void setState(String value) {
        this.state = value;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String value) {
        this.zipCode = value;
    }
}
```

Attributes

Overview

Artix ESB supports the use of `attribute` elements and `attributeGroup` elements within the scope of a `complexType` element. When defining structures for an XML document attribute declarations provide a means of adding information that is specified within the tag, not the value that the tag contains. For example, when describing the XML element <value currency="euro">410<\value> in XML Schema the `currency` attribute is described using an `attribute` element as shown in Example 87 on page 179.

The `attributeGroup` element allows you to define a group of reusable attributes that can be referenced by all complex types defined by the schema. For example, if you are defining a series of elements that all use the attributes `category` and `pubDate`, you could define an attribute group with these attributes and reference them in all the elements that use them. This is shown in Example 89 on page 180.

When describing data types for use in developing application logic, attributes whose `use` attribute is set to either `optional` or `required` are treated as elements of a structure. For each attribute declaration contained within a complex type description, an element is generated in the class for the attribute, along with the appropriate getter and setter methods.

Defining an attribute in XML Schema

An XML Schema `attribute` element has one required attribute, `name`, that is used to identify the attribute. It also has four optional attributes that are described in Table 19 on page 178.

Table 19. Optional Attributes Used to Define Attributes in XML Schema

Attribute	Description
`use`	Specifies if the attribute is required. Valid values are `required`, `optional`, or `prohibited`. `optional` is the default value.
`type`	Specifies the type of value the attribute can take. If it is not used the schema type of the attribute must be defined in-line.
`default`	Specifies a default value to use for the attribute. It is only used when the `attribute` element's `use` attribute is set to `optional`.

Attribute	Description
`fixed`	Specifies a fixed value to use for the attribute. It is only used when the `attribute` element's `use` attribute is set to `optional`.

Example 87 on page 179 shows an attribute element defining an attribute, currency, whose value is a string.

Example 87. XML Schema Defining and Attribute

```
<element name="value">
  <complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="currency" type="xsd:string"
                       use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

If the `type` attribute is omitted from the `attribute` element, the format of the data must be described in-line. Example 88 on page 179 shows an `attribute` element for an attribute, `category`, that can take the values `autobiography`, `non-fiction`, or `fiction`.

Example 88. Attribute with an In-Line Data Description

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

Using an attribute group in XML Schema

Using an attribute group in a complex type definition is a two step process:

1. Define the attribute group.

   An attribute group is defined using an `attributeGroup` element with a number of `attribute` child elements. The `attributeGroup` requires

a `name` attribute that defines the string used to refer to the attribute group. The `attribute` elements define the members of the attribute group and are specified as shown in Defining an attribute in XML Schema on page 178. Example 89 on page 180 shows the description of the attribute group `catalogIndecies`. The attribute group has two members: `category`, which is optional, and `pubDate`, which is required.

Example 89. Attribute Group Definition

```
<attributeGroup name="catalogIndices">
  <attribute name="category" type="catagoryType" />
  <attribute name="pubDate" type="dateTime"
             use="required" />
</attributeGroup>
```

2. Use the attribute group in the definition of a complex type.

   You use attribute groups in complex type definitions by using the `attributeGroup` element with the `ref` attribute. The value of the `ref` attribute is the name given the attribute group that you want to use as part of the type definition. For example if you want to use the attribute group `catalogIndecies` in the complex type dvdType, you would use <attributeGroup ref="catalogIndecies" /> as shown in Example 90 on page 180.

   Example 90. Complex Type with an Attribute Group

```
<complexType name="dvdType">
  <sequence>
    <element name="title" type="xsd:string" />
    <element name="director" type="xsd:string" />
    <element name="numCopies" type="xsd:int" />
  </sequence>
  <attributeGroup ref="catalogIndices" />
</complexType>
```

Mapping attributes to Java

Attributes are mapped to Java in much the same way that member elements are mapped to Java. Required attributes and optional attributes are mapped to member variables in the generated Java class. The member variables are decorated with the `@XmlAttribute` annotation. If the attribute is required, the `@XmlAttribute` annotation's required property is set to `true`.

The complex type defined in Example 91 on page 181 is mapped to the Java class shown in Example 92 on page 181.

Example 91. techDoc Description

```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  </all>
  <attribute name="usefullness" type="xsd:float"
             use="optional" default="0.01" />
</complexType>
```

Example 92. techDoc Java Class

```
@XmlType(name = "techDoc", propOrder = {

})
public class TechDoc {

    @XmlElement(required = true)
    protected String product;
    protected short version;
    @XmlAttribute
    protected Float usefullness;

    public String getProduct() {
        return product;
    }

    public void setProduct(String value) {
        this.product = value;
    }

    public short getVersion() {
        return version;
    }

    public void setVersion(short value) {
        this.version = value;
    }

    public float getUsefullness() {
        if (usefullness == null) {
            return  0.01F;
        } else {
            return usefullness;
        }
```

```
    }

    public void setUsefullness(Float value) {
        this.usefullness = value;
    }
}
```

As shown in Example 92 on page 181, the `default` attribute and the `fixed` attribute instruct the code generators to add code to the getter method generated for the attribute. This additional code ensures that the specified value is returned if no value is set.

Important

The `fixed` attribute is treated the same as the `default` attribute.

If you want the `fixed` attribute to be treated as a Java constant you can use the customization described in Customizing Fixed Value Attribute Mapping on page 258.

Mapping attribute groups to Java

Attribute groups are mapped to Java as if the members of the group were explicitly used in the type definition. If the attribute group has three members, and it is used in a complex type, the generated class for that type will include a member variable, along with the getter and setter methods, for each member of the attribute group. For example, the complex type defined in Example 90 on page 180, Artix ESB generates a class containing the member variables `category` and `pubDate` to support the members of the attribute group as shown in Example 93 on page 182.

Example 93. dvdType Java Class

```
@XmlType(name = "dvdType", propOrder = {
    "title",
    "director",
    "numCopies"
})
public class DvdType {

    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String director;
    protected int numCopies;
    @XmlAttribute
    protected CatagoryType category;
```

```
    @XmlAttribute(required = true)
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar pubDate;

    public String getTitle() {
        return title;
    }

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

    public String getDirector() {
        return director;
    }

    public void setDirector(String value) {
        this.director = value;
    }

    public int getNumCopies() {
        return numCopies;
    }

    public void setNumCopies(int value) {
        this.numCopies = value;
    }

    public CatagoryType getCatagory() {
        return catagory;
    }

    public void setCatagory(CatagoryType value) {
        this.catagory = value;
    }

    public XMLGregorianCalendar getPubDate() {
        return pubDate;
    }

    public void setPubDate(XMLGregorianCalendar value) {
        this.pubDate = value;
    }

}
```

Deriving Complex Types from Simple Types

Overview

Artix ESB supports derivation of a complex type from a simple type. A simple type has, by definition, neither sub-elements nor attributes. Hence, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type.

There are two ways of deriving a complex type from a simple type:

- By extension
- By restriction

Derivation by extension

Example 94 on page 184 shows an example of a complex type, internationalPrice, derived by extension from the xsd:decimal primitive type to include a currency attribute.

Example 94. Deriving a Complex Type from a Simple Type by Extension

```
<complexType name="internationalPrice">
    <simpleContent>
        <extension base="xsd:decimal">
            <attribute name="currency" type="xsd:string"/>
        </extension>
    </simpleContent>
    </complexType>
```

The `simpleContent` element indicates that the new type does not contain any sub-elements. The `extension` element specifies that the new type extends xsd:decimal.

Derivation by restriction

Example 95 on page 184 shows an example of a complex type, idType, that is derived by restriction from xsd:string. The defined type restricts the possible values of xsd:stringto values that are ten characters in length. It also adds an attribute to the type.

Example 95. Deriving a Complex Type from a Simple Type by Restriction

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
     <attribute name="expires" type="xsd:dateTime" />
```

```
    </restriction>
  </simpleContent>
</complexType>
```

As in Example 94 on page 184 the `simpleContent` element signals that the new type does not contain any children. This example uses a `restriction` element to constrain the possible values used in the new type. The `attribute` element adds the element to the new type.

Mapping to Java

A complex type derived from a simple type is mapped to a Java class that is decorated with the `@XmlType` annotation. The generated class contains a member variable, `value`, of the simple type from which the complex type is derived. The member variable is decorated with the `@XmlValue` annotation. The class also has a `getValue()` method and a `setValue()` method. In addition, the generated class has a member variable, and the associated getter and setter methods, for each attribute that extends the simple type.

Example 96 on page 185 shows the Java class generated for the idType type defined in Example 95 on page 184.

Example 96. idType Java Class

```
@XmlType(name = "idType", propOrder = {
    "value"
})
public class IdType {

    @XmlValue
    protected String value;
    @XmlAttribute
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar expires;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public XMLGregorianCalendar getExpires() {
        return expires;
    }
```

```
    public void setExpires(XMLGregorianCalendar value) {
        this.expires = value;
    }

}
```

Deriving Complex Types from Complex Types

Overview

Using XML Schema, you can derive new complex types by either extending or restricting other complex types using the `complexContent` element. When generating the Java class to represent the derived complex type, Artix ESB extends the base type's class. In this way, the generated Java code preserves the inheritance hierarchy intended in the XML Schema.

Schema syntax

You derive complex types from other complex types by using the `complexContent` element, and either the `extension` element or the `restriction` element. The `complexContent` element specifies that the included data description includes more than one field. The `extension` element and the `restriction` element, which are children of the `complexContent` element, specify the base type being modified to create the new type. The base type is specified by the `base` attribute.

Extending a complex type

To extend a complex type use the `extension` element to define the additional elements and attributes that make up the new type. All elements that are allowed in a complex type description are allowable as part of the new type's definition. For example, you can add an anonymous enumeration to the new type, or you can use the `choice` element to specify that only one of the new fields can be valid at a time.

Example 97 on page 187 shows an XML Schema fragment that defines two complex types, widgetOrderInfo and widgetOrderBillInfo. widgetOrderBillInfo is derived by extending widgetOrderInfo to include two new elements: `orderNumber` and `amtDue`.

Example 97. Deriving a Complex Type by Extension

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:boolean" use="optional" />
</complexType>
<complexType name="widgetOrderBillInfo">
```

```
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:decimal"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
      <attribute name="paid" type="xsd:boolean"
                 default="false" />
    </extension>
  </complexContent>
</complexType>
```

Restricting a complex type

To restrict a complex type use the `restriction` element to limit the possible values of the base type's elements or attributes. When restricting a complex type you must list all of the elements and attributes of the base type. For each element you can add restrictive attributes to the definition. For example, you can add a `maxOccurs` attribute to an element to limit the number of times it can occur. You can also use the `fixed` attribute to force one or more of the elements to have predetermined values.

Example 98 on page 188 shows an example of defining a complex type by restricting another complex type. The restricted type, wallawallaAddress, can only be used for addresses in Walla Walla, Washington because the values for the `city` element, the `state` element, and the `zipCode` element are fixed.

Example 98. Defining a Complex Type by Restriction

```
<complexType name="Address">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="street" type="xsd:short" maxOccurs="3"/>
    <element name="city" type="xsd:string"/>
    <element name="state" type="xsd:string"/>
    <element name="zipCode" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="wallawallaAddress">
  <complexContent>
    <restriction base="xsd1:Address">
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="street" type="xsd:short"
                 maxOccurs="3"/>
        <element name="city" type="xsd:string"
                 fixed="WallaWalla"/>
```

```
        <element name="state" type="xsd:string"
                 fixed="WA" />
        <element name="zipCode" type="xsd:string"
                 fixed="99362" />
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

Mapping to Java

As it does with all complex types, Artix ESB generates a class to represent complex types derived from another complex type. The Java class generated for the derived complex type extends the Java class generated to support the base complex type. The base Java class is also modified to include the `@XmlSeeAlso` annotation. The base class' `@XmlSeeAlso` annotation lists all of the classes that extend the base class.

When the new complex type is derived by extension, the generated class will include member variables for all of the added elements and attributes. The new member variables will be generated according to the same mappings as all other elements.

When the new complex type is derived by restriction, the generated class will have no new member variables. The generated class will simply be a shell that does not provide any additional functionality. It is entirely up to you to ensure that the restrictions defined in the XML Schema are enforced.

For example, the schema in Example 97 on page 187 results in the generation of two Java classes: `WidgetOrderInfo` and `WidgetBillOrderInfo`.

`WidgetOrderBillInfo` extends `WidgetOrderInfo` because widgetOrderBillInfo is derived by extension from widgetOrderInfo. Example 99 on page 189 shows the generated class for widgetOrderBillInfo.

Example 99. WidgetOrderBillInfo

```
@XmlType(name = "widgetOrderBillInfo", propOrder = {
    "amtDue",
    "orderNumber"
})
public class WidgetOrderBillInfo
    extends WidgetOrderInfo
{
    @XmlElement(required = true)
    protected BigDecimal amtDue;
    @XmlElement(required = true)
    protected String orderNumber;
```

```
    @XmlAttribute
    protected Boolean paid;

    public BigDecimal getAmtDue() {
        return amtDue;
    }

    public void setAmtDue(BigDecimal value) {
        this.amtDue = value;
    }

    public String getOrderNumber() {
        return orderNumber;
    }

    public void setOrderNumber(String value) {
        this.orderNumber = value;
    }

    public boolean isPaid() {
        if (paid == null) {
            return false;
        } else {
            return paid;
        }
    }

    public void setPaid(Boolean value) {
        this.paid = value;
    }
}
```

Occurrence Constraints

XML Schema allows you to specify the occurrence constraints on four of the XML Schema elements that make up a complex type definition:

- `all`
- `choice`
- `element`
- `sequence`

Occurrence Constraints on the All Element

XML Schema

Complex types defined with the `all` element do not allow for multiple occurrences of the structure defined by the `all` element. You can, however, make the structure defined by the `all` element optional by setting its `minOccurs` attribute to `0`.

Mapping to Java

Setting the `all` element's `minOccurs` attribute to `0` has no effect on the generated Java class.

Occurrence Constraints on the Choice Element

Overview

By default, the results of a `choice` element can only appear once in an instance of a complex type. You can change the number of times the element chosen to represent the structure defined by a `choice` element is allowed to appear using its `minOccurs` attribute and its `mxOccurs` attribute. Using these attributes you can specify that the choice type can occur zero to an unlimited number of times in an instance of a complex type. The element chosen for the choice type does not need to be the same for each occurrence of the type.

Using in XML Schema

The `minOccurs` attribute specifies the minimum number of times the choice type must appear. Its value can be any positive integer. Setting the `minOccurs` attribute to `0` specifies that the choice type does not need to appear inside an instance of the complex type.

The `maxOccurs` attribute specifies the maximum number of times the choice type can appear. Its value can be any non-zero, positive integer or `unbounded`. Setting the `maxOccurs` attribute to `unbounded` specifies that the choice type can appear an infinite number of times.

Example 100 on page 193 shows the definition of a choice type, ClubEvent, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 100. Choice Occurrence Constraints

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

Mapping to Java

Unlike single instance choice structures, XML Schema choice structures that can occur multiple times are mapped to a Java class with a single member variable. This single member variable is a `List<T>` object that holds all of the data for the multiple occurrences of the sequence. For example, if the sequence defined in Example 100 on page 193 occurred two times, then the list would have two items.

The name of the Java class' member variable is derived by concatenating the names of the member elements. The element names are separated by `Or` and the first letter of the variable name is converted to lower case. For example, the member variable generated from Example 100 on page 193 would be named `memberNameOrGuestName`.

The type of object stored in the list depends on the relationship between the types of the member elements. For example:

- If the member elements are of the same type the generated list will contain `JAXBElement<T>` objects. The base type of the `JAXBElement<T>` objects is determined by the normal mapping of the member elements' type.
- If the member elements are of different types and their Java representations implement a common interface, the list will contains objects of the common interface.
- If the member elements are of different types and their Java representations extend a common base class, the list will contains objects of the common base class.
- If none of the other conditions are met, the list will contain `Object` objects.

The generated Java class will only have a getter method for the member variable. The getter method returns a reference to the live list. Any modifications made to the returned list will effect the actual object.

The Java class is decorated with the `@XmlType` annotation. The annotation's name property is set to the value of the `name` attribute from the parent element of the XML Schema definition. The annotation's propOrder property contains the single member variable representing the elements in the sequence.

The member variable representing the elements in the choice structure are decorated with the `@XmlElements` annotation. The `@XmlElements` annotation contains a comma separated list of `@XmlElement` annotations. The list has one `@XmlElement` annotation for each member element defined in the XML Schema definition of the type. The `@XmlElement` annotations in the list have their name property set to the value of the XML Schema `element` element's `name` attribute and their type property set to the Java class resulting from the mapping of the XML Schema `element` element's type.

Example 101 on page 195 shows the Java mapping for the XML Schema choice structure defined in Example 100 on page 193.

Example 101. Java Representation of Choice Structure with an Occurrence Constraint

```
@XmlType(name = "ClubEvent", propOrder = {
    "memberNameOrGuestName"
})
public class ClubEvent {

    @XmlElementRefs({
        @XmlElementRef(name = "GuestName", type = JAXBElement.class),
        @XmlElementRef(name = "MemberName", type = JAXBElement.class)
    })
    protected List<JAXBElement<String>> memberNameOrGuestName;

    public List<JAXBElement<String>> getMemberNameOrGuestName() {
        if (memberNameOrGuestName == null) {
            memberNameOrGuestName = new ArrayList<JAXBElement<String>>();
        }
        return this.memberNameOrGuestName;
    }

}
```

minOccurs set to 0

If only the `minOccurs` element is specified and its value is `0`, the code generators generate the Java class as if the `minOccurs` attribute were not set.

Occurrence Constraints on Elements

Overview

You can specify how many times a specific element in a complex type appears using the `element` element's `minOccurs` attribute and `maxOccurs` attribute. The default value for both attributes is `1`.

minOccurs set to 0

When you set one of the complex type's member element's `minOccurs` attribute to `0`, the `@XmlElement` annotation decorating the corresponding Java member variable is changed. Instead of having its required property set to `true`, the `@XmlElement` annotation's required property is set to `false`.

minOccurs set to a value greater than 1

In XML Schema you can specify that an element must occur more than once in an instance of the type by setting the `element` element's `minOccurs` attribute to a value greater than one. However, the generated Java class will not support the XML Schema constraint. Artix ESB generates the supporting Java member variable as if the `minOccurs` attribute were not set.

Elements with maxOccurs set

When you want a member element to appear multiple times in an instance of a complex type, you set the element's `maxOccurs` attribute to a value greater than `1`. You can set the `maxOccurs` attribute's value to `unbounded` to specify that the member element can appear an unlimited number of times.

The code generators map a member element with the `maxOccurs` attribute set to a value greater than `1` to a Java member variable that is a `List<T>` object. The base class of the list is determined by mapping the element's type to Java. For XML Schema primitive types, the wrapper classes are used as described in Wrapper classes on page 155. For example, if the member element is of type xsd:int the generated member variable is a `List<Integer>` object.

Occurrence Constraints on Sequences

Overview

By default, the contents of a `sequence` element can only appear once in an instance of a complex type. You can change the number of times the sequence of elements defined by a `sequence` element is allowed to appear using its `minOccurs` attribute and its `maxOccurs` attribute. Using these attributes you can specify that the sequence type can occur zero to an unlimited number of times in an instance of a complex type.

Using XML Schema

The `minOccurs` attribute specifies the minimum number of times the sequence must occur in an instance of the defined complex type. Its value can be any positive integer. Setting the `minOccurs` attribute to `0` specifies that the sequence does not need to appear inside an instance of the complex type.

The `maxOccurs` attribute specifies the upper limit for how many times the sequence can occur in an instance of the defined complex type. Its value can be any non-zero, positive integer or `unbounded`. Setting the `maxOccurs` attribute to `unbounded` specifies that the sequence can appear an infinite number of times.

Example 102 on page 197 shows the definition of a sequence type, CultureInfo, with sequence occurrence constraints. The sequence can be repeated 0 to 2 times.

Example 102. Sequence with Occurrence Constraints

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

Mapping to Java

Unlike single instance sequences, XML Schema sequences that can occur multiple times are mapped to a Java class with a single member variable. This single member variable is a `List<T>` object that holds all of the data for the multiple occurrences of the sequence. For example, if the sequence defined in Example 102 on page 197 occurred two times, then the list would have four items.

The name of the Java class' member variable is derived by concatenating the names of the member elements. The element names are separated by `And` and the first letter of the variable name is converted to lower case. For example, the member variable generated from Example 102 on page 197 is named `nameAndLcid`.

The type of object stored in the list depends on the relationship between the types of the member elements. For example:

- If the member elements are of the same type the generated list will contain `JAXBElement<T>` objects. The base type of the `JAXBElement<T>` objects is determined by the normal mapping of the member elements' type.
- If the member elements are of different types and their Java representations implement a common interface, the list will contains objects of the common interface.
- If the member elements are of different types and their Java representations extend a common base class, the list will contain objects of the common base class.
- If none of the other conditions are met, the list will contain `Object` objects.

The generated Java class only has a getter method for the member variable. The getter method returns a reference to the live list. Any modifications made to the returned list effects the actual object.

The Java class is decorated with the `@XmlType` annotation. The annotation's name property is set to the value of the `name` attribute from the parent element of the XML Schema definition. The annotation's propOrder property contains the single member variable representing the elements in the sequence.

The member variable representing the elements in the sequence are decorated with the `@XmlElements` annotation. The `@XmlElements` annotation contains a comma separated list of `@XmlElement` annotations. The list has one `@XmlElement` annotation for each member element defined in the XML Schema definition of the type. The `@XmlElement` annotations in the list have their name property set to the value of the XML Schema `element` element's `name` attribute and their type property set to the Java class resulting from the mapping of the XML Schema `element` element's type.

Example 103 on page 199 shows the Java mapping for the XML Schema sequence defined in Example 102 on page 197.

Example 103. Java Representation of Sequence with an Occurrence Constraint

```
@XmlType(name = "CultureInfo", propOrder = {
    "nameAndLcid"
})
public class CultureInfo {

    @XmlElements({
        @XmlElement(name = "Name", type = String.class),
        @XmlElement(name = "Lcid", type = Integer.class)
    })
    protected List<Serializable> nameAndLcid;

    public List<Serializable> getNameAndLcid() {
        if (nameAndLcid == null) {
            nameAndLcid = new ArrayList<Serializable>();
        }
        return this.nameAndLcid;
    }

}
```

minOccurs set to 0

If only the `minOccurs` element is specified and its value is `0`, the code generators generate the Java class as if the `minOccurs` attribute is not set.

Using Model Groups

Overview

XML Schema model groups are convenient shortcuts that allows you to reference a group of elements from a user-defined complex type.For example, you can define a group of elements that are common to several types in your application and then reference the group repeatedly. Model groups are defined using the `group` element, and are similar to complex type definitions. The mapping of model groups to Java is also similar to the mapping for complex types.

Defining a model group in XML Schema

You define a model group in XML Schema using the `group` element with the `name` attribute. The value of the `name` attribute is a string that is used to refer to the group throughout the schema. The `group` element, like the `complexType` element, can have the `sequence` element, the `all` element, or the `choice` element as its immediate child.

Inside the child element, you define the members of the group using `element` elements. For each member of the group, specify one `element` element. Group members can use any of the standard attributes for the `element` element including `minOccurs` and `maxOccurs`. So, if your group has three elements and one of them can occur up to three times, you define a group with three `element` elements, one of which uses maxOccurs="3". Example 104 on page 200 shows a model group with three elements.

Example 104. XML Schema Model Group

```
<group name="passenger">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="clubNum" type="xsd:long" />
    <element name="seatPref" type="xsd:string"
             maxOccurs="3" />
  </sequence>
</group>
```

Using a model group in a type definition

Once a model group has been defined, it can be used as part of a complex type definition. To use a model group in a complex type definition, use the `group` element with the `ref` attribute. The value of the `ref` attribute is the

name given to the group when it was defined. For example, to use the group defined in Example 104 on page 200 you use <group ref="tns:passenger" /> as shown in Example 105 on page 201.

Example 105. Complex Type with a Model Group

```
<complexType name="reservation">
  <sequence>
    <group ref="tns:passenger" />
    <element name="origin" type="xsd:string" />
    <element name="destination" type="xsd:string" />
    <element name="fltNum" type="xsd:long" />
  </sequence>
</complexType>
```

When a model group is used in a type definition, the group becomes a member of the type. So an instance of reservation has four member elements. The first element is the `passenger` element and it contains the member elements defined by the group shown in Example 104 on page 200. An example of an instance of reservation is shown in Example 106 on page 201.

Example 106. Instance of a Type with a Model Group

```
<reservation>
  <passenger>
    <name>A. Smart</name>
    <clubNum>99</clubNum>
    <seatPref>isle1</seatPref>
  </passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

Mapping to Java

By default, a model group is only mapped to Java artifacts when it is included in a complex type definition. When generating code for a complex type that includes a model group, Artix ESB simply includes the member variables for the model group into the Java class generated for the type. The member variables representing the model group are annotated based on the definitions of the model group.

Example 107 on page 202 shows the Java class generated for the complex type defined in Example 105 on page 201.

Example 107. Type with a Group

```
@XmlType(name = "reservation", propOrder = {
    "name",
    "clubNum",
    "seatPref",
    "origin",
    "destination",
    "fltNum"
})
public class Reservation {

    @XmlElement(required = true)
    protected String name;
    protected long clubNum;
    @XmlElement(required = true)
    protected List<String> seatPref;
    @XmlElement(required = true)
    protected String origin;
    @XmlElement(required = true)
    protected String destination;
    protected long fltNum;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public long getClubNum() {
        return clubNum;
    }

    public void setClubNum(long value) {
        this.clubNum = value;
    }

    public List<String> getSeatPref() {
        if (seatPref == null) {
            seatPref = new ArrayList<String>();
        }
        return this.seatPref;
    }

    public String getOrigin() {
        return origin;
    }
```

```
public void setOrigin(String value) {
    this.origin = value;
}

public String getDestination() {
    return destination;
}

public void setDestination(String value) {
    this.destination = value;
}

public long getFltNum() {
    return fltNum;
}

public void setFltNum(long value) {
    this.fltNum = value;
}
```

Multiple occurrences

You can specify that the model group appears more than once by setting the `group` element's `maxOccurs` attribute to a value greater than one. To allow for multiple occurrences of the model group Artix ESB maps the model group to a `List<T>` object. The `List<T>` object is generated following the rules for the group's first child:

- If the group is defined using a `sequence` element see Occurrence Constraints on Sequences on page 197.
- If the group is defined using a `choice` element see Occurrence Constraints on the Choice Element on page 193.

Using Wild Card Types

There are instances when a schema author wants to defer binding elements or attributes to a defined type. For these cases, XML Schema provides three mechanisms for specifying wild card place holders. These are all mapped to Java in ways that preserve their XML Schema functionality.

Using Any Elements

Overview

The XML Schema `any` element is used to create a wild card place holder in complex type definitions. When an XML element is instantiated for an XML Schema `any` element, it can be any valid XML element. The `any` element does not place any restrictions on either the content or the name of the instantiated XML element.

For example, given the complex type defined in Example 108 on page 206 you can instantiate either of the XML elements shown in Example 109 on page 206.

Example 108. XML Schema Type Defined with an Any Element

```
<element name="FlyBoy">
  <complexType>
    <sequence>
      <any />
      <element name="rank" type="xsd:int" />
    </sequence>
  </complexType>
</element>
```

Example 109. XML Document with an Any Element

```
<FlyBoy>
  <learJet>CL-215</learJet>
  <rank>2</rank>
</element>
<FlyBoy>
  <viper>Mark II</viper>
  <rank>1</rank>
</element>
```

XML Schema `any` elements are mapped to either a Java `Object` object or a Java `org.w3c.dom.Element` object.

Specifying in XML Schema

The `any` element can be used when defining sequence complex types and choice complex types. In most cases, the `any` element is an empty element. It can, however, take an `annotation` element as a child.

Table 20 on page 207 describes the `any` element's attributes.

Table 20. Attributes of the XML Schema Any Element

Attribute	Description

`namespace`	Specifies the namespace of the elements that can be used to instantiate the element in an XML document. The valid values are:
	
	`##any`
	Specifies that elements from any namespace can be used. This is the default.
	
	`##other`
	Specifies that elements from any namespace *other than the parent element's namespace* can be used.
	
	`##local`
	Specifies elements without a namespace must be used.
	
	`##targetNamespace`
	Specifies that elements from the parent element's namespace must be used.
	
	A space delimited list of URIs, `##local`, and `##targetNamespace`
	Specifies that elements from any of the listed namespaces can be used.

`maxOccurs`	Specifies the maximum number of times an instance of the element can appear in the parent element. The default value is `1`. To specify that an instance of the element can appear an unlimited number of times, you can set the attribute's value to `unbounded`.
`minOccurs`	Specifies the minimum number of times an instance of the element can appear in the parent element. The default value is `1`.

`processContents`	Specifies how the element used to instantiate the any element should be validated. Valid values are:
	
	`strict`
	Specifies that the element must be validated against the proper schema. This is the default value.
	
	`lax`
	Specifies that the element should be validated against the proper schema. If it cannot be validated, no errors are thrown.
	
	`skip`
	Specifies that the element should not be validated.

Example 110 on page 208 shows a complex type defined with an `any` element

Example 110. Complex Type Defined with an Any Element

```
<complexType name="surprisePackage">
  <sequence>
    <any processContents="lax" />
    <element name="to" type="xsd:string" />
    <element name="from" type="xsd:string" />
  </sequence>
</complexType>
```

Mapping to Java

XML Schema `any` elements result in the creation of a Java property named `any`. The property has associated getter and setter methods. The type of the resulting property depends on the value of the element's `processContents` attribute. If the `any` element's `processContents` attribute is set to `skip`, the element is mapped to a `org.w3c.dom.Element` object. For all other values of the `processContents` attribute an `any` element is mapped to a Java `Object` object.

The generated property is decorated with the `@XmlAnyElement` annotation. This annotation has an optional lax property that instructs the runtime what to do when marshaling the data. Its default value is `false` which instructs the runtime to automatically marshal the data into a `org.w3c.dom.Element` object. Setting lax to `true` instructs the runtime to attempt to marshal the data into JAXB types. When the `any` element's `processContents` attribute is set to `skip`, the lax property is set to its default value. For all other values of the `processContents` attribute, lax is set to `true`.

Example 111 on page 208 shows how the complex type defined in Example 110 on page 208 is mapped to a Java class.

Example 111. Java Class with an Any Element

```
public class SurprisePackage {

    @XmlAnyElement(lax = true)
    protected Object any;
    @XmlElement(required = true)
    protected String to;
```

```
    @XmlElement(required = true)
    protected String from;

    public Object getAny() {
        return any;
    }

    public void setAny(Object value) {
        this.any = value;
    }

    public String getTo() {
        return to;
    }

    public void setTo(String value) {
        this.to = value;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String value) {
        this.from = value;
    }

}
```

Marshalling

If the Java property for an `any` element has its lax set to `false`, or the property is not specified, the runtime makes no attempt to parse the XML data into JAXB objects. The data is always stored in a DOM `Element` object.

If the Java property for an `any` element has its lax set to `true`, the runtime attempts to marshal the XML data into the appropriate JAXB objects. The runtime attempts to identify the proper JAXB classes using the following procedure:

1. It checks the element tag of the XML element against the list of elements known to the runtime. If it finds a match, the runtime marshals the XML data into the proper JAXB class for the element.
2. It checks the XML element's `xsi:type` attribute. If it finds a match, the runtime marshals the XML element into the proper JAXB class for that type.

3. If it cannot find a match it marshals the XML data into a DOM `Element` object.

Usually an application's runtime knows about all of the types generated from the schema's included in its contract. This includes the types defined in the contract's `wsdl:types` element, any data types added to the contract through inclusion, and any types added to the contract through importing other schemas. You can also make the runtime aware of additional types using the `@XmlSeeAlso` annotation which is described in Adding Classes to the Runtime Marshaller on page 144.

Unmarshalling

If the Java property for an `any` element has its lax set to `false`, or the property is not specified, the runtime will only accept DOM `Element` objects. Attempting to use any other type of object will result in a marshalling error.

If the Java property for an `any` element has its lax set to `true`, the runtime uses its internal map between Java data types and the XML Schema constructs they represent to determine the XML structure to write to the wire. If the runtime knows the class and can map it to an XML Schema construct, it writes out the data and inserts an `xsi:type` attribute to identify the type of data the element contains.

If the runtime cannot map the Java object to a known XML Schema construct, it will throw a marshaling exception. You can add types to the runtime's map using the `@XmlSeeAlso` annotation which is described in Adding Classes to the Runtime Marshaller on page 144.

Using the XML Schema anyType Type

Overview

The XML Schema type xsd:anyType is the root type for all XML Schema types. All of the primitives are derivatives of this type, as are all user defined complex types. As a result, elements defined as being of xsd:anyType can contain data in the form of any of the XML Schema primitives as well as any complex type defined in a schema document.

In Java the closest matching type is the `Object` class. It is the class from which all other Java classes are sub-typed.

Using in XML Schema

You use the xsd:anyType type as you would any other XML Schema complex type. It can be used as the value of an `element` element's `type` element. It can also be used as the base type from which other types are defined.

Example 112 on page 211 shows an example of a complex type that contains an element of type xsd:anyType.

Example 112. Complex Type with a Wild Card Element

```
<complexType name="wildStar">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="ship" type="xsd:anyType" />
  </sequence>
</complexType>
```

Mapping to Java

Elements that are of type xsd:anyType are mapped to `Object` objects.

Example 113 on page 211 shows the mapping of Example 112 on page 211 to a Java class.

Example 113. Java Representation of a Wild Card Element

```
public class WildStar {

    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected Object ship;

    public String getName() {
        return name;
    }
```

```
    public void setName(String value) {
        this.name = value;
    }

    public Object getShip() {
        return ship;
    }

    public void setShip(Object value) {
        this.ship = value;
    }
}
```

This mapping allows you to place any data into the property representing the wild card element. The Artix ESB runtime handles the marshaling and unmarshaling of the data into usable Java representation.

Marshalling

When Artix ESB marshals XML data into Java types, it attempts to marshal anyType elements into known JAXB objects. To determine if it is possible to marshal an anyType element into a JAXB generated object, the runtime inspects the element's `xsi:type` attribute to determine the actual type used to construct the data in the element. If the `xsi:type` attribute is not present, the runtime attempts to identify the element's actual data type by introspection. If the element's actual data type is determined to be one of the types known by the application's JAXB context, the element is marshaled into a JAXB object of the proper type.

If the runtime cannot determine the actual data type of the element, or the actual data type of the element is not a known type, the runtime marshals the content into a `org.w3c.dom.Element` object. You will then need to work with the element's content using the DOM APIs.

An application's runtime usually knows about all of the types generated from the schema's included in its contract. This includes the types defined in the contract's `wsdl:types` element, any data types added to the contract through inclusion, and any types added to the contract through importing other schema documents. You can also make the runtime aware of additional types using the `@XmlSeeAlso` annotation which is described in Adding Classes to the Runtime Marshaller on page 144.

Unmarshalling

When Artix ESB unmarshals Java types into XML data, it uses an internal map between Java data types and the XML Schema constructs they represent

to determine the XML structure to write to the wire. If the runtime knows the class and can map the class to an XML Schema construct, it writes out the data and inserts an `xsi:type` attribute to identify the type of data the element contains. If the data is stored in a `org.w3c.dom.Element` object, the runtime writes the XML structure represented by the object but it does not include an `xsi:type` attribute.

If the runtime cannot map the Java object to a known XML Schema construct, it throws a marshaling exception. You can add types to the runtime's map using the `@XmlSeeAlso` annotation which is described in Adding Classes to the Runtime Marshaller on page 144.

Using Unbound Attributes

Overview

XML Schema has a mechanism that allows you to leave a place holder for an arbitrary attribute in a complex type definition. Using this mechanism, you can define a complex type that can have any attribute. For example, you can create a type that defines the elements <robot name="epsilon" />, <robot age="10000" />, or <robot type="weevil" /> without specifying the three attributes. This can be particularly useful when flexibility in your data is required.

Defining in XML Schema

Undeclared attributes are defined in XML Schema using the `anyAttribute` element. It can be used wherever an attribute element can be used. The `anyAttribute` element has no attributes, as shown in Example 114 on page 214.

Example 114. Complex Type with an Undeclared Attribute

```
<complexType name="arbitter">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="rate" type="xsd:float" />
  </sequence>
  <anyAttribute />
</complexType>
```

The defined type, arbitter, has two elements and can have one attribute of any type. The elements three elements shown in Example 115 on page 214 can all be generated from the complex type arbitter.

Example 115. Examples of Elements Defined with a Wild Card Attribute

```
<officer rank="12"><name>...</name><rate>...</rate></officer>
<lawyer type="divorce"><name>...</name><rate>...</rate></law
yer>
<judge><name>...</name><rate>...</rate></judge>
```

Mapping to Java

When a complex type containing an `anyAttribute` element is mapped to Java, the code generator adds a member called `otherAttributes` to the generated class. `otherAttributes` is of type `java.util.Map<QName, String>` and it has a getter method that returns a live instance of the map. Because the map returned from the getter is live, any modifications to the

map are automatically applied. Example 116 on page 215 shows the class generated for the complex type defined in Example 114 on page 214.

Example 116. Class for a Complex Type with an Undeclared Attribute

```
public class Arbitter {

    @XmlElement(required = true)
    protected String name;
    protected float rate;

    @XmlAnyAttribute
    private Map<QName, String> otherAttributes = new HashMap<QName, String>();

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public float getRate() {
        return rate;
    }

    public void setRate(float value) {
        this.rate = value;
    }

    public Map<QName, String> getOtherAttributes() {
        return otherAttributes;
    }

}
```

Working with undeclared attributes

The `otherAttributes` member of the generated class expects to be populated with a `Map` object. The map is keyed using `QNames`. Once you get the map , you can access any attributes set on the object and set new attributes on the object.

Example 117 on page 216 shows sample code for working with undeclared attributes.

Example 117. Working with Undeclared Attributes

```
Arbitter judge = new Arbitter();
Map<QName, String> otherAtts = judge.getOtherAttributes(); ❶

QName at1 = new QName("test.apache.org", "house"); ❷
QName at2 = new QName("test.apache.org", "veteran");

otherAtts.put(at1, "Cape"); ❸
otherAtts.put(at2, "false");

String vetStatus = otherAtts.get(at2); ❹
```

The code in Example 117 on page 216 does the following:

❶ Gets the map containing the undeclared attributes.

❷ Creates QNames to work with the attributes.

❸ Sets the values for the attributes into the map.

❷ Retrieves the value for one of the attributes.

Element Substitution

XML Schema substitution groups allow you to define a group of elements that can replace a top level, or head, element. This is useful in cases where you have multiple elements that share a common base type or with elements that need to be interchangeable.

Substitution Groups in XML Schema

Overview

A substitution group is a feature of XML schema that allows you to specify elements that can replace another element in documents generated from that schema. The replaceable element is called the head element and must be defined in the schema's global scope. The elements of the substitution group must be of the same type as the head element or a type that is derived from the head element's type.

In essence, a substitution group allows you to build a collection of elements that can be specified using a generic element. For example, if you are building an ordering system for a company that sells three types of widgets you might define a generic widget element that contains a set of common data for all three widget types. Then you can define a substitution group that contains a more specific set of data for each type of widget. In your contract you can then specify the generic widget element as a message part instead of defining a specific ordering operation for each type of widget. When the actual message is built, the message can contain any of the elements of the substitution group.

Syntax

Substitution groups are defined using the `substitutionGroup` attribute of the XML Schema `element` element. The value of the `substitutionGroup` attribute is the name of the element that the element being defined replaces. For example, if your head element is `widget`, adding the attribute substitutionGroup="widget" to an element named `woodWidget` specifies that anywhere a `widget` element is used, you can substitute a `woodWidget` element. This is shown in Example 118 on page 218.

Example 118. Using a Substitution Group

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
         substitutionGroup="widget" />
```

Type restrictions

The elements of a substitution group must be of the same type as the head element or of a type derived from the head element's type. For example, if the head element is of type xsd:int all members of the substitution group must be of type xsd:int or of a type derived from xsd:int. You can also define a substitution group similar to the one shown in Example 119 on page 219

where the elements of the substitution group are of types derived from the head element's type.

Example 119. Substitution Group with Complex Types

```
<complexType name="widgetType">
  <sequence>
    <element name="shape" type="xsd:string" />
    <element name="color" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="woodWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="woodType" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="plasticWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="moldProcess" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="widget" type="widgetType" />
<element name="woodWidget" type="woodWidgetType"
         substitutionGroup="widget" />
<element name="plasticWidget" type="plasticWidgetType"
         substitutionGroup="widget" />
<complexType name="partType">
  <sequence>
    <element ref="widget" />
  </sequence>
</complexType>
<element name="part" type="partType" />
```

The head element of the substitution group, `widget`, is defined as being of type widgetType. Each element of the substitution group extends widgetType to include data that is specific to ordering that type of widget.

Based on the schema in Example 119 on page 219, the `part` elements in Example 120 on page 220 are valid.

Example 120. XML Document using a Substitution Group

```
<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
    <color>blue</color>
    <moldProcess>sandCast</moldProcess>
  </plasticWidget>
</part>
<part>
  <woodWidget>
    <shape>round</shape>
    <color>blue</color>
    <woodType>elm</woodType>
  </woodWidget>
</part>
```

Abstract head elements

You can define an abstract head element that can never appear in a document produced using your schema. Abstract head elements are similar to abstract classes in Java because they are used as the basis for defining more specific implementations of a generic class. Abstract heads also prevent the use of the generic element in the final product.

You declare an abstract head element by setting the `abstract` attribute of an `element` element to `true`, as shown in Example 121 on page 220. Using this schema, a valid `review` element can contain either a `positiveComment` element or a `negativeComment` element, but cannot contain a `comment` element.

Example 121. Abstract Head Definition

```
<element name="comment" type="xsd:string" abstract="true" />
<element name="positiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="negtiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="review">
  <complexContent>
    <all>
```

```
      <element name="custName" type="xsd:string" />
      <element name="impression" ref="comment" />
    </all>
  </complexContent>
</element>
```

Substitution Groups in Java

Overview

Artix ESB, as specified in the JAXB specification, supports substitution groups using Java's native class hierarchy in combination with the ability of the `JAXBElement` class' support for wildcard definitions. Because the members of a substitution group must all share a common base type, the classes generated to support the elements' types also share a common base type. In addition, Artix ESB maps instances of the head element to `JAXBElement<? extends T>` properties.

Generated object factory methods

The object factory generated to support a package containing a substitution group has methods for each of the elements in the substitution group. For each of the members of the substitution group, except for the head element, the `@XmlElementDecl` annotation decorating the object factory method includes two additional properties, as described in Table 21 on page 222.

Table 21. Properties for Declaring a JAXB Element is a Member of a Substitution Group

Property	Description
substitutionHeadNamespace	Specifies the namespace where the head element is defined.
substitutionHeadName	Specifies the value of the head element's `name` attribute.

The object factory method for the head element of the substitution group's `@XmlElementDecl` contains only the default namespace property and the default name property.

In addition to the element instantiation methods, the object factory contains a method for instantiating an object representing the head element. If the members of the substitution group are all of complex types, the object factory also contains methods for instantiating instances of each complex type used.

Example 122 on page 222 shows the object factory method for the substitution group defined in Example 119 on page 219.

Example 122. Object Factory Method for a Substitution Group

```
public class ObjectFactory {

    private final static QName _Widget_QNAME = new QName(...);
    private final static QName _PlasticWidget_QNAME = new QName(...);
    private final static QName _WoodWidget_QNAME = new QName(...);
```

```
    public ObjectFactory() {
    }

    public WidgetType createWidgetType() {
        return new WidgetType();
    }

    public PlasticWidgetType createPlasticWidgetType() {
        return new PlasticWidgetType();
    }

    public WoodWidgetType createWoodWidgetType() {
        return new WoodWidgetType();
    }

    @XmlElementDecl(namespace="...", name = "widget")
    public JAXBElement<WidgetType> createWidget(WidgetType value) {
        return new JAXBElement<WidgetType>(_Widget_QNAME, WidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "plasticWidget", substitutionHeadNamespace =
 "...", substitutionHeadName = "widget")
    public JAXBElement<PlasticWidgetType> createPlasticWidget(PlasticWidgetType value) {
        return new JAXBElement<PlasticWidgetType>(_PlasticWidget_QNAME, PlasticWidget
Type.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "woodWidget", substitutionHeadNamespace =
"...", substitutionHeadName = "widget")
    public JAXBElement<WoodWidgetType> createWoodWidget(WoodWidgetType value) {
        return new JAXBElement<WoodWidgetType>(_WoodWidget_QNAME, WoodWidgetType.class,
null, value);
    }

}
```

Substitution groups in interfaces

If the head element of a substitution group is used as a message part in one of an operation's messages, the resulting method parameter will be an object of the class generated to support that element. It will not necessarily be an instance of the `JAXBElement<? extends T>` class. The runtime relies on Java's native type hierarchy to support the type substitution, and Java will catch any attempts to use unsupported types.

To ensure that the runtime knows all of the classes needed to support the element substitution, the SEI is decorated with the `@XmlSeeAlso` annotation.

This annotation specifies a list of classes required by the runtime for marshalling. Fore more information on using the `@XmlSeeAlso` annotation see Adding Classes to the Runtime Marshaller on page 144.

Example 124 on page 224 shows the SEI generated for the interface shown in Example 123 on page 224. The interface uses the substitution group defined in Example 119 on page 219.

Example 123. WSDL Interface Using a Substitution Group

```
<message name="widgetMessage">
    <part name="widgetPart" element="xsd1:widget" />
  </message>
  <message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int" />
  </message>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order" />
      <output message="tns:widgetOrderBill" name="bill" />
      <fault message="tns:badSize" name="sizeFault" />
    </operation>
    <operation name="checkWidgets">
      <input message="tns:widgetMessage" name="request" />
      <output message="tns:numWidgets" name="response" />
    </operation>
  </portType>
```

Example 124. Generated Interface Using a Substitution Group

```
@WebService(targetNamespace = "...", name = "orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace = "...")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );
}
```

Tip

The SEI shown in Example 124 on page 224 lists the object factory in the `@XmlSeeAlso` annotation. Listing the object factory for a namespace provides access to all of the generated classes for that namespace.

Substitution groups in complex types

When the head element of a substitution group is used as an element in a complex type, the code generator maps the element to a `JAXBElement<? extends T>` property. It does not map it to a property containing an instance of the generated class generated to support the substitution group.

For example, the complex type defined in Example 125 on page 225 results in the Java class shown in Example 126 on page 225. The complex type uses the substitution group defined in Example 119 on page 219.

Example 125. Complex Type Using a Substitution Group

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
  </sequence>
</complexType>
```

Example 126. Java Class for a Complex Type Using a Substitution Group

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "widgetOrderInfo", propOrder =
{"amount","widget",})
public class WidgetOrderInfo {

    protected int amount;
    @XmlElementRef(name = "widget", namespace = "...", type
= JAXBElement.class)
    protected JAXBElement<? extends WidgetType> widget;
    public int getAmount() {
        return amount;
    }

    public void setAmount(int value) {
        this.amount = value;
    }

    public JAXBElement<? extends WidgetType> getWidget() {
```

```
        return widget;
    }

    public void setWidget(JAXBElement<? extends WidgetType>
value) {
        this.widget = ((JAXBElement<? extends WidgetType> )
value);
    }

}
```

Setting a substitution group property

How you work with a substitution group depends on whether the code generator mapped the group to a straight Java class or to a `JAXBElement<? extends T>` class. When the element is simply mapped to an object of the generated value class, you work with the object the same way you work with other Java objects that are part of a type hierarchy. You can substitute any of the subclasses for the parent class. You can inspect the object to determine its exact class, and cast it appropriately.

Tip

The JAXB specification recommends that you use the object factory methods for instantiating objects of the generated classes.

When the code generators create a `JAXBElement<? extends T>` object to hold instances of a substitution group, you must wrap the element's value in a `JAXBElement<? extends T>` object. The best method to do this is to use the element creation methods provided by the object factory. They provide an easy means for creating an element based on its value.

Example 127 on page 226 shows code for setting an instance of a substitution group.

Example 127. Setting a Member of a Substitution Group

```
ObjectFactory of = new ObjectFactory(); ❶
PlasticWidgetType pWidget = of.createPlasticWidgetType(); ❷
pWidget.setShape = "round';
pWidget.setColor = "green";
pWidget.setMoldProcess = "injection";

JAXBElement<PlasticWidgetType> widget = of.createPlasticWidget(pWidget); ❸
```

```
WidgetOrderInfo order = of.createWidgetOrderInfo(); ❹
order.setWidget(widget); ❺
```

The code in Example 127 on page 226 does the following:

❶ Instantiates an object factory.

❷ Instantiates a `PlasticWidgetType` object.

❸ Instantiates a `JAXBElement<PlasticWidgetType>` object to hold a plastic widget element.

❹ Instantiates a `WidgetOrderInfo` object.

❺ Sets the `WidgetOrderInfo` object's widget to the `JAXBElement` object holding the plastic widget element.

Getting the value of a substitution group property

The object factory methods do not help when extracting the element's value from a `JAXBElement<? extends T>` object. You must to use the `JAXBElement<? extends T>` object's `getValue()` method. The following options determine the type of object returned by the `getValue()` method:

- Use the `isInstance()` method of all the possible classes to determine the class of the element's value object.
- Use the `JAXBElement<? extends T>` object's `getName()` method to determine the element's name.

  The `getName()` method returns a QName. Using the local name of the element, you can determine the proper class for the value object.
- Use the `JAXBElement<? extends T>` object's `getDeclaredType()` method to determine the class of the value object.

  The `getDeclaredType()` method returns the `Class` object of the element's value object.

Warning

There is a possibility that the `getDeclaredType()` method will return the base class for the head element regardless of the actual class of the value object.

Example 128 on page 228 shows code retrieving the value from a substitution group. To determine the proper class of the element's value object the example uses the element's `getName()` method.

Example 128. Getting the Value of a Member of the Substitution Group

```
String elementName = order.getWidget().getName().getLocalPart();
if (elementName.equals("woodWidget")
{
  WoodWidgetType widget=order.getWidget().getValue();
}
else if (elementName.equals("plasticWidget")
{
  PlasticWidgetType widget=order.getWidget().getValue();
}
else
{
  WidgetType widget=order.getWidget().getValue();
}
```

Widget Vendor Example

This section shows an example of substitution groups being used in Artix ESB to solve a real world application. A service and consumer are developed using the widget substitution group defined in Example 119 on page 219. The service offers two operations: `checkWidgets` and `placeWidgetOrder`.

Example 129 on page 229 shows the interface for the ordering service.

Example 129. Widget Ordering Interface

```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
</message>
<message name="widgetMessage">
  <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

Example 130 on page 229 shows the generated Java SEI for the interface.

Example 130. Widget Ordering SEI

```
@WebService(targetNamespace = "http://widgetVendor.com/widgetOrderForm", name = "orderWid
gets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
```

```
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace = "http://wid
getVendor.com/types/widgetTypes")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "widgetOrderConformation", targetNamespace = "", partName = "widget
OrderConformation")
    @WebMethod
    public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo placeWidgetOrder(
        @WebParam(partName = "widgetOrderForm", name = "widgetOrderForm", targetNamespace
= "")
        com.widgetvendor.types.widgettypes.WidgetOrderInfo widgetOrderForm
    ) throws BadSize;
}
```

Note

Because the example only demonstrates the use of substitution groups, some of the business logic is not shown.

The checkWidgets Operation

Overview

`checkWidgets` is a simple operation that has a parameter that is the head member of a substitution group. This operation demonstrates how to deal with individual parameters that are members of a substitution group. The consumer must ensure that the parameter is a valid member of the substitution group. The service must properly determine which member of the substitution group was sent in the request.

Consumer implementation

The generated method signature uses the Java class supporting the type of the substitution group's head element. Because the member elements of a substitution group are either of the same type as the head element or of a type derived from the head element's type, the Java classes generated to support the members of the substitution group inherit from the Java class generated to support the head element. Java's type hierarchy natively supports using subclasses in place of the parent class.

Because of how Artix ESB generates the types for a substitution group and Java's type hierarchy, the client can invoke `checkWidgets()` without using any special code. When developing the logic to invoke `checkWidgets()` you can pass in an object of one of the classes generated to support the widget substitution group.

Example 131 on page 231 shows a consumer invoking `checkWidgets()`.

Example 131. Consumer Invoking `checkWidgets()`

```
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
  case '1':
  {
    WidgetType widget = new WidgetType();
    ...
    break;
  }
```

```
  case '2':
  {
    WoodWidgetType widget = new WoodWidgetType();
    ...
    break;
  }
  case '3':
  {
    PlasticWidgetType widget = new PlasticWidgetType();
    ...
    break;
  }
  default :
    System.out.println("Invaid Widget Selection!!");
}

proxy.checkWidgets(widgets);
```

Service implementation

The service's implementation of `checkWidgets()` gets a widget description as a `WidgetType` object, checks the inventory of widgets, and returns the number of widgets in stock. Because all of the classes used to implement the substitution group inherit from the same base class, you can implement `checkWidgets()` without using any JAXB specific APIs.

All of the classes generated to support the members of the substitution group for `widget` extend the `WidgetType` class. Because of this fact, you can use `instanceof` to determine what type of widget was passed in and simply cast the *`widgetPart`* object into the more restrictive type if appropriate. Once you have the proper type of object, you can check the inventory of the right kind of widget.

Example 132 on page 232 shows a possible implementation.

Example 132. Service Implementation of `checkWidgets()`

```
public int checkWidgets(WidgetType widgetPart)
{
  if (widgetPart instanceof WidgetType)
  {
    return checkWidgetInventory(widgetType);
  }
  else if (widgetPart instanceof WoodWidgetType)
  {
    WoodWidgetType widget = (WoodWidgetType)widgetPart;
```

```
    return checkWoodWidgetInventory(widget);
  }
  else if (widgetPart instanceof PlasticWidgetType)
  {
    PlasticWidgetType widget = (PlasticWidgetType)widgetPart;
    return checkPlasticWidgetInventory(widget);
  }
}
```

The placeWidgetOrder Operation

Overview

`placeWidgetOrder` uses two complex types containing the substitution group. This operation demonstrates to use such a structure in a Java implementation. Both the consumer and the service must get and set members of a substitution group.

Consumer implementation

To invoke `placeWidgetOrder()` the consumer must construct a widget order containing one element of the widget substitution group. When adding the widget to the order, the consumer should use the object factory methods generated for each element of the substitution group. This ensures that the runtime and the service can correctly process the order. For example, if an order is being placed for a plastic widget, the `ObjectFactory.createPlasticWidget()` method is used to create the element before adding it to the order.

Example 133 on page 234 shows consumer code for setting the widget property of the `WidgetOrderInfo` object.

Example 133. Setting a Substitution Group Member

```
ObjectFactory of = new ObjectFactory();

WidgetOrderInfo order = new of.createWidgetOrderInfo();
...
System.out.println();
System.out.println("What color widgets do you want to order?");
String color = reader.readLine();
System.out.println();
System.out.println("What shape widgets do you want to order?");
String shape = reader.readLine();
System.out.println();
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
  case '1':
  {
```

```
    WidgetType widget = of.createWidgetType();
    widget.setColor(color);
    widget.setShape(shape);
    JAXB<WidgetType> widgetElement = of.createWidget(widget);
    order.setWidget(widgetElement);
    break;
  }
  case '2':
  {
    WoodWidgetType woodWidget = of.createWoodWidgetType();
    woodWidget.setColor(color);
    woodWidget.setShape(shape);
    System.out.println();
    System.out.println("What type of wood are your widgets?");
    String wood = reader.readLine();
    woodWidget.setWoodType(wood);
    JAXB<WoodWidgetType> widgetElement = of.createWoodWidget(woodWidget);
    order.setWoodWidget(widgetElement);
    break;
  }
  case '3':
  {
    PlasticWidgetType plasticWidget = of.createPlasticWidgetType();
    plasticWidget.setColor(color);
    plasticWidget.setShape(shape);
    System.out.println();
    System.out.println("What type of mold to use for your
                        widgets?");
    String mold = reader.readLine();
    plasticWidget.setMoldProcess(mold);
    JAXB<WidgetType> widgetElement = of.createPlasticWidget(plasticWidget);
    order.setPlasticWidget(widgetElement);
    break;
  }
  default :
    System.out.println("Invaid Widget Selection!!");
    }
```

Service implementation

The `placeWidgetOrder()` method receives an order in the form of a `WidgetOrderInfo` object, processes the order, and returns a bill to the consumer in the form of a `WidgetOrderBillInfo` object. The orders can be for a plain widget, a plastic widget, or a wooden widget. The type of widget ordered is determined by what type of object is stored in *`widgetOrderForm`* object's widget property. The widget property is a substitution group and can

contain a `widget` element, a `woodWidget` element, or a `plasticWidget` element.

The implementation must determine which of the possible elements is stored in the order. This can be accomplished using the `JAXBElement<? extends T>` object's `getName()` method to determine the element's QName. The QName can then be used to determine which element in the substitution group is in the order. Once the element included in the bill is known, you can extract its value into the proper type of object.

Example 134 on page 236 shows a possible implementation.

Example 134. Implementation of `placeWidgetOrder()`

```
public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo placeWidgetOrder(WidgetOrderInfo
 widgetOrderForm)
{
  ObjectFactory of = new ObjectFactory(); ❶

  WidgetOrderBillInfo bill = new WidgetOrderBillInfo() ❷

   // Copy the shipping address and the number of widgets
   // ordered from widgetOrderForm to bill
   ...

  int numOrdered = widgetOrderForm.getAmount(); ❸

  String elementName = widgetOrderForm.getWidget().getName().getLocalPart(); ❹
  if (elementName.equals("woodWidget") ❺
  {
    WoodWidgetType widget=order.getWidget().getValue(); ❻
    buildWoodWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<WoodWidgetType> widgetElement = of.createWoodWidget(widget); ❼
    bill.setWidget(widgetElement); ❽

    float amtDue = numOrdered * 0.75;
    bill.setAmountDue(amtDue); ❾
  }
  else if (elementName.equals("plasticWidget")
  {
    PlasticWidgetType widget=order.getWidget().getValue();
    buildPlasticWidget(widget, numOrdered);

    // Add the widget info to bill
```

```
    JAXBElement<PlasticWidgetType> widgetElement = of.createPlasticWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.90;
    bill.setAmountDue(amtDue);
  }
  else
  {
    WidgetType widget=order.getWidget().getValue();
    buildWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<WidgetType> widgetElement = of.createWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.30;
    bill.setAmountDue(amtDue);
  }

  return(bill);
}
```

The code in Example 134 on page 236 does the following:

❶ Instantiates an object factory to create elements.

❷ Instantiates a `WidgetOrderBillInfo` object to hold the bill.

❸ Gets the number of widgets ordered.

❹ Gets the local name of the element stored in the order.

❺ Checks to see if the element is a `woodWidget` element.

❻ Extracts the value of the element from the order to the proper type of object.

❼ Creates a `JAXBElement<T>` object placed into the bill.

❽ Sets the bill object's widget property.

❾ Sets the bill object's amountDue property.

Customizing How Types are Generated

The JAXB default mappings cover most uses of XML Schema used when using service-oriented design to create Java applications. For instances where the default mappings are insufficient, JAXB provides an extensive customization mechanism.

Important

JAXB customizations are ignored if you are using the **wsdlgen** tool.

Basics of Customizing Type Mappings

Overview

The JAXB specification defines a number of XML elements that customize how Java types are mapped to XML Schema constructs. These elements can be specified in-line with XML Schema constructs. If you cannot, or do not want to, modify the XML Schema definitions, you can specify the customizations in external binding document.

Namespace

The elements used to customize the JAXB data bindings are defined in the namespace http://java.sun.com/xml/ns/jaxb. You must add a namespace declaration similar to the one shown in Example 135 on page 240. This is added to the root element of all XML documents defining JAXB customizations.

Example 135. JAXB Customization Namespace

```
xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
```

Version declaration

When using the JAXB customizations, you must indicate the JAXB version being used. This is done by adding a `jaxb:version` attribute to the root element of the external binding declaration. If you are using in-line customization, you must include the `jaxb:version` attribute in the `schema` element containing the customizations. The value of the attribute is always `2.0`.

Example 136 on page 240 shows an example of the `jaxb:version` attribute used in a `schema` element.

Example 136. Specifying the JAXB Customization Version

```
< schema ...
        jaxb:version="2.0">
```

Using in-line customization

The most direct way to customize how the code generators map XML Schema constructs to Java constructs is to add the customization elements directly to the XML Schema definitions. The JAXB customization elements are placed inside the `xsd:appinfo` element of the XML schema construct that is being modified.

Example 137 on page 241 shows an example of a schema containing an in-line JAXB customization.

Example 137. Customized XML Schema

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="size">
    <annotation>
      <appinfo>
        <jaxb:class name="widgetSize" />
      </appinfo>
    </annotation>
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

Using an external binding declaration

When you cannot, or do not want to, make changes to the XML Schema document that defines your type, you can specify the customizations using an external binding declaration. An external binding declaration consists of a number of nested `jaxb:bindings` elements. Example 138 on page 241 shows the syntax of an external binding declaration.

Example 138. JAXB External Binding Declaration Syntax

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings [schemaLocation="schemaUri" | wsdlLocation="wsdlUri">
    <jaxb:bindings node="nodeXPath">
      binding declaration
    </jaxb:bindings>
    ...
  </jaxb:bindings>
<jaxb:bindings>
```

The `schemaLocation` attribute and the `wsdlLocation` attribute are used to identify the schema document to which the modifications are applied. Use the `schemaLocation` attribute if you are generating code from a schema

document. Use the `wsdlLocation` attribute if you are generating code from a WSDL document.

The `node` attribute is used to identify the specific XML schema construct that is to be modified. It is an XPath statement that resolves to an XML Schema element.

Given the schema document `widgetSchema.xsd`, shown in Example 139 on page 242, the external binding declaration shown in Example 140 on page 242 modifies the generation of the complex type size.

Example 139. XML Schema File

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        version="1.0">
  <complexType name="size">
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

Example 140. External Binding Declaration

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="wsdlSchema.xsd">
    <jaxb:bindings node="xsd:complexType[@name='size']">
        <jaxb:class name="widgetSize" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

To instruct the code generators to use the external binging declaration use the artix wsdl2java tool's `-b` *`binding-file`* option, as shown below:

```
artix wsdl2java -b widgetBinding.xml widget.wsdl
```

Specifying the Java Class of an XML Schema Primitive

Overview

By default, XML Schema types are mapped to Java primitive types. While this is the most logical mapping between XML Schema and Java, it does not always meet the requirements of the application developer. You might want to map an XML Schema primitive type to a Java class that can hold extra information, or you might want to map an XML primitive type to a class that allows for simple type substitution.

The JAXB `javaType` customization element allows you to customize the mapping between an XML Schema primitive type and a Java primitive type. It can be used to customize the mappings at both the global level and the individual instance level. You can use the `javaType` element as part of a simple type definition or as part of a complex type definition.

When using the `javaType` customization element you must specify methods for converting the XML representation of the primitive type to and from the target Java class. Some mappings have default conversion methods. For instances where there are no default mappings, Artix ESB provides JAXB methods to ease the development of the required methods.

Syntax

The `javaType` customization element takes four attributes, as described in Table 22 on page 243.

Table 22. Attributes for Customizing the Generation of a Java Class for an XML Schema Type

Attribute	Required	Description
`name`	Yes	Specifies the name of the Java class to which the XML Schema primitive type is mapped. It must be either a valid Java class name or the name of a Java primitive type. You must ensure that this class exists and is accessible to your application. The code generator does not check for this class.
`xmlType`	No	Specifies the XML Schema primitive type that is being customized. This attribute is only used when the `javaType` element is used as a child of the `globalBindings` element.
`parseMethod`	No	Specifies the method responsible for parsing the string-based XML representation of the data into an instance of the Java class. For more information see Specifying the converters on page 246.
`printMethod`	No	Specifies the method responsible for converting a Java object to the string-based XML representation of the data. For more information see Specifying the converters on page 246.

The `javaType` customization element can be used in three ways:

- To modify all instances of an XML Schema primitive type — The `javaType` element modifies all instances of an XML Schema type in the schema document when it is used as a child of the `globalBindings` customization element. When it is used in this manner, you must specify a value for the `xmlType` attribute that identifies the XML Schema primitive type being modified.

  Example 141 on page 244 shows an in-line global customization that instructs the code generators to use `java.lang.Integer` for all instances of xsd:short in the schema.

Example 141. Global Primitive Type Customization

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings ...>
        <jaxb:javaType name="java.lang.Integer"
                       xmlType="xsd:short" />
      </globalBindings
    </appinfo>
  </annotation>
  ...
</schema>
```

- To modify a simple type definition — The `javaType` element modifies the class generated for all instances of an XML simple type when it is applied to a named simple type definition. When using the `javaType` element to modify a simple type definition, do not use the `xmlType` attribute.

  Example 142 on page 244 shows an external binding file that modifies the generation of a simple type named zipCode.

Example 142. Binding File for Customizing a Simple Type

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
```

```
  <jaxb:bindings wsdlLocation="widgets.wsdl">
    <jaxb:bindings node="xsd:simpleType[@name='zipCode']">
        <jaxb:javaType name="com.widgetVendor.widgetTypes.zipCodeType"
                       parseMethod="com.widgetVendor.widgetTypes.support.parseZipCode"
                       printMethod="com.widgetVendor.widgetTypes.support.printZipCode" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

- To modify an element or attribute of a complex type definition — The `javaType` can be applied to individual parts of a complex type definition by including it as part of a JAXB property customization. The `javaType` element is placed as a child to the property's `baseType` element. When using the `javaType` element to modify a specific part of a complex type definition, do not use the `xmlType` attribute.

  Example 143 on page 245 shows a binding file that modifies an element of a complex type.

Example 143. Binding File for Customizing an Element in a Complex Type

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='cost']">
        <jaxb:property>
          <jaxb:baseType>
            <jaxb:javaType name="com.widgetVendor.widgetTypes.costType"
                           parseMethod="parseCost"
                           printMethod="printCost" >
          </jaxb:baseType>
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

For more information on using the `baseType` element see Specifying the Base Type of an Element or an Attribute on page 262.

Specifying the converters

The Artix ESB cannot convert XML Schema primitive types into random Java classes. When you use the `javaType` element to customize the mapping of an XML Schema primitive type, the code generator creates an adapter class that is used to marshal and unmarshal the customized XML Schema primitive type. A sample adapter class is shown in Example 144 on page 246.

Example 144. JAXB Adapter Class

```
public class Adapter1 extends XmlAdapter<String, javaType>
{
  public javaType unmarshal(String value)
  {
    return(parseMethod(value));
  }

  public String marshal(javaType value)
  {
    return(printMethod(value));
  }
}
```

`parseMethod` and *`printMethod`* are replaced by the value of the corresponding `parseMethod` attribute and `printMethod` attribute. The values must identify valid Java methods. You can specify the method's name in one of two ways:

- A fully qualified Java method name in the form of *`packagename.ClassName.methodName`*
- A simple method name in the form of *`methodName`*

  When you only provide a simple method name, the code generator assumes that the method exists in the class specified by the `javaType` element's `name` attribute.

Important

The code generators **do not** generate parse or print methods. You are responsible for supplying them. For information on developing parse and print methods see Implementing converters on page 249.

If a value for the `parseMethod` attribute is not provided, the code generator assumes that the Java class specified by the `name` attribute has a constructor whose first parameter is a Java `String` object. The generated adapter's `unmarshal()` method uses the assumed constructor to populate the Java object with the XML data.

If a value for the `printMethod` attribute is not provided, the code generator assumes that the Java class specified by the `name` attribute has a `toString()` method. The generated adapter's `marshal()` method uses the assumed `toString()` method to convert the Java object to XML data.

If the `javaType` element's `name` attribute specifies a Java primitive type, or one of the Java primitive's wrapper types, the code generators use the default converters. For more information on default converters see Default primitive type converters on page 250.

What is generated

As mentioned in Specifying the converters on page 246, using the `javaType` customization element triggers the generation of one adapter class for each customization of an XML Schema primitive type. The adapters are named in sequence using the pattern `Adapter`*`N`*. If you specify two primitive type customizations, the code generators create two adapter classes: `Adapter1` and `Adapter2`.

The code generated for an XML schema construct depends on whether the effected XML Schema construct is a globally defined element or is defined as part of a complex type.

When the XML Schema construct is a globally defined element, the object factory method generated for the type is modified from the default method as follows:

- The method is decorated with an `@XmlJavaTypeAdapter` annotation.

The annotation instructs the runtime which adapter class to use when processing instances of this element. The adapter class is specified as a class object.

- The default type is replaced by the class specified by the `javaType` element's `name` attribute.

Example 145 on page 248 shows the object factory method for an element affected by the customization shown in Example 141 on page 244.

Example 145. Customized Object Factory Method for a Global Element

```
@XmlElementDecl(namespace = "http://widgetVendor.com/types/wid
getTypes", name = "shorty")
   @XmlJavaTypeAdapter(org.w3._2001.xmlschema.Adapter1 .class)

    public JAXBElement<Integer> createShorty(Integer value)
{
        return new JAXBElement<Integer>(_Shorty_QNAME, In
teger.class, null, value);
    }
```

When the XML Schema construct is defined as part of a complex type, the generated Java property is modified as follows:

- The property is decorated with an `@XmlJavaTypeAdapter` annotation.

  The annotation instructs the runtime which adapter class to use when processing instances of this element. The adapter class is specified as a class object.

- The property's `@XmlElement` includes a type property.

  The value of the type property is the class object representing the generated object's default base type. In the case of XML Schema primitive types, the class is `String`.

- The property is decorated with an `@XmlSchemaType` annotation.

  The annotation identifies the XML Schema primitive type of the construct.

- The default type is replaced by the class specified by the `javaType` element's `name` attribute.

Example 146 on page 249 shows the object factory method for an element affected by the customization shown in Example 141 on page 244.

Example 146. Customized Complex Type

```
public class NumInventory {

    @XmlElement(required = true, type = String.class)
    @XmlJavaTypeAdapter(Adapter1 .class)
    @XmlSchemaType(name = "short")
    protected Integer numLeft;
    @XmlElement(required = true)
    protected String size;

    public Integer getNumLeft() {
        return numLeft;
    }

    public void setNumLeft(Integer value) {
        this.numLeft = value;
    }

    public String getSize() {
        return size;
    }

    public void setSize(String value) {
        this.size = value;
    }

}
```

Implementing converters

The Artix ESB runtime has does not know how to convert XML primitive types to and from the Java class specified by the `javaType` element, except that it should call the methods specified by the `parseMethod` attribute and the `printMethod` attribute. You are responsible for providing implementations of the methods the runtime calls. The implemented methods must be capable of working with the lexical structures of the XML primitive type.

To simplify the implementation of the data conversion methods, Artix ESB provides the `javax.xml.bind.DatatypeConverter` class. This class provides methods for parsing and printing all of the XML Schema primitive types. The parse methods take string representations of the XML data and they return an instance of the default type defined in Table 15 on page 154. The print

methods take an instance of the default type and they return a string representation of the XML data.

The Java documentation for the `DatatypeConverter` class can be found at http://java.sun.com/webservices/docs/1.6/api/javax/xml/bind/DatatypeConverter.html.

Default primitive type converters

When specifying a Java primitive type, or one of the Java primitive type Wrapper classes, in the `javaType` element's `name` attribute, it is not necessary to specify values for the `parseMethod` attribute or the `printMethod` attribute. The Artix ESB runtime substitutes default converters if no values are provided.

The default data converters use the JAXB `DatatypeConverter` class to parse the XML data. The default converters will also provide any type casting necessary to make the conversion work.

Generating Java Classes for Simple Types

Overview

By default, named simple types do not result in generated types unless they are enumerations. Elements defined using a simple type are mapped to properties of a Java primitive type.

There are instances when you need to have simple types generated into Java classes, such as is when you want to use type substitution.

To instruct the code generators to generate classes for all globally defined simple types, set the `globalBindings` customization element's `mapSimpleTypeDef` to `true`.

Adding the customization

To instruct the code generators to create Java classes for named simple types add the `globalBinding` element's `mapSimpleTypeDef` attribute and set its value to `true`.

Example 147 on page 251 shows an in-line customization that forces the code generator to generate Java classes for named simple types.

Example 147. in-Line Customization to Force Generation of Java Classes for SimpleTypes

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings mapSimpleTypeDef="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

Example 148 on page 251 shows an external binding file that customizes the generation of simple types.

Example 148. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
```

```
    <jaxb:globalBindings mapSimpleTypeDef="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

Important

This customization only affects *named* simple types that are defined in the *global* scope.

Generated classes

The class generated for a simple type has one property called value. The value property is of the Java type defined by the mappings in Primitive Types on page 154. The generated class has a getter and a setter for the value property.

Example 150 on page 252 shows the Java class generated for the simple type defined in Example 149 on page 252.

Example 149. Simple Type for Customized Mapping

```
<simpleType name="simpleton">
  <restriction base="xsd:string">
    <maxLength value="10"/>
  </restriction>
</simpleType>
```

Example 150. Customized Mapping of a Simple Type

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "simpleton", propOrder = {"value"})
public class Simpleton {

    @XmlValue
    protected String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

}
```

Customizing Enumeration Mapping

Overview

If you want enumerated types that are based on a schema type other than xsd:string, you must instruct the code generator to map it. You can also control the name of the generated enumeration constants.

The customization is done using the `jaxb:typesafeEnumClass` element along with one or more `jaxb:typesafeEnumMember` elements.

There might also be instances where the default settings for the code generator cannot create valid Java identifiers for all of the members of an enumeration. You can customize how the code generators handle this by using an attribute of the `globalBindings` customization.

Member name customizer

If the code generator encounters a naming collision when generating the members of an enumeration or if it cannot create a valid Java identifier for a member of the enumeration, the code generator, by default, generates a warning and does not generate a Java enum type for the enumeration.

You can alter this behavior by adding the `globalBinding` element's `typesafeEnumMemberName` attribute. The `typesafeEnumMemberName` attribute's values are described in Table 23 on page 253.

Table 23. Values for Customizing Enumeration Member Name Generation

Value	Description
`skipGeneration`(default)	Specifies that the Java enum type is not generated and generates a warning.
`generateName`	Specifies that member names will be generated following the pattern `VALUE_`*`N`*. *N* starts off at one, and is incremented for each member of the enumeration.
`generateError`	Specifies that the code generator generates an error when it cannot map an enumeration to a Java enum type.

Example 151 on page 254 shows an in-line customization that forces the code generator to generate type safe member names.

Example 151. Customization to Force Type Safe Member Names

```
<schema targetNamespace="http://widget.com/types/widgetTypes"

        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings typesafeEnumMemberName="generate
Name" />
    </appinfo>
  </annotation>
  ...
</schema>
```

Class customizer

The `jaxb:typesafeEnumClass` element specifies that an XML Schema enumeration should be mapped to a Java enum type. It has two attributes that are described in Table 24 on page 254. When the `jaxb:typesafeEnumClass` element is specified in-line, it must be placed inside the `xsd:annotation` element of the simple type it is modifying.

Table 24. Attributes for Customizing a Generated Enumeration Class

Attribute	Description
`name`	Specifies the name of the generated Java enum type. This value must be a valid Java identifier.
`map`	Specifies if the enumeration should be mapped to a Java enum type. The default value is `true`.

Member customizer

The `jaxb:typesafeEnumMember` element specifies the mapping between an XML Schema `enumeration` facet and a Java enum type constant. You must use one `jaxb:typesafeEnumMember` element for each `enumeration` facet in the enumeration being customized.

When using in-line customization, this element can be used in one of two ways:

- It can be placed inside the `xsd:annotation` element of the `enumeration` facet it is modifying.
- They can all be placed as children of the `jaxb:typesafeEnumClass` element used to customize the enumeration.

The `jaxb:typesafeEnumMember` element has a `name` attribute that is required. The `name` attribute specifies the name of the generated Java enum type constant. It's value must be a valid Java identifier.

The `jaxb:typesafeEnumMember` element also has a `value` attribute. The `value` is used to associate the `enumeration` facet with the proper `jaxb:typesafeEnumMember` element. The value of the `value` attribute must match one of the values of an `enumeration` facets' `value` attribute. This attribute is required when you use an external binding specification for customizing the type generation, or when you group the `jaxb:typesafeEnumMember` elements as children of the `jaxb:typesafeEnumClass` element.

Examples

Example 152 on page 255 shows an enumerated type that uses in-line customization and has the enumeration's members customized separately.

Example 152. In-line Customization of an Enumerated Type

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass />
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="one" />
          </appinfo>
        </annotation>
      </enumeration>
```

```
      <enumeration value="2">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="two" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="3">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="three" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="4">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="four" />
          </appinfo>
        </annotation>
      </enumeration>
    </restriction>
  </simpleType>
<schema>
```

Example 153 on page 256 shows an enumerated type that uses in-line customization and combines the member's customization in the class customization.

Example 153. In-line Customization of an Enumerated Type Using a Combined Mapping

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass>
            <jaxb:typesafeEnumMember value="1" name="one" />
            <jaxb:typesafeEnumMember value="2" name="two" />
            <jaxb:typesafeEnumMember value="3" name="three" />
            <jaxb:typesafeEnumMember value="4" name="four" />
        </jaxb:typesafeEnumClass>
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1" />
```

```
      <enumeration value="2" />
      <enumeration value="3" />
      <enumeration value="4" >
    </restriction>
  </simpleType>
<schema>
```

Example 154 on page 257 shows an external binding file that customizes an enumerated type.

Example 154. Binding File for Customizing an Enumeration

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:simpleType[@name='widgetInteger']">
        <jaxb:typesafeEnumClass>
            <jaxb:typesafeEnumMember value="1" name="one" />
            <jaxb:typesafeEnumMember value="2" name="two" />
            <jaxb:typesafeEnumMember value="3" name="three" />
            <jaxb:typesafeEnumMember value="4" name="four" />
        </jaxb:typesafeEnumClass>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

Customizing Fixed Value Attribute Mapping

Overview

By default, the code generators map attributes defined as having a fixed value to normal properties. When using schema validation, Artix ESB can enforce the schema definition. However, using schema validation increases message processing time.

Another way to map attributes that have fixed values to Java is to map them to Java constants. You can instruct the code generator to map fixed value attributes to Java constants using the `globalBindings` customization element. You can also customize the mapping of fixed value attributes to Java constants at a more localized level using the `property` element.

Global customization

You can alter this behavior by adding the `globalBinding` element's `fixedAttributeAsConstantProperty` attribute. Setting this attribute to `true` instructs the code generator to map any attribute defined using `fixed` attribute to a Java constant.

Example 155 on page 258 shows an in-line customization that forces the code generator to generate constants for attributes with fixed values.

Example 155. in-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"

        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings fixedAttributeAsConstantProp
erty="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

Example 156 on page 259 shows an external binding file that customizes the generation of fixed attributes.

Example 156. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

Local mapping

You can customize attribute mapping on a per-attribute basis using the `property` element's `fixedAttributeAsConstantProperty` attribute. Setting this attribute to `true` instructs the code generator to map any attribute defined using `fixed` attribute to a Java constant.

Example 157 on page 259 shows an in-line customization that forces the code generator to generate constants for a single attribute with a fixed value.

Example 157. In-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="widgetAttr">
    <sequence>
      ...
    </sequence>
    <attribute name="fixer" type="xsd:int" fixed="7">
      <annotation>
        <appinfo>
          <jaxb:property fixedAttributeAsConstantProperty="true" />
        </appinfo>
      </annotation>
    </attribute>
  </complexType>
  ...
</schema>
```

Example 158 on page 260 shows an external binding file that customizes the generation of a fixed attribute.

Example 158. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:bindings node="xsd:complexType[@name='widgetAttr']">
      <jaxb:bindings node="xsd:attribute[@name='fixer']">
            <jaxb:property fixedAttributeAsConstantProperty="true" />
        </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

Java mapping

In the default mapping, all attributes are mapped to standard Java properties with getter and setter methods. When this customization is applied to an attribute defined using the `fixed` attribute, the attribute is mapped to a Java constant, as shown in Example 159 on page 260.

Example 159. Mapping of a Fixed Value Attribute to a Java Constant

```
@XmlAttribute
public final static type NAME = value;
```

`type` is determined by mapping the base type of the attribute to a Java type using the mappings described in Primitive Types on page 154.

`NAME` is determined by converting the value of the `attribute` element's `name` attribute to all capital letters.

`value` is determined by the value of the `attribute` element's `fixed` attribute.

For example, the attribute defined in Example 157 on page 259 is mapped as shown in Example 160 on page 260.

Example 160. Fixed Value Attribute Mapped to a Java Constant

```
@XmlRootElement(name = "widgetAttr")
public class WidgetAttr {

    ...

  @XmlAttribute
  public final static int FIXER = 7;
```

```
    ...
}
```

Specifying the Base Type of an Element or an Attribute

Overview

Occasionally you need to customize the class of the object generated for an element, or for an attribute defined as part of an XML Schema complex type. For example, you might want to use a more generalized class of object to allow for simple type substitution.

One way to do this is to use the JAXB base type customization. It allows a developer, on a case by case basis, to specify the class of object generated to represent an element or an attribute. The base type customization allows you to specify an alternate mapping between the XML Schema construct and the generated Java object. This alternate mapping can be a simple specialization or a generalization of the default base class. It can also be a mapping of an XML Schema primitive type to a Java class.

Customization usage

To apply the JAXB base type property to an XML Schema construct use the JAXB `baseType` customization element. The `baseType` customization element is a child of the JAXB `property` element, so it must be properly nested.

Depending on how you want to customize the mapping of the XML Schema construct to Java object, you add either the `baseType` customization element's `name` attribute, or a `javaType` child element. The `name` attribute is used to map the default class of the generated object to another class within the same class hierarchy. The `javaType` element is used when you want to map XML Schema primitive types to a Java class.

Important

You cannot use both the `name` attribute and a `javaType` child element in the same `baseType` customization element.

Specializing or generalizing the default mapping

The `baseType` customization element's `name` attribute is used to redefine the class of the generated object to a class within the same Java class hierarchy. The attribute specifies the fully qualified name of the Java class to which the XML Schema construct is mapped. The specified Java class **must** be either a super-class or a sub-class of the Java class that the code generator normally generates for the XML Schema construct. For XML Schema primitive types

that map to Java primitive types, the wrapper class is used as the default base class for the purpose of customization.

For example, an element defined as being of xsd:int uses `java.lang.Integer` as its default base class. The value of the `name` attribute can specify any super-class of `Integer` such as `Number` or `Object`.

Tip

For simple type substitution, the most common customization is to map the primitive types to an `Object` object.

Example 161 on page 263 shows an in-line customization that maps one element in a complex type to a Java `Object` object.

Example 161. In-Line Customization of a Base Type

```
<complexType name="widgetOrderInfo">
  <all>
    <element name="amount" type="xsd:int" />
     <element name="shippingAdress" type="Address>
      <annotation>
        <appinfo>
          <jaxb:property>
            <jaxb:baseType name="java.lang.Object" />
          </jaxb:property>
        </appinfo>
      </annotation>
    </element>
    <element name="type" type="xsd:string"/>
  </all>
</complexType>
```

Example 162 on page 263 shows an external binding file for the customization shown in Example 161 on page 263.

Example 162. External Binding File to Customize a Base Type

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='shippingAddress']">
        <jaxb:property>
```

```
          <jaxb:baseType name="java.lang.Object" />
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

The resulting Java object's `@XmlElement` annotation includes a type property.

The value of the type property is the class object representing the generated object's default base type. In the case of XML Schema primitive types, the class is the wrapper class of the corresponding Java primitive type.

Example 163 on page 264 shows the class generated based on the schema definition in Example 162 on page 263.

Example 163. Java Class with a Modified Base Class

```
public class WidgetOrderInfo {

    protected int amount;
    @XmlElement(required = true)
    protected String type;
    @XmlElement(required = true, type = Address.class)
    protected Object shippingAddress;

    ...
    public Object getShippingAddress() {
        return shippingAddress;
    }


    public void setShippingAddress(Object value) {
        this.shippingAddress = value;
    }

}
```

Usage with javaType

The `javaType` element can be used to customize how elements and attributes defined using XML Schema primitive types are mapped to Java objects. Using the `javaType` element provides a lot more flexibility than simply using the `baseType` element's `name` attribute. The `javaType` element allows you to map a primitive type to any class of object.

For a detailed description of using the `javaType` element, see Specifying the Java Class of an XML Schema Primitive on page 243.

Using A JAXBContext Object

The JAXBContext object allows the Artix ESB's runtime to transform data between XML elements and Java object. Application developers need to instantiate a JAXBContext object they want to use JAXB objects in message handlers and when implementing consumers that work with raw XML messages.

Overview

The JAXBContext object is a low-level object used by the runtime. It allows the runtime to convert between XML elements and their corresponding Java representations. An application developer generally does not need to work with JAXBContext objects. The marshaling and unmarshaling of XML data is typically handled by the transport and binding layers of a JAX-WS application.

However, there are instances when an application will need to manipulate the XML message content directly. In two of these instances:

- Implementing consumers that use raw XML data on page 290
- Working with messages in a handler on page 337

You will need instantiate a JAXBContext object using one of the two available JAXBContext.newInstance() methods.

Best practices

JAXBContext objects are resource intensive to instantiate. It is recommended that an application create as few instances as possible. One way to do this is to create a single JAXBContext object that can manage all of the JAXB objects used by your application and share it among as many parts of your application as possible.

Tip

JAXBContext objects are thread safe.

Getting a JAXBContext object using an object factory

The JAXBContext class provides a newInstance() method, shown in Example 164 on page 268, that takes a list of classes that implement JAXB objects.

Example 164. Getting a JAXB Context Using Classes

```
static JAXBContext newInstance(Class... classesToBeBound)
  throws JAXBException;
```

The returned `JAXBObject` object will be able to marshal and unmarshal data for the JAXB object implemented by the classes passed into the method. It will also be able to work with any classes that are statically referenced from any of the classes passed into the method.

While it is possible to pass the name of every JAXB class used by your application to the `newInstance()` method it is not efficient. A more efficient way to accomplish the same goal is to pass in the object factory, or object factories, generated for your application. The resulting `JAXBContext` object will be able to manage any JAXB classes the specified object factories can instantiate.

Getting a JAXBContext object using package names

The `JAXBContext` class provides a `newInstance()` method, shown in Example 165 on page 268, that takes a colon (`:`) seperated list of package names. The specified packages should contain JAXB objects derived from XML Schema.

Example 165. Getting a JAXB Context Using Classes

```
static JAXBContext newInstance(String contextPath)
  throws JAXBException;
```

The returned `JAXBContext` object will be able to marshal and unmarshal data for all of the JAXB objects implemented by the classes in the specified packages.

Part VI. Advanced Programming Tasks

The JAX-WS programming model offers a number of advanced features.

Developing Asynchronous Applications

JAX-WS provides an easy mechanism for accessing services asynchronously. The SEI can specify additional methods that can be used to access a service asynchronously. The Artix ESB code generators generate the extra methods for you. You simply add the business logic.

In addition to the usual synchronous mode of invocation, Artix ESB supports two forms of asynchronous invocation:

- Polling approach — To invoke the remote operation using the polling approach, you call a method that has no output parameters, but returns a `javax.xml.ws.Response` object. The `Response` object (which inherits from the `javax.util.concurrency.Future` interface) can be polled to check whether or not a response message has arrived.
- Callback approach — To invoke the remote operation using the callback approach, you call a method that takes a reference to a callback object (of `javax.xml.ws.AsyncHandler` type) as one of its parameters. When the response message arrives at the client, the runtime calls back on the `AsyncHandler` object, and gives it the contents of the response message.

WSDL for Asynchronous Examples

Example 166 on page 274 shows the WSDL contract that is used for the asynchronous examples. The contract defines a single interface, `GreeterAsync`, which contains a single operation, greetMeSometime.

Example 166. WSDL Contract for Asynchronous Example

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions xmlns="http://schem
as.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_async_soap_http"
                  xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  targetNamespace="http://apache.org/hello_world_async_soap_http"
                  name="HelloWorld">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_async_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
            elementFormDefault="qualified">
      <element name="greetMeSometime">
        <complexType>
          <sequence>
            <element name="requestType" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeSometimeResponse">
        <complexType>
          <sequence>
            <element name="responseType"
                     type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>

  <wsdl:message name="greetMeSometimeRequest">
    <wsdl:part name="in" element="x1:greetMeSometime"/>
  </wsdl:message>
  <wsdl:message name="greetMeSometimeResponse">
    <wsdl:part name="out"
               element="x1:greetMeSometimeResponse"/>
  </wsdl:message>
```

```
  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="GreeterAsync_SOAPBinding"
                type="tns:GreeterAsync">
    ...
  </wsdl:binding>

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

Generating the Stub Code

Overview

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

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

External Binding Beclaration
: When using an external binding declaration the `jaxws:bindings` element is defined in a file separatel from the WSDL contract. You specify the location of the binding declaration file to **artix wsdl2java** when you generate the stub code.

Embedded Binding Declaration
: When using an embedded binding declaration you embed the `jaxws:bindings` element directly in a WSDL contract, treating it as a WSDL extension. In this case, the settings in `jaxws:bindings` apply only to the immediate parent element.

Using an external binding declaration

The template for a binding declaration file that switches on asynchronous invocations is shown in Example 167 on page 276.

Example 167. Template for an Asynchronous Binding Declaration

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

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

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

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

```
artix wsdl2java -client -b async_binding.xml hello_world.wsdl
```

When you run **artix wsdl2java**, you specify the location of the binding declaration file using the `-b` option.

For more information on artix wsdl2java see artix wsdl2java in the *Artix® ESB Command Reference*.

Using an embedded binding declaration

You can also embed the binding customization directly into the WSDL document defining the service by placing the `jaxws:bindings` element and its associated `jaxws:enableAsynchMapping` child directly into the WSDL. You also must add a namespace declaration for the jaxws prefix.

Example 168 on page 277 shows a WSDL file with an embedded binding declaration that activates the asynchronous mapping for an operation.

Example 168. WSDL with Embedded Binding Declaration for Asynchronous Mapping

```
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
                  ...
                  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
                  ...>
  ...
  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <jaxws:bindings>
```

```
        <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
      </jaxws:bindings>
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

When embedding the binding declaration into the WSDL document you can control the scope affected by the declaration by changing where you place the declaration. When the declaration is placed as a child of the `wsdl:definitions` element the code generator creates asynchronous methods for all of the operations defined in the WSDL document. If it is placed as a child of a `wsdl:portType` element the code generator creates asynchronous methods for all of the operations defined in the interface. If it is placed as a child of a `wsdl:operation` element the code generator creates asynchronous methods for only that operation.

It is not necessary to pass any special options to the code generator when using embedded declarations. The code generator will recognize them and act accordingly.

Generated interface

After generating the stub code in this way, the `GreeterAsync` SEI (in the file `GreeterAsync.java`) is defined as shown in Example 169 on page 278.

Example 169. Service Endpoint Interface with Methods for Asynchronous Invocations

```
package org.apache.hello_world_async_soap_http;

import org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse;
...

public interface GreeterAsync
{
  public Future<?> greetMeSometimeAsync(
        java.lang.String requestType,
        AsyncHandler<GreetMeSometimeResponse> asyncHandler
    );

    public Response<GreetMeSometimeResponse> greetMeSometimeAsync(
        java.lang.String requestType
    );
```

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

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

- Callback approach

```
public Future<?> greetMeSomtimeAsync(java.lang.String requestType,
                                     AsyncHandler<GreetMeSomtimeResponse> asyncHandler);
```

- Polling approach

```
public Response<GreetMeSomeTimeResponse> greetMeSometimeAsync(java.lang.String requestType);
```

Implementing an Asynchronous Client with the Polling Approach

The polling approach is the more straightforward of the two approaches to developing an asynchronous application. The client invokes the asynchronous method called *OperationName*`Async()` and is returned a `Response<T>` object that it polls for a response. What the client does while it is waiting for a response is depends on the requirements of the application. There are two basic patterns for handling the polling:

- Non-blocking polling — You periodically check to see if the result is ready by calling the non-blocking `Response<T>.isDone()` method. If the result is ready, the client processes it. If it not, the client continues doing other things.
- Blocking polling — You call `Response<T>.get()` right away, and block until the response arrives (optionally specifying a timeout).

Using the non-blocking pattern

Example 170 on page 280 illustrates using non-blocking polling to make an asynchronous invocation on the greetMeSometime operation defined in Example 166 on page 274. The client invokes the asynchronous operation and periodically checks to see if the result is returned.

Example 170. Non-Blocking Polling Approach for an Asynchronous Operation Call

```
package demo.hw.client;

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

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

import org.apache.hello_world_async_soap_http.*;

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

  private Client() {}
```

```
  public static void main(String args[]) throws Exception {

    // set up the proxy for the client

❶    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));

❷      while (!greetMeSomeTimeResp.isDone()) {
      // client does some work
      }
❸      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response

      System.exit(0);
  }
}
```

The code in Example 170 on page 280 does the following:

❶ Invokes the `greetMeSometimeAsync()` on the proxy.

The method call returns the `Response<GreetMeSometimeResponse>` object to the client immediately. The Artix ESB runtime handles the details of receiving the reply from the remote endpoint and populating the `Response<GreetMeSometimeResponse>` object.

Note

The runtime transmits the request to the remote endpoint's `greetMeSometime()` method and handles the details of the asynchronous nature of the call transparently. The endpoint, and therefore the service implementation, never worries about the details of how the client intends to wait for a response.

❷ Checks to see if a response has arrived by checking the `isDone()` of the returned `Response` object.

If the response has not arrived, the client continues working before checking again.

❸ When the response arrives, the client retrieves it from the `Response` object using the `get()` method.

Using the blocking pattern

When using the block polling pattern, the `Response` object's `isDone()` is never called. Instead, the `Response` object's `get()` method is called immediately after invoking the remote operation. The `get()` blocks until the response is available.

Tip

You can also pass a timeout limit to the `get()` method.

Example 171 on page 282 shows a client that uses blocking polling.

Example 171. Blocking Polling Approach for an Asynchronous Operation Call

```
package demo.hw.client;

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

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

import org.apache.hello_world_async_soap_http.*;

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

  private Client() {}

  public static void main(String args[]) throws Exception {

    // set up the proxy for the client

    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
      System.exit(0);
  }
}
```

Implementing an Asynchronous Client with the Callback Approach

An alternative approach to making an asynchronous operation invocation is to implement a callback class. You then call the asynchronous remote method that takes the callback object as a parameter. The runtime returns the response to the callback object.

To implement an application that uses callbacks, do the following:

1. Create a callback class that implements the `AsyncHandler` interface.

   Note

   Your callback object can perform any amount of response processing required by your application.

2. Make remote invocations using the *`operationName`*`Async()` that takes the callback object as a parameter and returns a `Future<?>` object.

3. If your client requires access to the response data, you can poll the returned `Future<?>` object's `isDone()` method to see if the remote endpoint has sent the response.

   Tip

   If the callback object does all of the response processing, it is not necessary to check if the response has arrived.

Implementing the callback

The callback class must implement the `javax.xml.ws.AsyncHandler` interface. The interface defines a single method:

```
void handleResponse(Response<T> res);
```

The Artix ESB runtime calls the `handleResponse()` method to notify the client that the response has arrived. Example 172 on page 284 shows an outline of the `AsyncHandler` interface that you must implement.

Example 172. The `javax.xml.ws.AsyncHandler` Interface

```
public interface javax.xml.ws.AsyncHandler
{
  void handleResponse(Response<T> res)
}
```

Example 173 on page 284 shows a callback class for the greetMeSometime operation defined in Example 166 on page 274.

Example 173. Callback Implementation Class

```
package demo.hw.client;

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

import org.apache.hello_world_async_soap_http.types.*;

public class GreeterAsyncHandler implements AsyncHandler<GreetMeSometimeResponse>
{
❶  private GreetMeSometimeResponse reply;

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

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

The callback implementation shown in Example 173 on page 284 does the following:

❶ Defines a member variable, `response`, that holds the response returned from the remote endpoint.

❷ Implements `handleResponse()`.

This implementation simply extracts the response and assigns it to the member variable `reply`.

❸ Implements an added method called `getResponse()`.

This method is a convenience method that extracts the data from `reply` and returns it.

Implementing the consumer

Example 174 on page 285 illustrates a client that uses the callback approach to make an asynchronous call to the GreetMeSometime operation defined in Example 166 on page 274.

Example 174. Callback Approach for an Asynchronous Operation Call

```
package demo.hw.client;

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

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

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  ...

  public static void main(String args[]) throws Exception
  {
    ...
    // Callback approach
❶    GreeterAsyncHandler callback = new GreeterAsyncHandler();

❷    Future<?> response =
      port.greetMeSometimeAsync(System.getProperty("user.name"),
                                callback);
❸    while (!response.isDone())
    {
      // Do some work
    }
❹    resp = callback.getResponse();
    ...
    System.exit(0);
  }
}
```

The code in Example 174 on page 285 does the following:

❶ Instantiates a callback object.

❷ Invokes the `greetMeSometimeAsync()` that takes the callback object on the proxy.

The method call returns the `Future<?>` object to the client immediately.

The Artix ESB runtime handles the details of receiving the reply from the remote endpoint, invoking the callback object's `handleResponse()` method, and populating the `Response<GreetMeSometimeResponse>` object.

Note

The runtime transmits the request to the remote endpoint's `greetMeSometime()` method and handles the details of the asynchronous nature of the call without the remote endpoint's knowledge. The endpoint, and therefore the service implementation, does not need to worry about the details of how the client intends to wait for a response.

❸ Uses the returned `Future<?>` object's `isDone()` method to check if the response has arrived from the remote endpoint.

❹ Invokes the callback object's `getResponse()` method to get the response data.

Catching Exceptions Returned from a Remote Service

Overview

Consumers making asynchronous requests will not receive the same exceptions returned than when they make synchronous requests. Any exceptions returned to the consumer asynchronously are wrapped in an `ExecutionException` exception. The actual exception thrown by the service is stored in the `ExecutionException` exception's `cause` field.

Catching the exception

Exceptions generated by a remote service are thrown locally by the method that passes the response to the consumer's business logic. When the consumer makes a synchronous request, the method making the remote invocation throws the exception. When the consumer makes an asynchronous request, the `Response<T>` object's `get()` method throws the exception. The consumer will not discover that an error was encountered in processing the request until it attempts to retrieve the response message.

Unlike the methods generated by the JAX-WS framework, the `Response<T>` object's `get()` method does not throw either user modeled exceptions nor the generic JAX-WS exceptions. Instead, it throws a `java.util.concurrent.ExecutionException` exception.

Getting the exception details

The framework stores the exception returned from the remote service in the `ExecutionException` exception's `cause` field. The details about the remote exception are extracted by getting the value of the `cause` field and examining the stored exception. The stored exception can be any user defined exception or one of the generic JAX-WS exceptions.

Example

Example 175 on page 287 shows an example of catching an exception using the polling approach.

Example 175. Catching an Exception using the Polling Approach

```
package demo.hw.client;

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

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

```
import org.apache.hello_world_async_soap_http.*;

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

  private Client() {}

  public static void main(String args[]) throws Exception
  {
    ...
    // port is a previously established proxy object.
    Response<GreetMeSometimeResponse> resp =
       port.greetMeSometimeAsync(System.getProperty("user.name"));

    while (!resp.isDone())
    {
      // client does some work
    }

    try ❶
    {
      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
    }
    catch (ExecutionException ee) ❷
    {
       Throwable cause = ee.getCause(); ❸
       System.out.println("Exception "+cause.getClass().getName()+" thrown by the remote
service.");
    }
  }
}
```

The code in Example 175 on page 287 does the following:

❶ Wraps the call to the `Response<T>` object's `get()` method in a try/catch block.

❷ Catches a `ExecutionException` exception.

❸ Extracts the `cause` field from the exception.

If the consumer was using the callback approach the code used to catch the exception would be placed in the callback object where the service's response is extracted.

Using Raw XML Messages

The high-level JAX-WS APIs shield the developer from using native XML messages by marshaling the data into JAXB objects. However, there are cases when it is better to have direct access to the raw XML message data that is passing on the wire. The JAX-WS APIs provide two interfaces that provide access to the raw XML: the `Dispatch` *interface is the client-side interface, and the* `Provider` *interface is the server-side interface.*

Using XML in a Consumer

The `Dispatch` interface is a low-level JAX-WS API that allows you work directly with raw messages. It accepts and returns messages, or payloads, of a number of types including DOM objects, SOAP messages, and JAXB objects. Because it is a low-level API, the `Dispatch` interface does not perform any of the message preparation that the higher-level JAX-WS APIs perform. You must ensure that the messages, or payloads, that you pass to the `Dispatch` object are properly constructed, and make sense for the remote operation being invoked.

Usage Modes

Overview

`Dispatch` objects have two *usage modes*:

- Message mode
- Message Payload mode (Payload mode)

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

Message mode

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

Tip

Message mode is not ideal when working with JAXB objects.

To specify that a `Dispatch` object uses message mode provide the value `java.xml.ws.Service.Mode.MESSAGE` when creating the `Dispatch` object. For more information about creating a `Dispatch` object see Creating a Dispatch object on page 296.

Payload mode

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

Tip

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

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

For more information about creating a `Dispatch` object see Creating a Dispatch object on page 296.

Data Types

Overview

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

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

Using Source objects

A `Dispatch` object accepts and returns objects that are derived from the `javax.xml.transform.Source` interface. `Source` objects are supported by any binding, and in either message mode or payload mode.

`Source` objects are low level objects that hold XML documents. Each `Source` implementation provides methods that access the stored XML documents and then manipulate its contents. The following objects implement the `Source` interface:

`DOMSource`

: Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of `Node` objects that are accessed using the `getNode()` method. Nodes can be either updated or added to the DOM tree using the `setNode()` method.

`SAXSource`

: Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an `InputSource` object that holds the raw data and an `XMLReader` object that parses the raw data.

`StreamSource`

Holds XML messages as a data stream. The data stream can be manipulated the same as any other data stream.

Using SOAPMessage objects

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

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

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

Using DataSource objects

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

- The `Dispatch` object is using the HTTP binding
- The `Dispatch` object is using message mode

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

Using JAXB objects

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

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

For information on creating a `JAXBContext` object see *Using A JAXBContext Object* on page 267.

Working with `Dispatch` Objects

Procedure

To use a `Dispatch` object to invoke a remote service the following sequence should be followed:

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

Creating a Dispatch object

To create a `Dispatch` object do the following:

1. Create a `Service` object to represent the `wsdl:service` element that defines the service on which the `Dispatch` object will make invocations. See Creating a Service Object on page 50.
2. Create the `Dispatch` object using the `Service` object's `createDispatch()` method, shown in Example 176 on page 296.

Example 176. The `createDispatch()` Method

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

Note

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

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

Table 25 on page 297 describes the parameters for the `createDispatch()` method.

Table 25. Parameters for `createDispatch()`

Parameter	Description
portName	Specifies the QName of the `wsdl:port` element that represents the service provider where the `Dispatch` object will make invocations.
type	Specifies the data type of the objects used by the `Dispatch` object. See Data Types on page 293. When working with JAXB objects, this parameter specifies the `JAXBContext` object used to marshal and unmarshal the JAXB objects.
mode	Specifies the usage mode for the `Dispatch` object. See Usage Modes on page 291.

Example 177 on page 297 shows the code for creating a `Dispatch` object that works with `DOMSource` objects in payload mode.

Example 177. Creating a `Dispatch` Object

```
package com.iona.demo;

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

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

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

Constructing request messages

When working with `Dispatch` objects, requests must be built from scratch. The developer is responsible for ensuring that the messages passed to a

`Dispatch` object match a request that the targeted service provider can process. This requires precise knowledge about the messages used by the service provider and what, if any, header information it requires.

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

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

Warning

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

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

Important

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

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

For more information about how services use XML messages see, the WS-I Basic Profile[1].

Synchronous invocation

For consumers that make synchronous invocations that generate a response, use the `Dispatch` object's `invoke()` method shown in Example 178 on page 299.

Example 178. The `Dispatch.invoke()` Method

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

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

Note

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

Example 179 on page 299 shows code for making a synchronous invocation on a remote service using a `DOMSource` object.

Example 179. Making a Synchronous Invocation Using a `Dispatch` Object

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

[1] http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

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

Asynchronous invocation

`Dispatch` objects also support asynchronous invocations. As with the higher level asynchronous APIs discussed in *Developing Asynchronous Applications* on page 273, `Dispatch` objects can use both the polling approach and the callback approach.

When using the polling approach, the `invokeAsync()` method returns a `Response<t>` object that can be polled to see if the response has arrived. Example 180 on page 300 shows the signature of the method used to make an asynchronous invocation using the polling approach.

Example 180. The `Dispatch.invokeAsync()` Method for Polling

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

For detailed information on using the polling approach for asynchronous invocations see Implementing an Asynchronous Client with the Polling Approach on page 280.

When using the callback approach, the `invokeAsync()` method takes an `AsyncHandler` implementation that processes the response when it is returned. Example 181 on page 300 shows the signature of the method used to make an asynchronous invocation using the callback approach.

Example 181. The `Dispatch.invokeAsync()` Method Using a Callback

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

For detailed information on using the callback approach for asynchronous invocations see Implementing an Asynchronous Client with the Callback Approach on page 283.

Note

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

Oneway invocation

When a request does not generate a response, make remote invocations using the `Dispatch` object's `invokeOneWay()`. Example 182 on page 301 shows the signature for this method.

Example 182. The `Dispatch.invokeOneWay()` Method

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

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

Note

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

Example 183 on page 301 shows code for making a oneway invocation on a remote service using a JAXB object.

Example 183. Making a One Way Invocation Using a `Dispatch` Object

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

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

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

Using XML in a Service Provider

The `Provider` interface is a low-level JAX-WS API that allows you to implement a service provider that works directly with messages as raw XML. The messages are not packaged into JAXB objects before being passed to an object that implements the `Provider` interface.

Messaging Modes

Overview

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

- Message mode
- Payload mode

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

Message mode

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

To specify that a `Provider` implementation uses message mode by provide the value `java.xml.ws.Service.Mode.MESSAGE` as the value to the `javax.xml.ws.ServiceMode` annotation, as shown in Example 184 on page 304.

Example 184. Specifying that a `Provider` Implementation Uses Message Mode

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

Payload mode

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

Tip

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

To specify that a `Provider` implementation uses payload mode by provide the value `java.xml.ws.Service.Mode.PAYLOAD` as the value to the `javax.xml.ws.ServiceMode` annotation, as shown in Example 185 on page 305.

Example 185. Specifying that a `Provider` Implementation Uses Payload Mode

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

Tip

If you do not provide a value for the `@ServiceMode` annotation, the `Provider` implementation uses payload mode.

Data Types

Overview

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

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

Using Source objects

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

`DOMSource`

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of `Node` objects that are accessed using the `getNode()` method. Nodes can be either updated or added to the DOM tree using the `setNode()` method.

`SAXSource`

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an `InputSource` object that holds the raw data and an `XMLReader` object that parses the raw data.

`StreamSource`

Holds XML messages as a data stream. The data stream can be manipulated the same as any other data stream.

Important

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

Using SOAPMessage objects

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

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

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

Using DataSource objects

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

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

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

Implementing a `Provider` Object

Overview

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

- An implementation must have the `@WebServiceProvider` annotation.
- An implementation must have a default public constructor.
- An implementation must implement a typed version of the `Provider` interface.

  In other words, you cannot implement a `Provider<T>` interface. You must implement a version of the interface that uses a concrete data type as listed in Data Types on page 306. For example, you can implement an instance of a `Provider<SAXSource>`.

The complexity of implementing the `Provider` interface is in the logic handling the request messages and building the proper responses.

Working with messages

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

WS-I Basic Profile[2] provides guidelines about the messages used by services, including:

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

[2] http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

Warning

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

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

Important

The children of top-level elements might be namespace qualified, but to be certain you will must check their schema definitions.

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

The @WebServiceProvider annotation

To be recognized by JAX-WS as a service implementation, a `Provider` implementation must be decorated with the `@WebServiceProvider` annotation.

Table 26 on page 309 describes the properties that can be set for the `@WebServiceProvider` annotation.

Table 26. `@WebServiceProvider` Properties

Property	Description
portName	Specifies the value of the `name` attribute of the `wsdl:port` element that defines the service's endpoint.
serviceName	Specifies the value of the `name` attribute of the `wsdl:service` element that contains the service's endpoint.
targetNamespace	Specifies the targetname space of the service's WSDL definition.

Property	Description
wsdlLocation	Specifies the URI for the WSDL document defining the service.

All of these properties are optional, and are empty by default. If you leave them empty, Artix ESB creates values using information from the implementation class.

Implementing the invoke() method

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

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

Examples

Example 186 on page 310 shows a `Provider` implementation that works with `SOAPMessage` objects in message mode.

Example 186. `Provider<SOAPMessage>` Implementation

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

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

```
  }

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

❽    SOAPMessage response = mf.createMessage();
     SOAPBody respBody = response.getSOAPBody();
     Name bodyName = sf.createName("getStockPriceResponse");
     respBody.addBodyElement(bodyName);
     SOAPElement respContent = respBody.addChildElement("price");
     respContent.setValue("123.00");
     response.saveChanges();
❾    return response;
    }
    ...
  }
}
```

The code in Example 186 on page 310 does the following:

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

❷ Specifies that this `Provider` implementation uses message mode.

❸ Provides the required default public constructor.

❹ Provides an implementation of the `invoke()` method that takes a `SOAPMessage` object and returns a `SOAPMessage` object.

❺ Extracts the request message from the body of the incoming SOAP message.

❻ Checks the root element of the request message to determine how to process the request.

❼ Creates the factories required for building the response.

❽ Builds the SOAP message for the response.

❾ Returns the response as a `SOAPMessage` object.

Example 187 on page 312 shows an example of a `Provider` implementation using `DOMSource` objects in payload mode.

Example 187. `Provider<DOMSource>` Implementation

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

❶@WebServiceProvider(portName="stockQuoteReporterPort" servi
ceName="stockQuoteReporter")
❷@ServiceMode(value="Service.Mode.PAYLOAD")
public class  stockQuoteReporterProvider implements Pro
vider<DOMSource>
❸public stockQuoteReporterProvider()
  {
  }

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

The code in Example 187 on page 312 does the following:

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

❷ Specifies that this `Provider` implementation uses payload mode.

❸ Provides the required default public constructor.

❹ Provides an implementation of the `invoke()` method that takes a `DOMSource` object and returns a `DOMSource` object.

Working with Contexts

JAX-WS uses contexts to pass metadata along the messaging chain. This metadata, depending on its scope, is accessible to implementation level code. It is also accessible to JAX-WS handlers that operate on the message below the implementation level.

Understanding Contexts

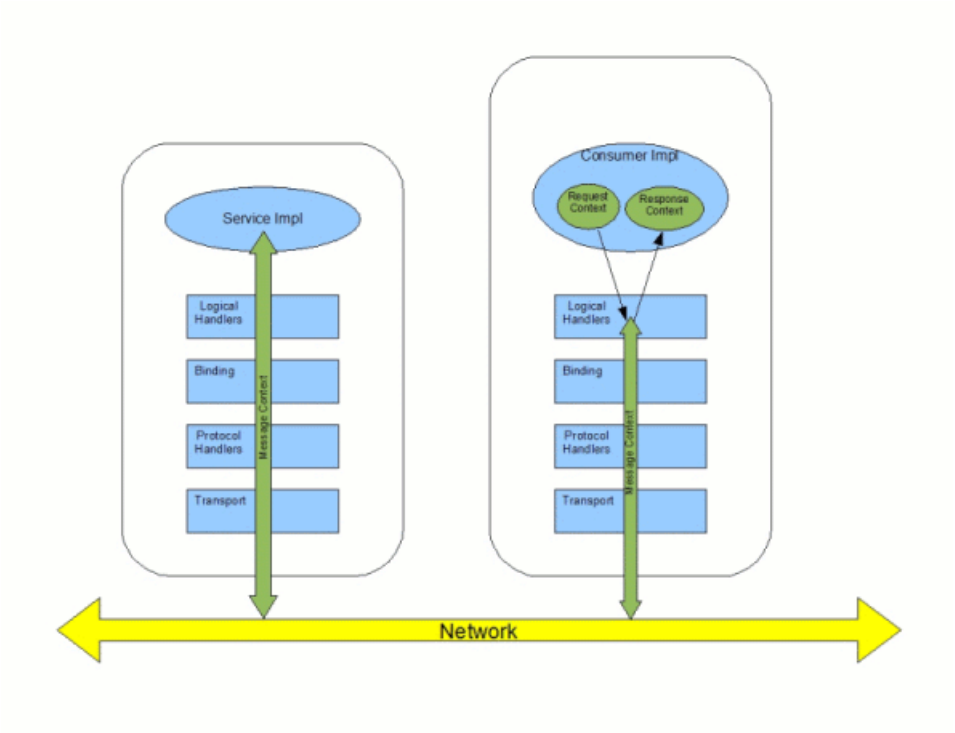
Overview

In many instances it is necessary to pass information about a message to other parts of an application. Artix ESB does this using a context mechanism. Contexts are maps that hold properties relating to an outgoing or an incoming message. The properties stored in the context are typically metadata about the message, and the underlying transport used to communicate the message. For example, the transport specific headers used in transmitting the message, such as the HTTP response code or the JMS correlation ID, are stored in the JAX-WS contexts.

The contexts are available at all levels of a JAX-WS application. However, they differ in subtle ways depending upon where in the message processing stack you are accessing the context. JAX-WS `Handler` implementations have direct access to the contexts and can access all properties that are set in them. Service implementations access contexts by having them injected, and can only access properties that are set in the `APPLICATION` scope. Consumer implementations can only access properties that are set in the `APPLICATION` scope.

Figure 1 on page 315 shows how the context properties pass through Artix ESB. As a message passes through the messaging chain, its associated message context passes along with it.

Figure 1. Message Contexts and Message Processing Path



How properties are stored in a context

The message contexts are all implementations of the `javax.xml.ws.handler.MessageContext` interface. The `MessageContext` interface extends the `java.util.Map<String key, Object value>` interface. `Map` objects store information as key value pairs.

In a message context, properties are stored as name/value pairs. A property's key is a `String` that identifies the property. The value of a property can be any value stored in any Java object. When the value is returned from a message context, the application must know the type to expect and cast accordingly. For example, if a property's value is stored in a `UserInfo` object it is still returned from a message context as an `Object` object that must be cast back into a `UserInfo` object.

Properties in a message context also have a scope. The scope determines where a property can be accessed in the message processing chain.

Property scopes

Properties in a message context are scoped. A property can be in one of the following scopes:

`APPLICATION`

Properties scoped as `APPLICATION` are available to JAX-WS `Handler` implementations, consumer implementation code, and service provider implementation code. If a handler needs to pass a property to the service provider implementation, it sets the property's scope to `APPLICATION`.

All properties set from either the consumer implementation or the service provider implementation contexts are automatically scoped as `APPLICATION`.

`HANDLER`

Properties scoped as `HANDLER` are only available to JAX-WS `Handler` implementations. Properties stored in a message context from a `Handler` implementation are scoped as `HANDLER` by default.

You can change a property's scope using the message context's `setScope()` method. Example 188 on page 316 shows the method's signature.

Example 188. The `MessageContext.setScope()` Method

```
void setScope(String key,
              MessageContext.Scope scope)
  throws java.lang.IllegalArgumentException;
```

The first parameter specifies the property's key. The second parameter specifies the new scope for the property. The scope can be either:

- `MessageContext.Scope.APPLICATION`
- `MessageContext.Scope.HANDLER`

Overview of contexts in handlers

Classes that implement the JAX-WS `Handler` interface have direct access to a message's context information. The message's context information is passed

into the `Handler` implementation's `handleMessage()`, `handleFault()`, and `close()` methods.

`Handler` implementations have access to all of the properties stored in the message context, regardless of their scope. In addition, logical handlers use a specialized message context called a `LogicalMessageContext`. `LogicalMessageContext` objects have methods that access the contents of the message body.

Overview of contexts in service implementations

Service implementations can access properties scoped as `APPLICATION` from the message context. The service provider's implementation object accesses the message context through the `WebServiceContext` object.

For more information see Working with Contexts in a Service Implementation on page 318.

Overview of contexts in consumer implementations

Consumer implementations have indirect access to the contents of the message context. The consumer implementation has two separate message contexts:

- Request context — holds a copy of the properties used for outgoing requests
- Response context — holds a copy of the properties from an incoming response

The dispatch layer transfers the properties between the consumer implementation's message contexts and the message context used by the `Handler` implementations.

When a request is passed to the dispatch layer from the consumer implementation, the contents of the request context are copied into the message context that is used by the dispatch layer. When the response is returned from the service, the dispatch layer processes the message and sets the appropriate properties into its message context. After the dispatch layer processes a response, it copies all of the properties scoped as `APPLICATION` in its message context to the consumer implementation's response context.

For more information see Working with Contexts in a Consumer Implementation on page 325.

Working with Contexts in a Service Implementation

Overview

Context information is made available to service implementations using the `WebServiceContext` interface. From the `WebServiceContext` object you can obtain a `MessageContext` object that is populated with the current request's context properties in the application scope. You can manipulate the values of the properties, and they are propagated back through the response chain.

Note

The `MessageContext` interface inherits from the `java.util.Map` interface. Its contents can be manipulated using the `Map` interface's methods.

Obtaining a context

To obtain the message context in a service implementation do the following:

1. Declare a variable of type `WebServiceContext`.
2. Decorate the variable with the `javax.annotation.Resource` annotation to indicate that the context information is being injected into the variable.
3. Obtain the `MessageContext` object from the `WebServiceContext` object using the `getMessageContext()` method.

Important

`getMessageContext()` can only be used in methods that are decorated with the `@WebMethod` annotation.

Example 189 on page 318 shows code for obtaining a context object.

Example 189. Obtaining a Context Object in a Service Implementation

```
import javax.xml.ws.*;
import javax.xml.ws.handler.*;
import javax.annotation.*;
```

```
@WebServiceProvider
public class WidgetServiceImpl
{
  @Resource
  WebServiceContext wsc;

  @WebMethod
  public String getColor(String itemNum)
  {
    MessageContext context = wsc.getMessageContext();
  }

  ...
}
```

Reading a property from a context

Once you have obtained the `MessageContext` object for your implementation, you can access the properties stored there using the `get()` method shown in Example 190 on page 319.

Example 190. The `MessageContext.get()` Method

```
V get(Object key);
```

Note

This `get()` is inherited from the `Map` interface.

The *`key`* parameter is the string representing the property you want to retrieve from the context. The `get()` returns an object that must be cast to the proper type for the property. Table 27 on page 321 lists a number of the properties that are available in a service implementation's context.

Important

Changing the values of the object returned from the context also changes the value of the property in the context.

Example 191 on page 320 shows code for getting the name of the WSDL `operation` element that represents the invoked operation.

Example 191. Getting a Property from a Service's Message Context

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

  ...
  // MessageContext context retrieved in a previous example
  QName wsdl_operation = (QName)context.get(Message.WSDL_OPER
ATION);
```

Setting properties in a context

Once you have obtained the `MessageContext` object for your implementation, you can set properties, and change existing properties, using the `put()` method shown in Example 192 on page 320.

Example 192. The `MessageContext.put()` Method

```
V put(K key,
      V value)
  throws ClassCastException, IllegalArgumentException, NullPointerException;
```

If the property being set already exists in the message context, the `put()` method replaces the existing value with the new value and returns the old value. If the property does not already exist in the message context, the `put()` method sets the property and returns `null`.

Example 193 on page 320 shows code for setting the response code for an HTTP request.

Example 193. Setting a Property in a Service's Message Context

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

  ...
  // MessageContext context retrieved in a previous example
  context.put(Message.RESPONSE_CODE, new Integer(404));
```

Supported contexts

Table 27 on page 321 lists the properties accessible through the context in a service implementation object.

Table 27. Properties Available in the Service Implementation Context

Base Class	
Property Name	**Description**
`org.apache.cxf.message.Message`	
`PROTOCOL_HEADERS`[a]	Specifies the transport specific header information. The value is stored as a `java.util.Map<String, List<String>>`.
`RESPONSE_CODE`[a]	Specifies the response code returned to the consumer. The value is stored as an `Integer` object.
`ENDPOINT_ADDRESS`	Specifies the address of the service provider. The value is stored as a `String`.
`HTTP_REQUEST_METHOD`[a]	Specifies the HTTP verb sent with a request. The value is stored as a `String`.
`PATH_INFO`[a]	Specifies the path of the resource being requested. The value is stored as a `String`. The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URI is `http://cxf.apache.org/demo/widgets` the path is `/demo/widgets`.
`QUERY_STRING`[a]	Specifies the query, if any, attached to the URI used to invoke the request. The value is stored as a `String`. Queries appear at the end of the URI after a `?`. For example, if a request is made to `http://cxf.apache.org/demo/widgets?color` the query is `color`.
`MTOM_ENABLED`	Specifies whether or not the service provider can use MTOM for SOAP attachments. The value is stored as a `Boolean`.
`SCHEMA_VALIDATION_ENABLED`	Specifies whether or not the service provider validates messages against a schema. The value is stored as a `Boolean`.
`FAULT_STACKTRACE_ENABLED`	Specifies if the runtime provides a stack trace along with a fault message. The value is stored as a `Boolean`.
`CONTENT_TYPE`	Specifies the MIME type of the message. The value is stored as a `String`.
`BASE_PATH`	Specifies the path of the resource being requested. The value is stored as a `java.net.URL`.

Base Class	
Property Name	**Description**
	The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URL is `http://cxf.apache.org/demo/widgets` the base path is `/demo/widgets`.
`ENCODING`	Specifies the encoding of the message. The value is stored as a `String`.
`FIXED_PARAMETER_ORDER`	Specifies whether the parameters must appear in the message in a particular order. The value is stored as a `Boolean`.
`MAINTAIN_SESSION`	Specifies if the consumer wants to maintain the current session for future requests. The value is stored as a `Boolean`.
`WSDL_DESCRIPTION`[a]	Specifies the WSDL document that defines the service being implemented. The value is stored as a `org.xml.sax.InputSource` object.
`WSDL_SERVICE`[a]	Specifies the qualified name of the `wsdl:service` element that defines the service being implemented. The value is stored as a `QName`.
`WSDL_PORT`[a]	Specifies the qualified name of the `wsdl:port` element that defines the endpoint used to access the service. The value is stored as a `QName`.
`WSDL_INTERFACE`[a]	Specifies the qualified name of the `wsdl:portType` element that defines the service being implemented. The value is stored as a `QName`.
`WSDL_OPERATION`[a]	Specifies the qualified name of the `wsdl:operation` element that corresponds to the operation invoked by the consumer. The value is stored as a `QName`.
`javax.xml.ws.handler.MessageContext`	
`MESSAGE_OUTBOUND_PROPERTY`	Specifies if a message is outbound. The value is stored as a `Boolean`. `true` specifies that a message is outbound.
`INBOUND_MESSAGE_ATTACHMENTS`	Contains any attachments included in the request message. The value is stored as a `java.util.Map<String, DataHandler>`. The key value for the map is the MIME Content-ID for the header.
`OUTBOUND_MESSAGE_ATTACHMENTS`	Contains any attachments for the response message. The value is stored as a `java.util.Map<String, DataHandler>`. The key value for the map is the MIME Content-ID for the header.

Base Class	
Property Name	**Description**
WSDL_DESCRIPTION	Specifies the WSDL document that defines the service being implemented. The value is stored as a `org.xml.sax.InputSource` object.
WSDL_SERVICE	Specifies the qualified name of the `wsdl:service` element that defines the service being implemented. The value is stored as a `QName`.
WSDL_PORT	Specifies the qualified name of the `wsdl:port` element that defines the endpoint used to access the service. The value is stored as a `QName`.
WSDL_INTERFACE	Specifies the qualified name of the `wsdl:portType` element that defines the service being implemented. The value is stored as a `QName`.
WSDL_OPERATION	Specifies the qualified name of the `wsdl:operation` element that corresponds to the operation invoked by the consumer. The value is stored as a `QName`.
HTTP_RESPONSE_CODE	Specifies the response code returned to the consumer. The value is stored as an `Integer` object.
HTTP_REQUEST_HEADERS	Specifies the HTTP headers on a request. The value is stored as a `java.util.Map<String, List<String>>`.
HTTP_RESPONSE_HEADERS	Specifies the HTTP headers for the response. The value is stored as a `java.util.Map<String, List<String>>`.
HTTP_REQUEST_METHOD	Specifies the HTTP verb sent with a request. The value is stored as a `String`.
SERVLET_REQUEST	Contains the servlet's request object. The value is stored as a `javax.servlet.http.HttpServletRequest`.
SERVLET_RESPONSE	Contains the servlet's response object. The value is stored as a `javax.servlet.http.HttpResponse`.
SERVLET_CONTEXT	Contains the servlet's context object. The value is stored as a `javax.servlet.ServletContext`.
PATH_INFO	Specifies the path of the resource being requested. The value is stored as a `String`. The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URL is `http://cxf.apache.org/demo/widgets` the path is `/demo/widgets`.

Base Class	
Property Name	**Description**
QUERY_STRING	Specifies the query, if any, attached to the URI used to invoke the request. The value is stored as a `String`. Queries appear at the end of the URI after a `?`. For example, if a request is made to `http://cxf.apache.org/demo/widgets?color` the query string is `color`.
REFERENCE_PARAMETERS	Specifies the WS-Addressing reference parameters. This includes all of the SOAP headers whose `wsa:IsReferenceParameter` attribute is set to `true`. The value is stored as a `java.util.List`.
org.apache.cxf.transport.jms.JMSConstants	
JMS_SERVER_HEADERS	Contains the JMS message headers. For more information see Working with JMS Message Properties on page 329.

[a]When using HTTP this property is the same as the standard JAX-WS defined property.

Working with Contexts in a Consumer Implementation

Overview

Consumer implementations have access to context information through the `BindingProvider` interface. The `BindingProvider` instance holds context information in two separate contexts:

Request Context

The *request context* enables you to set properties that affect outbound messages. Request context properties are applied to a specific port instance and, once set, the properties affect every subsequent operation invocation made on the port, until such time as a property is explicitly cleared. For example, you might use a request context property to set a connection timeout or to initialize data for sending in a header.

Response Context

The *response context* enables you to read the property values set by the response to the last operation invocation made from the current thread. Response context properties are reset after every operation invocation. For example, you might access a response context property to read header information received from the last inbound message.

Important

Only information that is placed in the application scope of a message context can be accessed by the consumer implementation.

Obtaining a context

Contexts are obtained using the `javax.xml.ws.BindingProvider` interface. The `BindingProvider` interface has two methods for obtaining a context:

`getRequestContext()`

The `getRequestContext()` method, shown in Example 194 on page 326, returns the request context as a `Map` object. The returned `Map` object can be used to directly manipulate the contents of the context.

Example 194. The `getRequestContext()` Method

```
Map<String, Object> getRequestContext();
```

`getResponseContext()`

The `getResponseContext()`, shown in Example 195 on page 326, returns the response context as a `Map` object. The returned `Map` object's contents reflect the state of the response context's contents from the most recent successful request on a remote service made in the current thread.

Example 195. The `getResponseContext()` Method

```
Map<String, Object> getResponseContext();
```

Since proxy objects implement the `BindingProvider` interface, a `BindingProvider` object can be obtained by casting a proxy object. The contexts obtained from the `BindingProvider` object are only valid for operations invoked on the proxy object used to create it.

Example 196 on page 326 shows code for obtaining the request context for a proxy.

Example 196. Getting a Consumer's Request Context

```
// Proxy widgetProxy obtained previously
BindingProvider bp = (BindingProvider)widgetProxy;
Map<String, Object> responseContext = bp.getResponseContext();
```

Reading a property from a context

Consumer contexts are stored in `java.util.Map<String, Object>` objects. The map has keys that are String objects and values that contain arbitrary objects. Use `java.util.Map.get()` to access an entry in the map of response context properties.

To retrieve a particular context property, *`ContextPropertyName`*, use the code shown in Example 197 on page 327.

Example 197. Reading a Response Context Property

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

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

Setting properties in a context

Consumer contexts are hash maps stored in `java.util.Map<String, Object>` objects. The map has keys that are String objects and values that are arbitrary objects. To set a property in a context use the `java.util.Map.put()` method.

Tip

While you can set properties in both the request context and the response context, only the changes made to the request context have any impact on message processing. The properties in the response context are reset when each remote invocation is completed on the current thread.

The code shown in Example 198 on page 327 changes the address of the target service provider by setting the value of the `BindingProvider.ENDPOINT_ADDRESS_PROPERTY`.

Example 198. Setting a Request Context Property

```
// Set request context property.
java.util.Map<String, Object> requestContext =
     ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, "http://localhost:8080/wid
gets");

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

Important

Once a property is set in the request context its value is used for all subsequent remote invocations. You can change the value and the changed value will then be used.

Supported contexts

Artix ESB supports the following context properties in consumer implementations:

Table 28. Consumer Context Properties

Base Class	
Property Name	**Description**
javax.xml.ws.BindingProvider	
ENDPOINT_ADDRESS_PROPERTY	Specifies the address of the target service. The value is stored as a `String`.
USERNAME_PROPERTY[a]	Specifies the username used for HTTP basic authentication. The value is stored as a `String`.
PASSWORD_PROPERTY[b]	Specifies the password used for HTTP basic authentication. The value is stored as a `String`.
SESSION_MAINTAIN_PROPERTY[c]	Specifies if the client wants to maintain session information. The value is stored as a `Boolean` object.
org.apache.cxf.ws.addressing.JAXWSAConstants	
CLIENT_ADDRESSING_PROPERTIES	Specifies the WS-Addressing information used by the consumer to contact the desired service provider. The value is stored as a `org.apache.cxf.ws.addressing.AddressingProperties`.
org.apache.cxf.transports.jms.context.JMSConstants	
JMS_CLIENT_REQUEST_HEADERS	Contains the JMS headers for the message. For more information see Working with JMS Message Properties on page 329.

[a] This property is overridden by the username defined in the HTTP security settings.

[b] This property is overridden by the password defined in the HTTP security settings.

[c] The Artix ESB ignores this property.

Working with JMS Message Properties

The Artix ESB JMS transport has a context mechanism that can be used to inspect a JMS message's properties. The context mechanism can also be used to set a JMS message's properties.

Inspecting JMS Message Headers

Consumers and services use different context mechanisms to access the JMS message header properties. However, both mechanisms return the header properties as a `org.apache.cxf.transports.jms.context.JMSMessageHeadersType` object.

Getting the JMS Message Headers in a Service

To get the JMS message header properties from the `WebServiceContext` object, do the following:

1. Obtain the context as described in Obtaining a context on page 318.
2. Get the message headers from the message context using the message context's `get()` method with the parameter `org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS`.

Example 199 on page 330 shows code for getting the JMS message headers from a service's message context:

Example 199. Getting JMS Message Headers in a Service Implementation

```
import org.apache.cxf.transport.jms.JMSConstants;
import org.apache.cxf.transports.jms.context.JMSMessageHeadersType;

@WebService(serviceName = "HelloWorldService",
                          portName = "HelloWorldPort",
                          endpointInterface = "org.apache.cxf.hello_world_jms.HelloWorld
PortType",
                          targetNamespace = "http://cxf.apache.org/hello_world_jms")
  public class GreeterImplTwoWayJMS implements HelloWorldPortType
  {
    @Resource
    protected WebServiceContext wsContext;
    ...

    @WebMethod
    public String greetMe(String me)
    {
      MessageContext mc = wsContext.getMessageContext();
      JMSMessageHeadersType headers = (JMSMessageHeadersType) mc.get(JMSConstants.JMS_SERV
ER_HEADERS);
       ...
     }
```

```
        ...
}
```

Getting JMS Message Header Properties in a Consumer

Once a message is successfully retrieved from the JMS transport you can inspect the JMS header properties using the consumer's response context. In addition, you can see how long the client waits for a response before timing out.

You can To get the JMS message headers from a consumer's response context do the following:

1. Get the response context as described in Obtaining a context on page 325.
2. Get the JMS message header properties from the response context using the context's `get()` method with

   `org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS`

   as the parameter.

Example 200 on page 331 shows code for getting the JMS message header properties from a consumer's response context.

Example 200. Getting the JMS Headers from a Consumer Response Header

```
import org.apache.cxf.transports.jms.context.*;
// Proxy greeter initialized previously
❶BindingProvider  bp = (BindingProvider)greeter;
❷Map<String, Object> responseContext = bp.getResponseContext();
❸JMSMessageHeadersType responseHdr = (JMSMessageHeadersType)
                           responseContext.get(JMSConstants.JMS_CLIENT_REQUEST_HEADERS);
...
}
```

The code in Example 200 on page 331 does the following:

❶ Casts the proxy to a `BindingProvider`.

❷ Gets the response context.

❸ Retrieves the JMS message headers from the response context.

Inspecting the Message Header Properties

Standard JMS Header Properties

Table 29 on page 332 lists the standard properties in the JMS header that you can inspect.

Table 29. JMS Header Properties

Property Name	Property Type	Getter Method
Correlation ID	string	`getJMSCorralationID()`
Delivery Mode	int	`getJMSDeliveryMode()`
Message Expiration	long	`getJMSExpiration()`
Message ID	string	`getJMSMessageID()`
Priority	int	`getJMSPriority()`
Redelivered	boolean	`getJMSRedlivered()`
Time Stamp	long	`getJMSTimeStamp()`
Type	string	`getJMSType()`
Time To Live	long	`getTimeToLive()`

Optional Header Properties

In addition, you can inspect any optional properties stored in the JMS header using `JMSMessageHeadersType.getProperty()`. The optional properties are returned as a `List` of `org.apache.cxf.transports.jms.context.JMSPropertyType`. Optional properties are stored as name/value pairs.

Example

Example 201 on page 332 shows code for inspecting some of the JMS properties using the response context.

Example 201. Reading the JMS Header Properties

```
// JMSMessageHeadersType messageHdr retrieved previously
❶System.out.println("Correlation ID: "+messageHdr.getJMSCorrelationID());
❷System.out.println("Message Priority: "+messageHdr.getJMSPriority());
❸System.out.println("Redelivered: "+messageHdr.getRedelivered());

JMSPropertyType prop = null;
```

```
❹List<JMSPropertyType> optProps = messageHdr.getProperty();
❺Iterator<JMSPropertyType> iter = optProps.iterator();
❻while (iter.hasNext())
{
  prop = iter.next();
   System.out.println("Property name: "+prop.getName());
   System.out.println("Property value: "+prop.getValue());
}
```

The code in Example 201 on page 332 does the following:

❶ Prints the value of the message's correlation ID.

❷ Prints the value of the message's priority property.

❸ Prints the value of the message's redelivered property.

❹ Gets the list of the message's optional header properties.

❺ Gets an `Iterator` to traverse the list of properties.

❻ Iterates through the list of optional properties and prints their name and value.

Setting JMS Properties

Using the request context in a consumer endpoint, you can set a number of the JMS message header properties and the consumer endpoint's timeout value. These properties are valid for a single invocation. You must reset them each time you invoke an operation on the service proxy.

Note

You cannot set header properties in a service.

JMS Header Properties

Table 30 on page 334 lists the properties in the JMS header that can be set using the consumer endpoint's request context.

Table 30. Settable JMS Header Properties

Property Name	Property Type	Setter Method
Correlation ID	string	`setJMSCorralationID()`
Delivery Mode	int	`setJMSDeliveryMode()`
Priority	int	`setJMSPriority()`
Time To Live	long	`setTimeToLive()`

To set these properties do the following:

1. Create an `org.apache.cxf.transports.jms.context.JMSMessageHeadersType` object.
2. Populate the values you want to set using the appropriate setter methods described in Table 30 on page 334.
3. Set the values to the request context by calling the request context's `put()` method using `org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_REQUEST_HEADERS`

as the first argument, and the new `JMSMessageHeadersType` object as the second argument.

Optional JMS Header Properties

You can also set optional properties to the JMS header. Optional JMS header properties are stored in the `JMSMessageHeadersType` object that is used to set the other JMS header properties. They are stored as a `List` object containing `org.apache.cxf.transports.jms.context.JMSPropertyType` objects. To add optional properties to the JMS header do the following:

1. Create a `JMSPropertyType` object.
2. Set the property's name field using `setName()`.
3. Set the property's value field using `setValue()`.
4. Add the property to the JMS message header using `JMSMessageHeadersType.getProperty().add(JMSPropertyType)`.
5. Repeat the procedure until all of the properties have been added to the message header.

Client Receive Timeout

In addition to the JMS header properties, you can set the amount of time a consumer endpoint waits for a response before timing out. You set the value by calling the request context's `put()` method with `org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT` as the first argument and a long representing the amount of time in milliseconds that you want the consumer to wait as the second argument.

Example

Example 202 on page 335 shows code for setting some of the JMS properties using the request context.

Example 202. Setting JMS Properties using the Request Context

```
import org.apache.cxf.transports.jms.context.*;
 // Proxy greeter initialized previously
❶InvocationHandler handler = Proxy.getInvocationHandler(greeter);


BindingProvider bp= null;
❷if (handler instanceof BindingProvider)
```

```
{
❸  bp = (BindingProvider)handler;
❹  Map<String, Object> requestContext = bp.getRequestContext();

❺  JMSMessageHeadersType requestHdr = new JMSMessageHeadersType();
❻  requestHdr.setJMSCorrelationID("WithBob");
❼  requestHdr.setJMSExpiration(3600000L);


❽  JMSPropertyType prop = new JMSPropertyType;
❾  prop.setName("MyProperty");
   prop.setValue("Bluebird");
❿  requestHdr.getProperty().add(prop);

⓫  requestContext.put(JMSConstants.CLIENT_REQUEST_HEADERS, requestHdr);

⓬  requestContext.put(JMSConstants.CLIENT_RECEIVE_TIMEOUT, new Long(1000));
}
```

The code in Example 202 on page 335 does the following:

❶ Gets the `InvocationHandler` for the proxy whose JMS properties you want to change.

❷ Checks to see if the `InvocationHandler` is a `BindingProvider`.

❸ Casts the returned `InvocationHandler` object into a `BindingProvider` object to retrieve the request context.

❹ Gets the request context.

❺ Creates a `JMSMessageHeadersType` object to hold the new message header values.

❻ Sets the Correlation ID.

❼ Sets the Expiration property to 60 minutes.

❽ Creates a new `JMSPropertyType` object.

❾ Sets the values for the optional property.

❿ Adds the optional property to the message header.

⓫ Sets the JMS message header values into the request context.

⓬ Sets the client receive timeout property to 1 second.

Writing Handlers

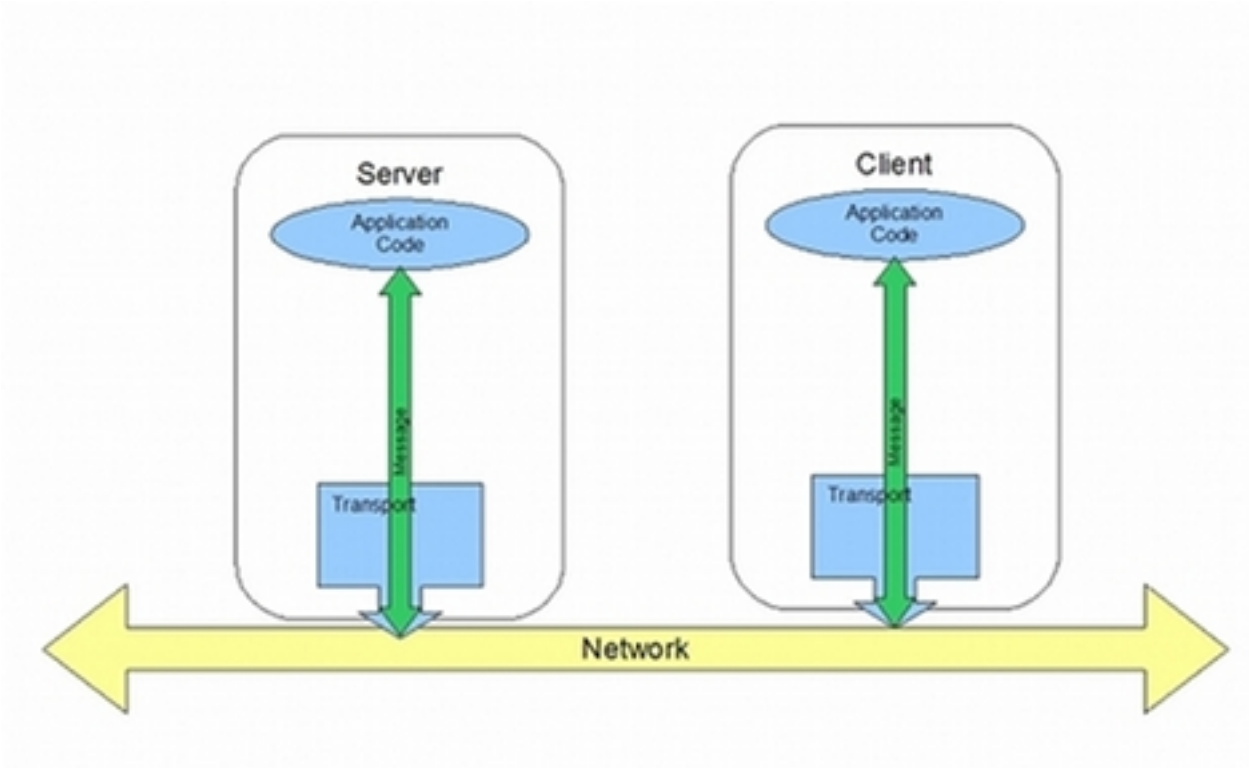
JAX-WS provides a flexible plug-in framework for adding message processing modules to an application. These modules, known as handlers, are independent of the application level code and can provide low-level message processing capabilities.

Handlers: An Introduction

Overview

When a service proxy invokes an operation on a service, the operation's parameters are passed to Artix ESB where they are built into a message and placed on the wire. When the message is received by the service, Artix ESB reads the message from the wire, reconstructs the message, and then passes the operation parameters to the application code responsible for implementing the operation. When the application code is finished processing the request, the reply message undergoes a similar chain of events on its trip to the service proxy that originated the request. This is shown in Figure 2 on page 338.

Figure 2. Message Exchange Path

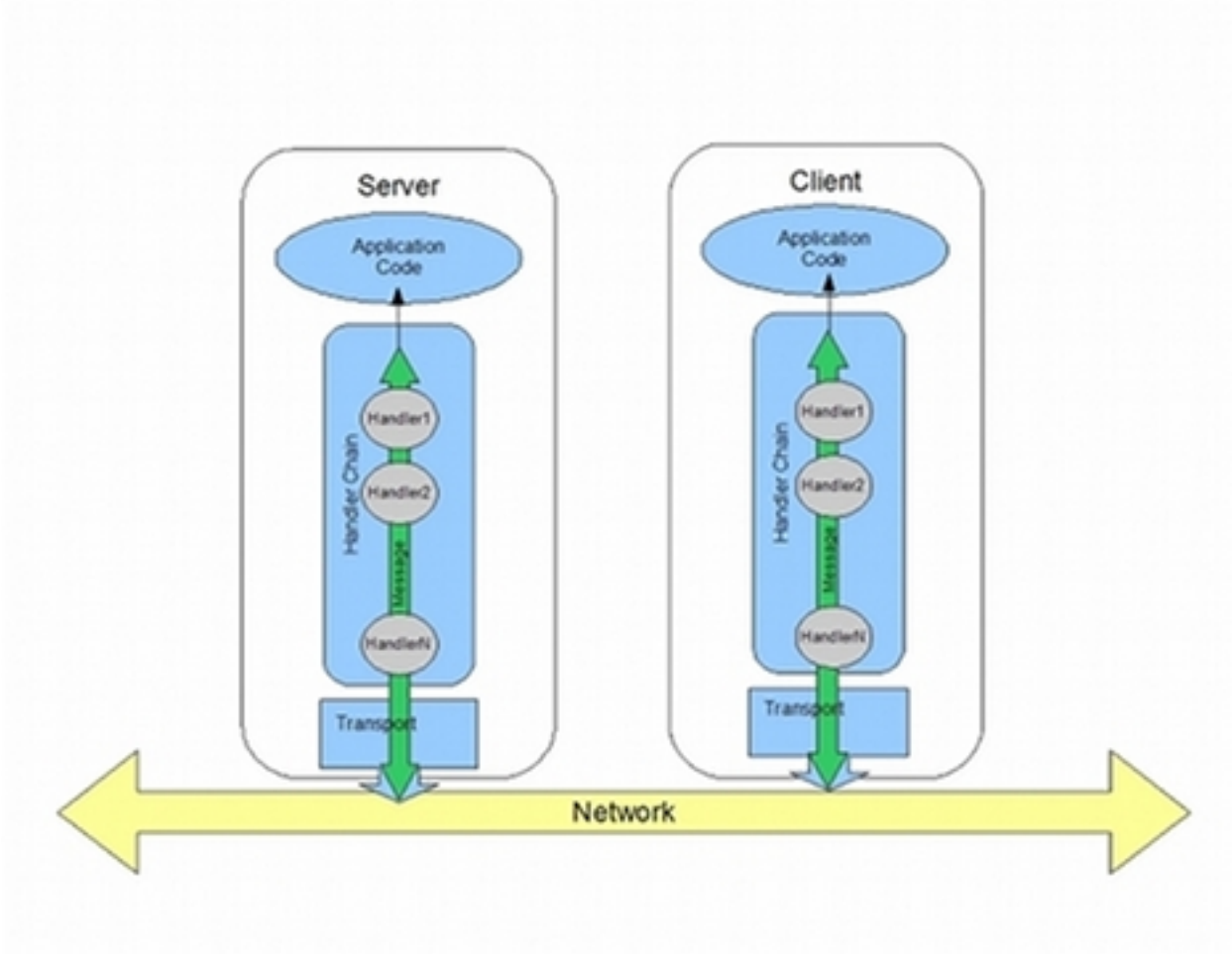


JAX-WS defines a mechanism for manipulating the message data between the application level code and the network. For example, you might want the

message data passed over the open network to be encrypted using a proprietary encryption mechanism. You could write a JAX-WS handler that encrypted and decrypted the data. Then you could insert the handler into the message processing chains of all clients and servers.

As shown in Figure 3 on page 340, the handlers are placed in a chain that is traversed between the application level code and the transport code that places the message onto the network.

Figure 3. Message Exchange Path with Handlers



Handler types

The JAX-WS specification defines two basic handler types:

Logical Handler

Logical handlers can process the message payload and the properties stored in the message context. For example, if the application uses pure

XML messages, the logical handlers have access to the entire message. If the application uses SOAP messages, the logical handlers have access to the contents of the SOAP body. They do not have access to either the SOAP headers or any attachments unless they were placed into the message context.

Logical handlers are placed closest to the application code on the handler chain. This means that they are executed first when a message is passed from the application code to the transport. When a message is received from the network and passed back to the application code, the logical handlers are executed last.

Protocol Handler

Protocol handlers can process the entire message received from the network and the properties stored in the message context. For example, if the application uses SOAP messages, the protocol handlers would have access to the contents of the SOAP body, the SOAP headers, and any attachments.

Protocol handlers are placed closest to the transport on the handler chain. This means that they are executed first when a message is received from the network. When a message is sent to the network from the application code, the protocol handlers are executed last.

Tip

The only protocol handler supported by Artix ESB is specific to SOAP.

Implementation of handlers

The differences between the two handler types are very subtle and they share a common base interface. Because of their common parentage, logical handlers and protocol handlers share a number of methods that must be implemented, including:

handleMessage()

The `handleMessage()` method is the central method in any handler. It is the method responsible for processing normal messages.

handleFault()

`handleFault()` is the method responsible for processing fault messages.

close()

`close()` is called on all executed handlers in a handler chain when a message has reached the end of the chain. It is used to clean up any resources consumed during message processing.

The differences between the implementation of a logical handler and the implementation of a protocol handler revolve around the following:

- The specific interface that is implemented

  All handlers implement an interface that derives from the `Handler` interface. Logical handlers implement the `LogicalHandler` interface. Protocol handlers implement protocol specific extensions of the `Handler` interface. For example, SOAP handlers implement the `SOAPHandler` interface.

- The amount of information available to the handler

  Protocol handlers have access to the contents of messages and all of the protocol specific information that is packaged with the message content. Logical handlers can only access the contents of the message. Logical handlers have no knowledge of protocol details.

Adding handlers to an application

To add a handler to an application you must do the following:

1. Determine whether the handler is going to be used on the service providers, the consumers, or both.
2. Determine which type of handler is the most appropriate for the job.
3. Implement the proper interface.

   To implement a logical handler see Implementing a Logical Handler on page 343.

   To implement a protocol handler see Implementing a Protocol Handler on page 352.

4. Configure on page 364 your endpoint(s) to use the handlers.

Implementing a Logical Handler

Overview

Logical handlers implement the `javax.xml.ws.handler.LogicalHandler` interface. The `LogicalHandler` interface, shown in Example 203 on page 343 passes a `LogicalMessageContext` object to the `handleMessage()` method and the `handleFault()` method. The context object provides access to the *body* of the message and to any properties set into the message exchange's context.

Example 203. `LogicalHandler` Synopsis

```
public interface LogicalHandler extends Handler
{
  boolean handleMessage(LogicalMessageContext context);
  boolean handleFault(LogicalMessageContext context);
  void close(LogicalMessageContext context);
}
```

Procedure

To implement a logical hander you do the following:

1. Implement any initialization logic required by the handler.
2. Implement the message handling logic.
3. Implement the fault handling logic.
4. Implement the logic for closing the handler when it is finished.
5. Implement any logic for cleaning up the handler's resources before it is destroyed.

Handling Messages in a Logical Handler

Overview

Normal message processing is handled by the `handleMessage()` method.

The `handleMessage()` method receives a `LogicalMessageHandler` object that provides access to the message body and any properties stored in the message context.

The `handleMessage()` method returns either `true` or `false` depending on how message processing is to continue. It can also throw an exception.

Getting the message data

The `LogicalMessageContext` object passed into logical message handlers allows access to the message body using the context's `getMessage()` method. The `getMessage()` method, shown in Example 204 on page 344, returns the message payload as a `LogicalMessage` object.

Example 204. Method for Getting the Message Payload in a Logical Handler

```
LogicalMessage getMessage();
```

Once you have the `LogicalMessage` object, you can use it to manipulate the message body. The `LogicalMessage` interface, shown in Example 205 on page 344, has getters and setters for working with the actual message body.

Example 205. Logical Message Holder

```
LogicalMessage {

  Source getPayload();

  Object getPayload(JAXBContext context);

  void setPayload(Object payload,
                  JAXBContext context);

  void setPayload(Source payload);

}
```

Important

The contents of the message payload are determined by the type of binding in use. The SOAP binding only allows access to the SOAP body of the message. The XML binding allows access to the entire message body.

Working with the message body as an XML object

One pair of getters and setters of the logical message work with the message payload as a `javax.xml.transform.dom.DOMSource` object.

The `getPayload()` method that has no parameters returns the message payload as a `DOMSource` object. The returned object is the actual message payload. Any changes made to the returned object change the message body immediately.

You can replace the body of the message with a `DOMSource` object using the `setPayload()` method that takes the single `Source` object.

Working with the message body as a JAXB object

The other pair of getters and setters allow you to work with the message payload as a JAXB object. They use a `JAXBContext` object to transform the message payload into JAXB objects.

To use the JAXB objects you do the following:

1. Get a `JAXBContext` object that can manage the data types in the message body.

   For information on creating a `JAXBContext` object see *Using A JAXBContext Object* on page 267.

2. Get the message body as shown in Example 206 .

   Example 206. Getting the Message Body as a JAXB Object

   ```
   JAXBContext jaxbc = JAXBContext(myObjectFactory.class);
   Object body = message.getPayload(jaxbc);
   ```

3. Cast the returned object to the proper type.

4. Manipulate the message body as needed.

5. Put the updated message body back into the context as shown in Example 207 .

Example 207. Updating the Message Body Using a JAXB Object

```
message.setPayload(body, jaxbc);
```

Working with context properties

The logical message context passed into a logical handler is an instance of the application's message context and can access all of the properties stored in it. Handlers have access to properties at both the `APPLICATION` scope and the `HANDLER` scope.

Like the application's message context, the logical message context is a subclass of Java `Map`. To access the properties stored in the context, you use the `get()` method and `put()` method inherited from the `Map` interface.

By default, any properties you set in the message context from inside a logical handler are assigned a scope of `HANDLER`. If you want the application code to be able to access the property you need to use the context's `setScope()` method to explicitly set the property's scope to `APPLICATION`.

For more information on working with properties in the message context see Understanding Contexts on page 314.

Determining the direction of the message

It is often important to know the direction a message is passing through the handler chain. For example, you would want to retrieve a security token from incoming requests and attach a security token to an outgoing response.

The direction of the message is stored in the message context's outbound message property. You retrieve the outbound message property from the message context using the `MessageContext.MESSAGE_OUTBOUND_PROPERTY` key as shown in Example 208 on page 346.

Example 208. Getting the Message's Direction from the SOAP Message Context

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUT
BOUND_PROPERTY);
```

The property is stored as a `Boolean` object. You can use the object's `booleanValue()` method to determine the property's value. If the property is set to `true`, the message is outbound. If the property is set to `false` the message is inbound.

Determining the return value

How the `handleMessage()` method completes its message processing has a direct impact on how message processing proceeds. It can complete by doing one of the following actions:

I. Return `true`—Returning `true` signals to the Artix ESB runtime that message processing should continue normally. The next handler, if any, has its `handleMessage()` invoked.

II. Return `false`—Returning `false` signals to the Artix ESB runtime that normal message processing must stop. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

For request-response message exchanges the following happens:

1. The direction of message processing is reversed.

   For example, if a request is being processed by a service provider, the message stops progressing toward the service's implementation object. Instead, it is sent back towards the binding for return to the consumer that originated the request.

2. Any message handlers that reside along the handler chain in the new processing direction have their `handleMessage()` method invoked in the order in which they reside in the chain.

3. When the message reaches the end of the handler chain it is dispatched.

For one-way message exchanges the following happens:

1. Message processing stops.

2. All previously invoked message handlers have their `close()` method invoked.

3. The message is dispatched.

III. Throw a `ProtocolException` exception—Throwing a `ProtocolException` exception, or a subclass of this exception, signals the Artix ESB runtime that fault message processing is beginning. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

For request-response message exchanges the following happens:

1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
2. The direction of message processing is reversed.

   For example, if a request is being processed by a service provider, the message stops progressing toward the service's implementation object. Instead, it is sent back towards the binding for return to the consumer that originated the request.
3. Any message handlers that reside along the handler chain in the new processing direction have their `handleFault()` method invoked in the order in which they reside in the chain.
4. When the fault message reaches the end of the handler chain it is dispatched.

For one-way message exchanges the following happens:

1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
2. Message processing stops.
3. All previously invoked message handlers have their `close()` method invoked.
4. The fault message is dispatched.

IV. Throw any other runtime exception—Throwing a runtime exception other than a `ProtocolException` exception signals the Artix ESB runtime that message processing is to stop. All previously invoked message handlers have the `close()` method invoked and the exception is dispatched. If the message is part of a request-response message exchange, the exception

is dispatched so that it is returned to the consumer that originated the request.

Example

Example 209 on page 349 shows an implementation of `handleMessage()` message for a logical message handler that is used by a service consumer. It processes requests before they are sent to the service provider.

Example 209. Logical Message Handler Message Processing

```
public class SmallNumberHandler implements LogicalHandler<LogicalMessageContext>
{
    public final boolean handleMessage(LogicalMessageContext messageContext)
    {
        try
        {
            boolean outbound = (Boolean)messageContext.get(MessageContext.MESSAGE_OUT
BOUND_PROPERTY);

            if (outbound) ❶
            {
                LogicalMessage msg = messageContext.getMessage(); ❷

                JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
                Object payload = msg.getPayload(jaxbContext); ❸
                if (payload instanceof JAXBElement)
                {
                    payload = ((JAXBElement)payload).getValue();
                }

                if (payload instanceof AddNumbers) ❹
                {
                    AddNumbers req = (AddNumbers)payload;

                    int a = req.getArg0();
                    int b = req.getArg1();
                    int answer = a + b;

                    if (answer < 20) ❺
                    {
                      AddNumbersResponse resp = new AddNumbersResponse(); ❻
                      resp.setReturn(answer);
                      msg.setPayload(new ObjectFactory().createAddNumbersResponse(resp),
                                                        jaxbContext);

                      return false; ❼
                    }
```

```
                }
                else
                {
                   throw new WebServiceException("Bad Request"); ❽
                }
            }
            return true; ❾
        }
        catch (JAXBException ex) ❿
        {
            throw new ProtocolException(ex);
        }
    }
...
}
```

The code in Example 209 on page 349 does the following:

❶ Checks if the message is an outbound request.

If the message is an outbound request, the handler does additional message processing.

❷ Gets the `LogicalMessage` representation of the message payload from the message context.

❸ Gets the actual message payload as a JAXB object.

❹ Checks to make sure the request is of the correct type.

If it is, the handler continues processing the message.

❺ Checks the value of the sum.

If it is less than the threshold of 20 then it builds a response and returns it to the client.

❻ Builds the response.

❼ Returns `false` to stop message processing and return the response to the client.

❽ Throws a runtime exception if the message is not of the correct type.

This exception is returned to the client.

❾ Returns `true` if the message is an inbound response or the sum does not meet the threshold.

Message processing continues normally.

❿ Throws a `ProtocolException` if a JAXB marshalling error is encountered.

The exception is passed back to the client after it is processed by the `handleFault()` method of the handlers between the current handler and the client.

Implementing a Protocol Handler

Overview

Protocol handlers are specific to the protocol in use. Artix ESB provides the SOAP protocol handler as specified by JAX-WS. A SOAP protocol handler implements the `javax.xml.ws.handler.soap.SOAPHandler` interface.

The `SOAPHandler` interface, shown in Example 210 on page 352, uses a SOAP specific message context that provides access to the message as a `SOAPMessage` object. It also allows you to access the SOAP headers.

Example 210. `SOAPHandler` Synopsis

```
public interface SOAPHandler extends Handler
{
  boolean handleMessage(SOAPMessageContext context);
  boolean handleFault(SOAPMessageContext context);
  void close(SOAPMessageContext context);
  Set<QName> getHeaders()
}
```

In addition to using a SOAP specific message context, SOAP protocol handlers require that you implement an additional method called `getHeaders()`. This additional method returns the QNames of the header blocks the handler can process.

Procedure

To implement a logical hander do the following:

1. Implement any initialization logic required by the handler.
2. Implement the message handling logic.
3. Implement the fault handling logic.
4. Implement the `getHeaders()` method.
5. Implement the logic for closing the handler when it is finished.

6. Implement any logic for cleaning up the handler's resources before it is destroyed.

Implementing the getHeaders() method

The `getHeaders()`, shown in Example 211 on page 353, method informs the Artix ESB runtime what SOAP headers the handler is responsible for processing. It returns the QNames of the outer element of each SOAP header the handler understands.

Example 211. The `SOAPHander.getHeaders()` Method

```
Set<QName> getHeaders();
```

For many cases simply returning `null` is sufficient. However, if the application uses the `mustUnderstand` attribute of any of the SOAP headers, then it is important to specify the headers understood by the application's SOAP handlers. The runtime checks the set of SOAP headers that all of the registered handlers understand against the list of headers with the `mustUnderstand` attribute set to `true`. If any of the flagged headers are not in the list of understood headers, the runtime rejects the message and throws a SOAP must understand exception.

Handling Messages in a SOAP Handler

Overview

Normal message processing is handled by the `handleMessage()` method.

The `handleMessage()` method receives a `SOAPMessageHandler` object that provides access to the message body as a `SOAPMessage` object and the SOAP headers associated with the message. In addition, the context provides access to any properties stored in the message context.

The `handleMessage()` method returns either `true` or `false` depending on how message processing is to continue. It can also throw an exception.

Working with the message body

You can get the SOAP message using the SOAP message context's `getMessage()` method. It returns the message as a live `SOAPMessage` object. Any changes to the message in the handler are automatically reflected in the message stored in the context.

If you wish to replace the existing message with a new one, you can use the context's `setMessage()` method. The `setMessage()` method takes a `SOAPMessage` object.

Getting the SOAP headers

You can access the SOAP message's headers using the `SOAPMessage` object's `getHeader()` method. This will return the SOAP header as a `SOAPHeader` object that you will need to inspect to find the header elements you wish to process.

The SOAP message context provides a `getHeaders()` method, shown in Example 212 on page 354, that will return an array containing JAXB objects for the specified SOAP headers.

Example 212. The `SOAPMessageContext.getHeaders()` Method

```
Ojbect[] getHeaders(QName header,
                    JAXBContext context,
                    boolean allRoles);
```

You specify the headers using the QName of their element. You can further limit the headers that are returned by setting the *`allRoles`* parameter to

`false`. That instructs the runtime to only return the SOAP headers that are applicable to the active SOAP roles.

If no headers are found, the method returns an empty array.

For more information about instantiating a `JAXBContext` object see *Using A JAXBContext Object* on page 267.

Working with context properties

The SOAP message context passed into a logical handler is an instance of the application's message context and can access all of the properties stored in it. Handlers have access to properties at both the `APPLICATION` scope and the `Handler` scope.

Like the application's message context, the SOAP message context is a subclass of Java `Map`. To access the properties stored in the context, you use the `get()` method and `put()` method inherited from the `Map` interface.

By default, any properties you set in the context from inside a logical handler will be assigned a scope of `HANDLER`. If you want the application code to be able to access the property you need to use the context's `setScope()` method to explicitly set the property's scope to `APPLICATION`.

For more information on working with properties in the message context see Understanding Contexts on page 314.

Determining the direction of the message

It is often important to know the direction a message is passing through the handler chain. For example, you would want to add headers to an outgoing message and strip headers from an incoming message.

The direction of the message is stored in the message context's outbound message property. You retrieve the outbound message property from the message context using the `MessageContext.MESSAGE_OUTBOUND_PROPERTY` key as shown in Example 213 on page 355.

Example 213. Getting the Message's Direction from the SOAP Message Context

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUT
BOUND_PROPERTY);
```

The property is stored as a `Boolean` object. You can use the object's `booleanValue()` method to determine the property's value. If the property is set to `true`, the message is outbound. If the property is set to `false` the message is inbound.

Determining the return value

How the `handleMessage()` method completes its message processing has a direct impact on how message processing proceeds. It can complete by doing one of the following actions:

I. return `true`—Returning `true` signals to the Artix ESB runtime that message processing should continue normally. The next handler, if any, has its `handleMessage()` invoked.

II. return `false`—Returning `false` signals to the Artix ESB runtime that normal message processing is to stop. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

For request-response message exchanges the following happens:

1. The direction of message processing is reversed.

   For example, if a request is being processed by a service provider, the message will stop progressing toward the service's implementation object. It will instead be sent back towards the binding for return to the consumer that originated the request.

2. Any message handlers that reside along the handler chain in the new processing direction have their `handleMessage()` method invoked in the order in which they reside in the chain.

3. When the message reaches the end of the handler chain it is dispatched.

For one-way message exchanges the following happens:

1. Message processing stops.

2. All previously invoked message handlers have their `close()` method invoked.

3. The message is dispatched.

III. throw a `ProtocolException` exception—Throwing a `ProtocolException` exception, or a subclass of this exception, signals the Artix ESB runtime that fault message processing is to start. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

For request-response message exchanges the following happens:

1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
2. The direction of message processing is reversed.

   For example, if a request is being processed by a service provider, the message will stop progressing toward the service's implementation object. It will be sent back towards the binding for return to the consumer that originated the request.
3. Any message handlers that reside along the handler chain in the new processing direction have their `handleFault()` method invoked in the order in which they reside in the chain.
4. When the fault message reaches the end of the handler chain it is dispatched.

For one-way message exchanges the following happens:

1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
2. Message processing stops.
3. All previously invoked message handlers have their `close()` method invoked.
4. The fault message is dispatched.

IV. throw any other runtime exception—Throwing a runtime exception other than a `ProtocolException` exception signals the Artix ESB runtime that message processing is to stop. All previously invoked message handlers have the `close()` method invoked and the exception is dispatched. If the

message is part of a request-response message exchange the exception is dispatched so that it is returned to the consumer that originated the request.

Example

Example 214 on page 358 shows a `handleMessage()` implementation that prints the SOAP message to the screen.

Example 214. Handling a Message in a SOAP Handler

```
public boolean handleMessage(SOAPMessageContext smc)
{
  PrintStream out;

  Boolean outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY); ❶

  if (outbound.booleanValue()) ❷
  {
    out.println("\nOutbound message:");
  }
  else
  {
    out.println("\nInbound message:");
  }

  SOAPMessage message = smc.getMessage(); ❸

  message.writeTo(out); ❹
  out.println();

  return true;
}
```

The code in Example 214 does the following:

❶ Retrieves the outbound property from the message context.

❷ Tests the messages direction and prints the appropriate message.

❸ Retrieves the SOAP message from the context.

❹ Prints the message to the console.

Initializing a Handler

Overview

When the runtime creates an instance of a handler, it creates all of the resources the hander needs to process messages. While you can place all of the logic for doing this in the handler's constructor, it may not be the most appropriate place. The handler framework performs a number of optional steps when it instantiates a handler. You can add resource injection and other initialization logic that will be executed during the optional steps.

Tip

You do not have to provide any initialization methods for a handler.

Order of initialization

The Artix ESB runtime initializes a handler in the following manner:

1. The handler's constructor is called.
2. Any resources that are specified by the `@Resource` annotation are injected.
3. The method decorated with `@PostConstruct` annotation, if it is present, is called.

   ## Note

   Methods decorated with the `@PostConstruct` annotation must have a void return type and have no parameters.

4. The handler is place in the `Ready` state.

Handling Fault Messages

Overview

Handlers use the `handleFault()` method for processing fault messages when a `ProtocolException` exception is thrown during message processing.

The `handleFault()` method receives either a `LogicalMessageContext` object or `SOAPMessageContext` object depending on the type of handler. The received context gives the handler's implementation access to the message payload.

The `handleFault()` method returns either `true` or `false`, depending on how fault message processing is to proceed. It can also throw an exception.

Getting the message payload

The context object received by the `handleFault()` method is similar to the one received by the `handleMessage()` method. You use the context's `getMessage()` method to access the message payload in the same way. The only difference is the payload contained in the context.

For more information on working with a `LogicalMessageContext` see Handling Messages in a Logical Handler on page 344.

For more information on working with a `SOAPMessageContext` see Handling Messages in a SOAP Handler on page 354.

Determining the return value

How the `handleFault()` method completes its message processing has a direct impact on how message processing proceeds. It completes by performing one of the following actions:

Return `true`

Returning `true` signals that fault processing should continue normally. The `handleFault()` method of the next handler in the chain will be invoked.

Return `false`

Returning `false` signals that fault processing stops. The `close()` method of the handlers that were invoked in processing the current message are invoked and the fault message is dispatched.

Throw an exception
: Throwing an exception stops fault message processing. The `close()` method of the handlers that were invoked in processing the current message are invoked and the exception is dispatched.

Example

Example 215 on page 361 shows an implementation of `handleFault()` that prints the message body to the screen.

Example 215. Handling a Fault in a Message Handler

```
public final boolean handleFault(LogicalMessageContext message
Context)
{
  System.out.println("handleFault() called with message:");

  LogicalMessage msg=messageContext.getMessage();
  System.out.println(msg.getPayload());

  return true;
}
```

Closing a Handler

When a handler chain is finished processing a message, the runtime calls each executed handler's `close()` method. This is the appropriate place to clean up any resources that were used by the handler during message processing or resetting any properties to a default state.

If a resource needs to persist beyond a single message exchange, you should not clean it up during in the handler's `close()` method.

Releasing a Handler

Overview

The runtime releases a handler when the service or service proxy to which the handler is bound is shutdown. The runtime will invoke an optional release method before invoking the handler's destructor. This optional release method can be used to release any resources used by the handler or perform other actions that would not be appropriate in the handler's destructor.

Tip

You do not have to provide any clean-up methods for a handler.

Order of release

The following happens when the handler is released:

1. The handler finishes processing any active messages.
2. The runtime invokes the method decorated with the `@PreDestroy` annotation.

   This method should clean up any resources used by the handler.
3. The handler's destructor is called.

Configuring Endpoints to Use Handlers

Programmatic Configuration

Important

Any handler chains configured using the Spring configuration override the handler chains configured programmaticaly.

Adding a Handler Chain to a Consumer

Overview

Adding a handler chain to a consumer involves explicitly building the chain of handlers. Then you set the handler chain directly on the service proxy's `Binding` object.

Procedure

To add a handler chain to a consumer you do the following:

1. Create a `List<Handler>` object to hold the handler chain.
2. Create an instance of each handler that will be added to the chain.
3. Add each of the instantiated handler objects to the list in the order they are to be invoked by the runtime.
4. Get the `Binding` object from the service proxy.

Tip

Artix ESB provides an implementation of the `Binding` interface called `org.apache.cxf.jaxws.binding.DefaultBindingImpl`.

5. Set the handler chain on the proxy using the `Binding` object's `setHandlerChain()` method.

Example

Example 216 on page 366 shows code for adding a handler chain to a consumer.

Example 216. Adding a Handler Chain to a Consumer

```
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.Handler;
import java.util.ArrayList;
import java.util.List;

import org.apache.cxf.jaxws.binding.DefaultBindingImpl;
...
SmallNumberHandler sh = new SmallNumberHandler(); ❶
List<Handler> handlerChain = new ArrayList<Handler>(); ❷
handlerChain.add(sh); ❸

DefaultBindingImpl binding = ((BindingProvider)proxy).getBinding(); ❹
binding.getBinding().setHandlerChain(handlerChain); ❺
```

The code in Example 216 on page 366 does the following:

❶ Instantiates a handler.

❷ Creates a `List` object to hold the chain.

❸ Adds the handler to the chain.

❹ Gets the `Binding` object from the proxy as a `DefaultBindingImpl` object.

❺ Assigns the handler chain to the proxy's binding.

Adding a Handler Chain to a Service Provider

Overview

You add a handler chain to a service provider by decorating either the SEI or the implementation class with the `@HandlerChain` annotation. The annotation points to a meta-data file defining the handler chain used by the service provider.

Procedure

To add handler chain to a service provider you do the following:

1. Decorate the provider's implementation class with the `@HandlerChain` annotation.

2. Create a handler configuration file that defines the handler chain.

The @HandlerChain annotation

The `javax.jws.HandlerChain` annotation decorates service provider's implementation class. It instructs the runtime to load the handler chain configuration file specified by its file property.

The annotation's file property supports two methods for identifying the handler configuration file to load:

- a URL
- a relative path name

Example 217 on page 367 shows a service provider implementation that will use the handler chain defined in a file called `handlers.xml`. `handlers.xml` must be located in the directory from which the service provider is run.

Example 217. Service Implementation that Loads a Handler Chain

```
import javax.jws.HandlerChain;
import javax.jws.WebService;
...

@WebService(name = "AddNumbers",
            targetNamespace = "http://apache.org/handlers",
            portName = "AddNumbersPort",
            endpointInterface = "org.apache.handlers.AddNumbers",
            serviceName = "AddNumbersService")
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl implements AddNumbers
{
...
}
```

Handler configuration file

The handler configuration file defines a handler chain using the XML grammar that accompanies JSR 109(Web Services for Java EE, Version 1.2). This grammar is defined in the http://java.sun.com/xml/ns/javaee.

The root element of the handler configuration file is the `handler-chains` element. The `handler-chains` element has one or more `handler-chain` elements.

The `handler-chain` element define a handler chain. Table 31 on page 368 describes the `handler-chain` element's children.

Table 31. Elements Used to Define a Server-Side Handler Chain

Element	Description
`handler`	Contains the elements that describe a handler.
`service-name-pattern`	Specifies the QName of the WSDL `service` element defining the service to which the handler chain is bound. You can use * as a wildcard when defining the QName.
`port-name-pattern`	Specifies the QName of the WSDL `port` element defining the endpoint to which the handler chain is bound. You can use * as a wildcard when defining the QName.
`protocol-binding`	Specifies the message binding for which the handler chain is used. The binding is specified as a URI or using one of the following aliases: `##SOAP11_HTTP`, `##SOAP11_HTTP_MTOM`, `##SOAP12_HTTP`, `##SOAP12_HTTP_MTOM`, or `##XML_HTTP`. For more information about message binding URIs see Appendix A in the *Artix® ESB Deployment Guide*.

The `handler-chain` element is only required to have a single `handler` element as a child. It can, however, support as many `handler` elements as needed to define the complete handler chain. The handlers in the chain are executed in the order they specified in the handler chain definition.

Important

The final order of execution will be determined by sorting the specified handlers into logical handlers and protocol handlers. Within the groupings, the order specified in the configuration will be used.

The other children, such as `protocol-binding`, are used to limit the scope of the defined handler chain. For example, if you use the `service-name-pattern` element, the handler chain will only be attached to service providers whose WSDL `port` element is a child of the specified WSDL `service` element. You can only use one of these limiting children in a `handler` element.

The `handler` element defines an individual handler in a handler chain. Its `handler-class` child element specifies the fully qualified name of the class implementing the handler. The `handler` element can also have an optional `handler-name` element that specifies a unique name for the handler.

Example 218 on page 369 shows a handler configuration file that defines a single handler chain. The chain is made up of two handlers.

Example 218. Handler Configuration File

```
<handler-chains 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">
  <handler-chain>
    <handler>
      <handler-name>LoggingHandler</handler-name>
      <handler-class>demo.handlers.common.LoggingHandler</handler-class>
    </handler>
    <handler>
      <handler-name>AddHeaderHandler</handler-name>
      <handler-class>demo.handlers.common.AddHeaderHandler</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

Spring Configuration

Overview

The easiest way to configure an endpoint to use a handler chain is to define the chain in the endpoint's configuration. This is done by adding a `jaxwxs:handlers` child to the element configuring the endpoint.

Important

A handler chain added through the configuration file takes precedence over a handler chain configured programatically.

Procedure

To configure an endpoint to load a handler chain you do the following:

1. If the endpoint does not already have a configuration element, add one.

   For more information on configuring Artix ESB endpoints see *Configuring Artix ESB Endpoints* in the *Artix® ESB Deployment Guide*.

2. Add a `jaxws:handlers` child element to the endpoint's configuration element.

3. For each handler in the chain, add a `bean` element specifying the class that implements the handler.

   ### Tip

   If your handler implementation is used in more than one place you can reference a `bean` element using the `ref` element.

The handlers element

The `jaxws:handlers` element defines a handler chain in an endpoint's configuration. It can appear as a child to all of the JAX-WS endpoint configuration elements. These are:

- `jaxws:endpoint` configures a service provider.
- `jaxws:server` also configures a service provider.
- `jaxws:client` configures a service consumer.

You add handlers to the handler chain in one of two ways:

- add a `bean` element defining the implementation class
- use a `ref` element to refer to a named `bean` element from elsewhere in the configuration file

The order in which the handlers are defined in the configuration is the order in which they will be executed. The order may be modified if you mix logical handlers and protocol handlers. The run time will sort them into the proper order while maintaining the basic order specified in the configuration.

Example

Example 219 on page 371 shows the configuration for a service provider that loads a handler chain.

Example 219. Configuring an Endpoint to Use a Handler Chain In Spring

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="HandlerExample"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo">
    <jaxws:handlers>
       <bean class="demo.handlers.common.LoggingHandler" />
       <bean class="demo.handlers.common.AddHeaderHandler" />
    </jaxws:handlers>
  </jaws:endpoint>
</beans>
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W