IONA®

Artix ESB

Developing Artix Applications with JAX-WS

Version 5.0
July 2007

Making Software Work Together™

Developing Artix Applications with JAX-WS

IONA Technologies

Version 5.0

Published 04 Oct 2007

Trademark and Disclaimer Notice

Copyright Notice

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Table of Contents

What is Covered in This Book

This book describes how to use the JAX-WS 2.0 APIs to develop applications with Artix ESB.

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:

- *Starting from Java Code* describes how to develop SOA applications with out using WSDL documents.
- *Starting from a WSDL Contract* describes how to develop SOA applications using a WSDL document as a starting point.
- *Publishing a Service* describes how to publish a service using a stand alone Java application.
- *Developing Asynchronous Applications* describes how to develop service consumers that can interact with service providers asynchronously.
- *Using Raw XML Messages* describes how to use the `Dispatch` and `Provider` interfaces to develop applications that work with raw XML instead of JAXB object.
- *Working with Contexts* describes how to manipulate message and transport properties programaticaly.
- *Developing RESTful Services* describes how to use the Artix ESB API's annotations to create RESTful services.

Part I. Basic Programming Tasks

Summary

The JAX-WS programming model makes it easy to develop service providers and consumers. You can either start directly with Java code, or you can start from WSDL documents. This part guides you through the steps for creating and publishing endpoints. It also inclludes a chapter on developing services that follow REST principles.

Table of Contents

Starting from Java Code

Summary

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.

Table of Contents

Service Enabling a Java Class

Table of Contents

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

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

   ### Tip

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

2. Add the required annotations to your code.

3. Generate the WSDL contract for your service.

   ### Tip

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

4. Publish the service as a service provider.

Creating the SEI

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 it. It 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 up to a developer to create the SEI.

There are two basic patterns for creating an SEI:

- Green field development

  You are developing a new service from the ground up. When starting fresh, it is best to start by creating the SEI first. You can then distribute the SEI to any developers that are responsible for implementing the 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 *Starting from a WSDL Contract*.

- Service enablement

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

  1. Create an SEI that contains **only** the operations that are going to be exposed as part of the service.

  2. Modify the existing Java class so that it implements the SEI.

Note

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

Writing the interface

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

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

Tip

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

Example 1, “Simple SEI” 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 just a standard Java class. If you started with a Java class you will need to modify it to implement the interface. If you are starting fresh, the implementation class will need to implement the SEI.

Example 2, “Simple Implementation Class” shows a class for implementing the interface in Example 1, “Simple SEI”.

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

Table of Contents

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

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

Tip

Most of the annotations have sensible defaults and do not need to be specified. However, the more information you provide in the annotations, the better defined your service definition. A solid service definition increases the 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 following properties:

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 under which the service is defined. If this property is not specified, the target namespace is derived from the package name.
serviceName	Specifies the name of the published service. This property is mapped to the `name` attribute of the `wsdl:service` element that defines the published service. The default is to use the name of the service's implementation class. [a]
wsdlLocation	Specifies the URI at which the service's WSDL contract is stored. The default is the URI at which the service is deployed.
endpointInterface	Specifies the full name of the SEI that the implementation class implements. This property is only used when the attribute is used on a service implementation class.
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

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

Annotating the SEI

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

Example 3, “Interface with the `@WebService` Annotation” shows the interface defined in Example 1, “Simple SEI” 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, “Interface with the `@WebService` Annotation” 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 have to annotate the service implementation class with the `@WebService` annotation. When adding the annotation to the service implementation class you only need to specify the endpointInterface property. As shown in Example 4, “Annotated Service Implementation Class” the property needs to 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 provide a lot of information about 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 will be for developers to implement applications that can use the functionality it defines. It will also provide for better generated WSDL contracts.

Defining the Binding Properties with Annotations

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

The @SOAPBinding annotation

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

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

Table 2, "`@SOAPBinding` Properties" 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 will appear inside a wrapper element within the `soap:body` element. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. If `DOCUMENT` style is specified, the contents of the SOAP body must be a valid XML document, but its form is not as tightly constrained.
use	`Use.LITERAL` (default) `Use.ENCODED`	Specifies how the data of the SOAP message is streamed.
parameterStyle[a]	`ParameterStyle.BARE` `ParameterStyle.WRAPPED` (default)	Specifies how the method parameters, which correspond to message parts in a WSDL contract, are placed into the SOAP message body. A parameter style of `BARE` means that each parameter is placed into the message body as a child element of the message root. A parameter style of `WRAPPED` means that all of the input parameters are wrapped into a single element on a request message and that all of the output parameters are wrapped into a single element in the response message.

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

Example 5, "Specifying an RPC/LITERAL SOAP Binding with the `@SOAPBinding` Annotation" shows an SEI that uses rpc/literal SOAP messages.

Example 5. Specifying an RPC/LITERAL SOAP Binding with the `@SOAPBinding` Annotation

```
package org.eric.demo;

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

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

Defining Operation Properties with Annotations

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

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

The @WebMethod annotation

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

Table 3, “`@WebMethod` Properties” 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. As the name implies, `@RequestWrapper` specifies the Java class that implements the wrapper bean for the method parameters that are included in the request message sent in a remote invocation. It is also used to specify the element names, and namespaces, used by the runtime when marshalling and unmarshalling the request messages.

Table 4, "`@RequestWrapper` Properties" 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 the name of the method or the value of the `@WebMethod` annotation's operationName property.
targetNamespace	Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.
className	Specifies the full name of the Java class that implements the wrapper element.

> **Tip**
>
> Only the className property is required.

The @ResponseWrapper annotation

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

Table 5, “`@ResponseWrapper` Properties” 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 the name of the method with `Response` appended or the value of the `@WebMethod` annotation's operationName property with `Response` appended.
targetNamespace	Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.
className	Specifies the full name of the Java class that implements the wrapper element.

> **Tip**
>
> Only the className property is required.

The @WebFault annotation

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

Table 6, "`@WebFault` Properties" 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 not reserving any resources to process a response.

Example

Example 6, "SEI with Annotated Methods" shows an SEI whose methods are 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 coresspond to the `wsdl:message` elements and their `wsdl:part` elements. JAX-WS provides annotations that allow you to describe the `wsdl:part` elements that are generated for the method parameters.

The @WebParam annotation

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

Table 7, “`@WebParam` Properties” 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 WSDL. For RPC bindings, this is name of the `wsdl:part` representing the parameter. For document bindings, this is the local name of the XML element representing the parameter. Per the JAX-WS specification, the default is `arg`*N*, where *N* is replaced with the zero-based argument index (i.e., arg0, arg1, etc.).
targetNamespace		Specifies the namespace for the parameter. It is only used with document bindings where the parameter maps to an XML element. The defaults is to use the service's namespace.
mode	`Mode.IN` (default) `Mode.OUT`	Specifies the direction of the parameter.

Property	Values	Description
	`Mode.INOUT`	
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 when the binding is document.

The @WebResult annotation

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

Table 8, "`@WebResult` Properties" 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 WSDL. For RPC bindings, this is name of the `wsdl:part` representing the return value. For document bindings, this is the local name of the XML element representing the return value. The default value is `return`.
targetNamespace	Specifies the namespace for the return value. It is only used with document bindings where the return value maps to an XML element. The defaults is to use the service's namespace.
header	Specifies if the return value is passed as part of the SOAP header.

Property	Description
partName	Specifies the value of the `name` attribute of the `wsdl:part` element for the return value when the binding is document.

Example

Example 7, “Fully Annotated SEI” 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 command line tools

Once you have annotated your code, you can generate a WSDL contract for your service using the **artix java2wsdl** command. For a detailed listing of options for the **artix java2wsdl** command see artix java2wsdl in *Artix ESB Command Reference*.

Using Artix Designer

Artix Designer automatically generates WSDL as you edit your Java code.

Example

Example 8, “Generated WSDL from an SEI” shows the WSDL contract generated for the SEI shown in Example 7, “Fully Annotated SEI”.

Example 8. 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

Table of Contents

To create a consumer without a WSDL contract you need to 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

The `javax.xml.ws.Service` class represents the `wsdl:service` element that 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 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 9, “`Service create()` Methods”, 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 there WSDL contracts. For SOAP/HTTP services the URI is usually the URI at which the service is published appended with `?wsdl`.

Example 9. `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 a number of 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 10, "Creating a `Service` Object" shows code for creating a `Service` object for the SEI shown in Example 7, "Fully Annotated SEI".

Example 10. 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 10, "Creating a `Service` Object" does the following:

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

❷ Call 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 an 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 will create 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 cannot create any proxy instances. In this case, you must provide the `Service` object with the information that would be in a `wsdl:port` element using the `addPort()` method.

The addPort() method

The `Service` class defines an `addPort()` method, shown in Example 11, “The `addPort()` Method”, 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, needed to create a proxy for a service implementation.

Example 11. 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 a number of ways:

1. It is specified in the portName property of the `@WebService` annotation.
2. You append `Port` to the value of the name property of the `@WebService` annotation.
3. You 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 would 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 will be determined by the binding developer.

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

Example

Example 12, "Adding a Port to a `Service` Object" shows code for adding a port to the `Service` object created in Example 10, "Creating a `Service` Object".

Example 12. Adding a Port to 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 portName = new QName("http://demo.iona.com", "stockQuoteReporterPort");
❷    s.addPort(portName,
❸              "http://schemas.xmlsoap.org/soap/",
❹              "http://localhost:9000/StockQuote");
    ...
  }
}
```

The code in Example 12, "Adding a Port to a `Service` Object" 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 at which 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 of which it is aware 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 13, “The `getPort()` Method”, returns a service proxy for the specified endpoint. The returned proxy is of the same class as the SEI.

Example 13. 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 class 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 14, “Getting a Service Proxy” shows code for getting a service proxy for the endpoint added in Example 12, “Adding a Port to a `Service` Object”.

Example 14. Getting a Service Proxy

```
package com.iona.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

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

Note

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

Example

Example 15, “Consumer Implemented without a WSDL Contract” shows a consumer for the service defined in Example 7, “Fully Annotated SEI”.

Example 15. Consumer Implemented without a WSDL Contract

```
package com.iona.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/EricStockQuote");

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 15, “Consumer Implemented without a WSDL Contract” 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.

Starting from a WSDL Contract

Summary

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.

Table of Contents

A WSDL Contract

Example 16, “HelloWorld WSDL Contract” 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 16. 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">
        <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 16, “HelloWorld WSDL Contract” 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.

Developing a Service Starting from a WSDL Contract

Table of Contents

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

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.

Artix Designer provides a wizard for generating starting point code from a WSDL document. This wizard provides you with options for control the code generation.

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

Using Artix Designer

When starting a WSDL first project by importing a WSDL document, Artix Designer asks you what code to generate. If you create the WSDl document using Artix Designer or need to regenerate the JAX-WS code, you used the Artix → Generate Code from any WSDL document's context menu.

Using the command line tools

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

```
artix wsdl2java -ant -impl -server -d outputDirmyService.wsdl
```

The command does the following:

- The `-ant` argument generates a Ant makefile, called `build.xml`, for your application.
- The `-impl` argument generates a shell implementation class for each `wsdl:portType` element in the WSDL contract.
- The `-server` argument generates a simple `main()` to launch your service provider as a stand alone application.

- The `-d` *`outputDir`* argument tells **wsdl2java** to write the generated code to a directory called *`outputDir`*.
- *`myService.wsdl`* is the WSDL contract from which code is generated.

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

Generated code

Table 9, “Generated Classes for a Service Provider” 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 will use to make requests on the service.
`portTypeName``Impl.java`	The skeleton implementation class. You will 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*.

In addition, the tools 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`, 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`, 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

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

Generating the implementation code

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

Tip

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

Generated code

The implementation code consists of two files:

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

Implement the operation's logic

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

Example

For example, an implementation class for the service defined in Example 16, “HelloWorld WSDL Contract” may look like Example 17, “Implementation of the Greeter Service”. Only the code portions highlighted in bold must be inserted by the programmer.

Example 17. 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 Starting from a WSDL Contract

Table of Contents

Generating the Stub Code

You use Artix ESB's code generation tools to generate the stub code from the WSDL document. The stub code provides the supporting code that is required to invoke operations on the remote service.

For consumers, the code generation tools can generate the following kinds 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.

Using Artix Designer

When starting a WSDL first project by importing a WSDL document, Artix Designer asks you what code to generate. If you create the WSDI document using Artix Designer or need to regenerate the JAX-WS code, you used the Artix → Generate Code from any WSDL document's context menu.

Using the command line tools

You generate consumer code using 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 you would like to put the generated files and `hello_world.wsdl` is a file containing the contract shown in Example 16, “HelloWorld WSDL Contract”. 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 **artix wsdl2java** see artix wsdl2java in *Artix ESB Command Reference*.

Generated code

The preceding command generates the following Java packages:

- `org.apache.hello_world_soap_http`

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

- `org.apache.hello_world_soap_http.types`

  This package name is generated from the `http://apache.org/hello_world_soap_http/types` target namespace. All of the XML types defined in this target namespace (that is, everything defined in the `wsdl:types` element of the HelloWorld contract) map to Java classes in the corresponding Java package.

The stub files generated by tools fall into the following categories:

- Classes representing WSDL entities (in the `org.apache.hello_world_soap_http` package) — the following classes are generated to represent WSDL entities:
  - `Greeter` is a Java interface that represents the Greeter `wsdl:portType` element. In JAX-WS terminology, this Java interface is the *service endpoint interface* (SEI).
  - `SOAPService` is a Java service class (extending `javax.xml.ws.Service`) that represents the SOAPService `wsdl:service` element.
  - `PingMeFault` is a Java exception class (extending `java.lang.Exception`) that represents the pingMeFault `wsdl:fault` element.
- Classes representing XML types (in the `org.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

This section describes how to write the code for a simple Java client, based on the WSDL contract from Example 16, “HelloWorld WSDL Contract”. To implement the consumer, you need to use the following stubs:

- Service class (`SOAPService`).
- SEI (`Greeter`).

Generated service class

Example 18, “Outline of a Generated Service Class” shows the typical outline of a generated service class, *`ServiceName`*`_Service`[1], which extends the `javax.xml.ws.Service` base class.

Example 18. 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="SoapPort")
  public Greeter getPortName() { }
  .
  .
  .
}
```

The `ServiceName` class in Example 18, “Outline of a Generated Service Class” defines the following methods:

- Constructor methods — the following forms of constructor are defined:

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

  - `ServiceName(URL wsdlLocation, QName serviceName)` constructs a service object based on the data in the `ServiceName` service in the WSDL contract that is obtainable from `wsdlLocation`.

  - `ServiceName()` is the default constructor, which constructs a service object based on the service name and WSDL contract that were provided at the time the stub code was generated (for example, when running **wsdl2java**). Using this constructor presupposes that the WSDL contract remains available at its original location.

- `getPortName()` methods — for every `PortName` port defined on the `ServiceName` service, **wsdl2java** generates a corresponding `getPortName()` method in Java. Therefore, a `wsdl:service` element that defines multiple endpoints will generate a service class with multiple `getPortName()` methods.

Service endpoint interface

For every port type 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 19, “The Greeter Service Endpoint Interface” shows the Greeter SEI, which is generated from the `wsdl:portType` element defined in Example 16, “HelloWorld WSDL Contract”. For simplicity, Example 19, “The Greeter Service Endpoint Interface” omits the standard JAXB and JAX-WS annotations.

Example 19. The Greeter Service Endpoint Interface

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

```
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 20, "Consumer Implementation Code" shows the generated 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 20. 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 20, “Consumer Implementation Code” proceeds as follows:

❶ The runtime is implicitly initialized — that is, provided the Artix ESB runtime classes are loaded. Hence, there is no need to call a special function in order 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 (passing in the WSDL contract's location and service name).

❹ Call the appropriate `get`*`PortName`*`()` method to obtain an instance of the particular port you need. In this case, the SOAPService service supports only the SoapPort port, which is of Greeter type.

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.

Publishing a Service

Summary

When you want to deploy a JAX-WS service as a standalone Java application, you need to write a server mainline. This mainline publishes an endpoint for your service.

Table of Contents

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. If you are deploying your service into one of the containers supported by Artix ESB you do not need to write any additional code. However, if you are going to deploy your service as a stand-alone Java application, you will need to write a `main()` that publishes the service as a self-contained service provider.

Generating a Server Mainline

The **wsdl2java** tool's `-server` flag causes the tool to generate a simple server mainline. The generated server mainline, as shown in Example 21, “Generated Server Mainline”, publishes one service provider for each `port` defined in the WSDL contract.

Example

Example 21, “Generated Server Mainline” shows a generated server mainline.

Example 21. 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 21, “Generated Server Mainline” 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()`.

Instantiating an service provider

You can instantiate an `Endpoint` using one of the following three methods provided by `Endpoint`:

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

  This `create()` method returns an `Endpoint` for the specified service implementation. The created `Endpoint` 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` will use a default SOAP 1.1/HTTP binding.

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

  This `create()` method returns an `Endpoint` 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 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` for the specified implementation and publishes it. The binding used for the `Endpoint` 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` 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 to 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 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 one 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 it also must be compatible with one of the service provider's available bindings.

Example

Example 22, “Custom Server Mainline” shows code for publishing a service provider.

Example 22. 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)
    {
      ...
    }

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

The code in Example 22, “Custom Server Mainline” 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`.

❹ Loop until the server should be shutdown.

Developing RESTful Services

Summary

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.

Table of Contents

Introduction to RESTful Services

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 23, “Invalid REST Request”.

Example 23. Invalid REST Request

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

Example 23, “Invalid REST Request” 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 24, “Wrapped REST Request”.

Example 24. 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)
- Java REST annotations (see Using Java REST Annotations)

Using Automatic REST Mappings

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.

Typical CRUD class

Example 25, “Widget Catalog CRUD Class” shows a CRUD based class for updating a catalog of widgets.

Example 25. 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

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. 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 26, “URI Template Syntax”.

Example 26. 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 27, “Using a URI Template” and wanted to access the record at id 42, you would perform a `GET` at `http://myexample.iona.com/records/42`.

Example 27. 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 25, “Widget Catalog CRUD Class” you may use an SEI like the one shown in Example 28, “SEI for a Widget Ordering Service”.

Example 28. 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 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 29, “`WidgetOrdering` with REST Annotations” shows `WidgetOrdering` with the required annotations.

Example 29. `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

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 30, "Setting a Server Factory's Service Class".

   Example 30. 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 31, "Setting Wrapped Mode".

   Example 31. Setting Wrapped Mode

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

   Note

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

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 32, "Selecting the REST Binding".

Example 32. 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 33, "Setting the Base URI".

Example 33. 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 34, "Setting the Service Invoker".

Example 34. Setting the Service Invoker

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

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

Example

Example 35, "Publishing the WidgetCatalog Service as a RESTful Endpoint" 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 35. 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 35, “Publishing the WidgetCatalog Service as a RESTful Endpoint” to publish the service defined by Example 25, “Widget Catalog CRUD Class”, 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 II. Advanced Programming Tasks

Summary

The JAX-WS programming model offers a number of advanced features.

Table of Contents

Developing Asynchronous Applications

Summary

JAX-WS provides an easy mechanism for accessing services asynchronously. The SEI can specify additional methods that a can use to access a service asynchronously. The Artix ESB code generators will generate the extra methods for you. You simply need to add the business logic.

Table of Contents

In addition to the usual synchronous mode of invocation, Artix ESB also supports two forms of asynchronous invocation:

- Polling approach

  In this case, to invoke the remote operation, you call a special method that has no output parameters, but returns a `javax.xml.ws.Response` 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

  In this case, to invoke the remote operation, you call another special method that takes a reference to a callback object (of `javax.xml.ws.AsyncHandler` type) as one of its parameters. Whenever the response message arrives at the client, the runtime calls back on the `AsyncHandler` object to give it the contents of the response message.

WSDL for Asynchronous Examples

Example 36, “WSDL Contract for Asynchronous Example” 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 36. WSDL Contract for Asynchronous Example

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

  <wsdl:message name="greetMeSometimeRequest">
    <wsdl:part name="in" element="x1:greetMeSometime"/>
  </wsdl:message>
  <wsdl:message name="greetMeSometimeResponse">
    <wsdl:part name="out"
               element="x1:greetMeSometimeResponse"/>
```

```
  </wsdl:message>

  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="GreeterAsync_SOAPBinding"
                type="tns:GreeterAsync">
    ...
  </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

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, however. To switch on the asynchronous feature and generate the requisite stub code, you must use the mapping customization feature from the WSDL 2.0 specification.

Defining the customization

Customization enables you to modify the way the **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 alternative ways of specifying a binding declaration:

- External binding declaration — the `jaxws:bindings` element is defined in a file separately from the WSDL contract. You specify the location of the binding declaration file to **wsdl2java** when you generate the stub code.
- Embedded binding declaration — you can also embed the `jaxws:bindings` element directly in a WSDL contract, treating it as a WSDL extension. In this case, the settings in `jaxws:bindings` apply only to the immediate parent element.

This section considers only the external binding declaration. The template for a binding declaration file that switches on asynchronous invocations is shown in Example 37, “Template for an Asynchronous Binding Declaration”.

Example 37. 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 could specify <bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']"> in the preceding binding declaration.

Running wsdl2java

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

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

When you run **wsdl2java**, you specify the location of the binding declaration file using the `-b` option.

Generated code

After generating the stub code in this way, the `GreeterAsync` SEI (in the file `GreeterAsync.java`) is defined as shown in Example 38, “Service Endpoint Interface with Methods for Asynchronous Invocations”.

Example 38. Service Endpoint Interface with Methods for Asynchronous Invocations

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

- `public Future<?> greetMeSomtimeAsync(java.lang.String requestType,`
  `AsyncHandler<GreetMeSomtimeResp`

  Call this method for the callback approach to asynchronous invocation.

- `public Response<GreetMeSomeTimeResponse> greetMeSometimeAsync(java.l`

  Call this method for the polling approach to asynchronous invocation.

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 can poll for a response. What the client does while it is waiting for a response is up to the requirements of the application. There are two basic patterns for how to handle 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 can process it. If it not, the client can continue 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 39, "Non-Blocking Polling Approach for an Asynchronous Operation Call" illustrates using non-blocking polling to make an asynchronous invocation on the greetMeSometime operation defined in Example 36, "WSDL Contract for Asynchronous Example". The client invokes the asynchronous operation and periodically checks to see if the result has returned.

Example 39. 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 39, “Non-Blocking Polling Approach for an Asynchronous Operation Call” 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 under the covers. The endpoint, and therefore the service implementation, never needs to worry 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 does some work before checking again.

❸ If the response has arrived, the client retrieves it from the `Response` object using the `get()`.

Using the blocking pattern

Using blocking polling to make asynchronous invocations on a remote operation follows the same steps as non-blocking polling. However, instead of using the `Response` object's `isDone()` to check if a response has been returned before calling the `get()` to retrieve the response, you immediately call the `get()`. The `get()` blocks until the response is available.

Tip

You can also pass a timeout limit to the `get()` method.

Example 40, “Blocking Polling Approach for an Asynchronous Operation Call” shows a client that uses blocking polling.

Example 40. 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 you need to 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 needs to access the response data, you can periodically use 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, you do not need to check if the response has arrived.

Implementing the callback

Your 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() to notify the client that the response has arrived. Example 41, "The `javax.xml.ws.AsyncHandler` Interface” shows an outline of the `AsyncHandler` interface that you need to implement.

Example 41. The `javax.xml.ws.AsyncHandler` Interface

```
public interface javax.xml.ws.AsyncHandler
{
  void handleResponse(Response<T> res)
}
```

Example 42, “Callback Implementation Class” shows a callback class for the greetMeSometime operation defined in Example 36, “WSDL Contract for Asynchronous Example”.

Example 42. 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 42, “Callback Implementation Class” does the following:

❶ Defines a member variable, `response`, to hold 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 43, “Callback Approach for an Asynchronous Operation Call” illustrates a client that uses the callback approach to make an asynchronous call to the GreetMeSometime operation defined in Example 36, “WSDL Contract for Asynchronous Example”.

Example 43. 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 43, “Callback Approach for an Asynchronous Operation Call” 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, never needs 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.

Using Raw XML Messages

Summary

The high-level JAX-WS APIs shield the developer from using native XML messages by marshelling 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: `Dispatch` is the client-side interface.

`Provider` is the server-side interface.

Table of Contents

Using XML in a Consumer with the `Dispatch` Interface

Table of Contents

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, `Dispatch` 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 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 would need to provide the `Dispatch` object's `invoke()` method a fully specified SOAP message. The `invoke()` method will also return a fully specified SOAP message. The consumer code is responsible for completing and reading the SOAP message's headers and the SOAP message's envelope information.

Tip

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

You specify that a `Dispatch` object uses message mode by providing the value `java.xml.ws.Service.Mode.MESSAGE` when creating the `Dispatch` object. For more information about creating a `Dispatch` object see Creating a `Dispatch` object.

Payload mode

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

Tip

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

You specify that a `Dispatch` object uses payload mode by providing the value `java.xml.ws.Service.Mode.PAYLOAD` when creating the `Dispatch` object. For more information about creating a `Dispatch` object see Creating a `Dispatch` object.

Data Types

Overview

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

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

Using Source objects

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

`DOMSource`

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of `Node` objects that can be accessed using the `getNode()` method. Nodes can be updated or added to the DOM tree using the `setNode()` method.

`SAXSource`

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an `InputSource` object that contains the raw data and an `XMLReader` object that parses the raw data.

`StreamSource`

Holds XML messages as a data stream. The data stream can be manipulated as would any other data stream.

Important

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

Using SOAPMessage objects

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

Using DataSource objects

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

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

Working with `Dispatch` Objects

Procedure

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

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

Creating a Dispatch object

To create a `Dispatch` object do the following:

1. Create a `Service` object to represent the `wsdl:service` element defining the service on which the `Dispatch` object will make invocations. See Creating a Service Object.
2. Create the `Dispatch` object using the `Service` object's `createDispatch()` method shown in Example 44, “The `createDispatch()` Method”.

Example 44. 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 10, "Parameters for `createDispatch()`" describes the parameters for `createDispatch()`.

Table 10. Parameters for `createDispatch()`

Parameter	Description
portName	Specifies the QName of the `wsdl:port` element that represent the service provider on which the `Dispatch` object will make invocations.
type	Specifies the data type of the objects used by the `Dispatch` object. See Data Types. **Note** If you are working with JAXB objects, this parameter is where you would specify the `JAXBContext` object used to marshal and unmarshal the JAXB objects.
mode	Specifies the usage mode for the `Dispatch` object. See Usage Modes.

Example 45, "Creating a `Dispatch` Object" shows code for creating a `Dispatch` object that works with `DOMSource` objects in payload mode.

Example 45. 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 = 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 XMLSchema document that defines the messages. While service providers vary greatly there are a few guidelines that can be followed:

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

Warning

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

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

Important

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

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

For more information about how services use XML messages see the WS-I Basic Profile [http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html].

Synchronous invocation

For consumers that make synchronous invocations that generate a response, you use the `Dispatch` object's `invoke()` method shown in Example 46, “The `Dispatch.invoke()` Method”.

Example 46. 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 created a `Dispatch` object using `createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE)` the response and the request would both be `SOAPMessage` objects.

Note

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

Example 47, “Making a Synchronous Invocation Using a `Dispatch` Object” shows code for making a synchronous invocation on a remote service using a `DOMSource` object.

Example 47. 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);
```

```
// 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*, `Dispatch` objects can use both the polling approach and the callback approach.

When using the polling approach the `invokeAsync()` method returns a `Response<t>` object that can be periodically polled to see if the response has arrived. Example 48, “The `Dispatch.invokeAsync()` Method for Polling” shows the signature of the method used to make an asynchronous invocation using the polling approach.

Example 48. 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.

When using the callback approach the `invokeAsync()` method takes an `AsyncHandler` implementation that processes the response when it is returned. Example 49, “The `Dispatch.invokeAsync()` Method Using a Callback” shows the signature of the method used to make an asynchronous invocation using the callback approach.

Example 49. 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.

Note

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

Oneway invocation

When a request does not generate a response, you make remote invocations using the `Dispatch` object's `invokeOneWay()`. Example 50, “The `Dispatch.invokeOneWay()` Method” shows the signature for this method.

Example 50. 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)` the request would be packaged into a `DOMSource` object.

Note

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

Example 51, “Making a One Way Invocation Using a `Dispatch` Object” shows code for making a oneway invocation on a remote service using a JAXB object.

Example 51. 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 with the `Provider` Interface

Table of Contents

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 as they are with the higher level SEI based objects.

Messaging Modes

Overview

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

- Message mode
- Payload mode

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

Message mode

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

You specify that a `Provider` implementation uses message mode by providing the value `java.xml.ws.Service.Mode.MESSAGE` as the value to the `javax.xml.ws.ServiceMode` annotation as shown in Example 52, "Specifying that a `Provider` Implementation Uses Message Mode".

Example 52. Specifying that a `Provider` Implementation Uses Message Mode

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

Payload mode

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

Tip

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

You specify that a `Provider` implementation uses payload mode by providing the value `java.xml.ws.Service.Mode.PAYLOAD` as the value to the `javax.xml.ws.ServiceMode` annotation as shown in Example 53, “Specifying that a `Provider` Implementation Uses Payload Mode”.

Example 53. 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 the `@ServiceMode` annotation, the `Provider` implementation will default to using payload mode.

Data Types

Overview

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

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

Using Source objects

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

`DOMSource`

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of `Node` objects that can be accessed using the `getNode()` method. Nodes can be updated or added to the DOM tree using the `setNode()` method.

`SAXSource`

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an `InputSource` object that contains the raw data and an `XMLReader` object that parses the raw data.

`StreamSource`

Holds XML messages as a data stream. The data stream can be manipulated as would any other data stream.

Important

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

Using SOAPMessage objects

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

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

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

Using DataSource objects

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

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

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

Implementing a Provider Object

Overview

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

- An implementation must have the @WebServiceProvider annotation.
- An implementation must have a default public constructor.
- An implementation must implement a typed version of the Provider interface.

  In other words, you cannot implement a Provider<T> interface. You must implement a version of the interface that uses a concrete data type as listed in Data Types. For example, you can implement an instance of a Provider<SAXSource>.

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

Working with messages

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

WS-I Basic Profile [http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html] provides guidelines about the messages used by services including:

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

Warning

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

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

Important

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

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

The @WebServiceProvider annotation

To be recognized by JAX-WS as a service implementation, a `Provider` implementation must be decorated with the `@WebServiceProvider` annotation.

Table 11, "`@WebServiceProvider` Properties" describes the properties you can set for the `@WebServiceProvider` annotation.

Table 11. `@WebServiceProvider` Properties

Property	Description
portName	Specifies the value of `name` attribute of the `wsdl:port` element that defines the service's endpoint.
serviceName	Specifies the value of `name` attribute of the `wsdl:service` element that contains the service's endpoint.
targetNamespace	Specifies the targetname space fop the service's WSDL definition.
wsdlLocation	Specifies the URI for the WSDL document definig the service.

All of these properties are optional and are empty by default. If you leave them empty, Artix ESB will create values using information from the implementation class.

Implementing the invoke() method

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

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

Examples

Example 54, “`Provider<SOAPMessage>` Implementation” shows a `Provider` implementation that works with `SOAPMessage` objects in message mode.

Example 54. `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 54, “`Provider<SOAPMessage>` Implementation” 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 needed for building the response.

❽ Builds the SOAP message for the response.

❾ Returns the response as a `SOAPMessage` object.

Example 55, “`Provider<DOMSource>` Implementation” shows an example of a `Provider` implementation using `DOMSource` objects in payload mode.

Example 55. `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"
serviceName="stockQuoteReporter")
❷@ServiceMode(value="Service.Mode.PAYLOAD")
public class  stockQuoteReporterProvider implements
Provider<DOMSource>
❸public stockQuoteReporterProvider()
  {
  }

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

The code in Example 55, “`Provider<DOMSource>` Implementation” 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

Summary

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.

Table of Contents

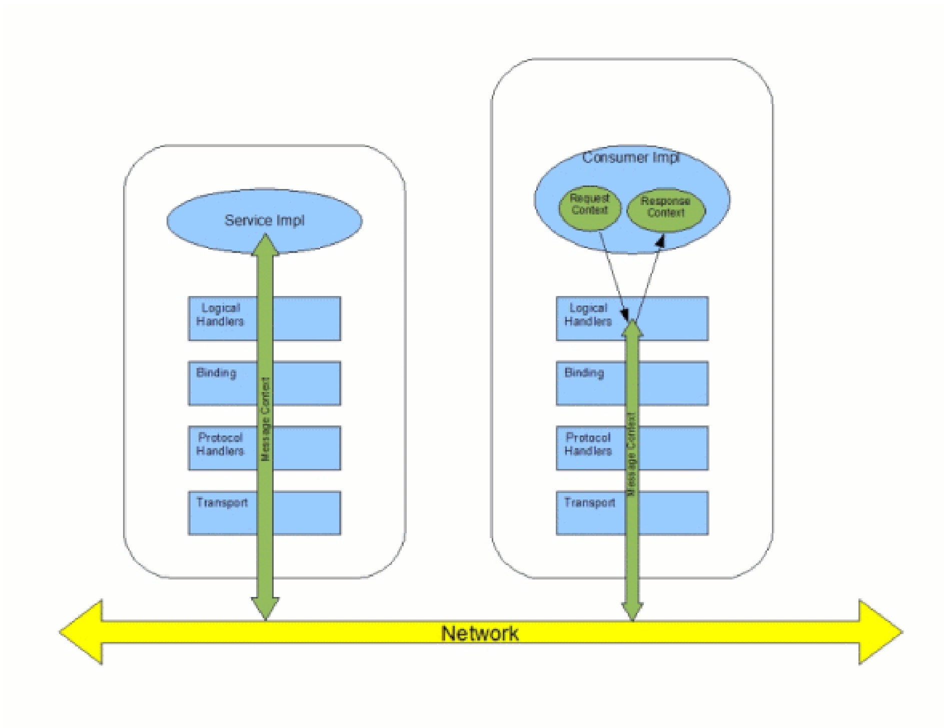
Understanding Contexts

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 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, “Message Contexts and Message Processing Path” 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 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 will still be returned from a message context as a plain `Object` object that must be cast back into a `UserInfo` object.

Properties in a message context also have a scope. The scope determines where in the message processing chain a property can be accessed.

Property scopes

Properties in a message context are scoped. A property can have one of two scopes:

`APPLICATION`

Properties scoped as `APPLICATION` are available to JAX-WS `Handler` implementations, consumer implementation code, and service provider implementation code. If a handler needed to pass a property to the service provider implementation, it would set 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 56, “The `MessageContext.setScope()` Method” shows the method's signature.

Example 56. 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 specifies the new scope for the property. The scope can be either

`MessageContext.Scope.APPLICATION` or `MessageContext.Scope.HANDLER`.

Overview contexts in Handler implementations

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. In addition, logical handlers can access the contents of the message body through the message context.

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.

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. One, the request context, holds a copy of the properties used for outgoing requests. The other, the 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 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.

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 that are 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 you need to 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 to be 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 57, “Obtaining a Context Object in a Service Implementation” shows code for obtaining a context object.

Example 57. 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 in it using the `get()` method shown in Example 58, “The `MessageContext.get()` Method”.

Example 58. 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 wish to retrieve from the context. The `get()` returns an object that must be cast to the proper type for the property. Table 12, “Properties Available in the Service Implementation Context” 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 will also change the value of the property in the context.

Example 59, “Getting a Property from a Service's Message Context” shows code for getting the name of the WSDL `operation` element that represents the invoked operation.

Example 59. 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_OPERATION);
```

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 60, “The `MessageContext.put()` Method”.

Example 60. 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 will replace the existing value with the new value and return the old value. If the property does not already exist in the message context, the `put()` method will set the property and return `null`.

Example 61, “Setting a Property in a Service's Message Context” shows code for setting the response code for an HTTP request.

Example 61. 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 12, "Properties Available in the Service Implementation Context" lists the properties accessible through the context in a service implementation object.

Table 12. 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 a Integer.
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 URL is http://cxf.apache.org/demo/widgets the path would be /demo/widgets.
QUERY_STRING[a]	Specifies the query, if any, attached to the URI used to invoke the request. The value is strored as a String.

Base Class	
Property Name	**Description**
	Queries appear at the end of the URI after a `?`. For example, if a request was made to `http://cxf.apache.org/demo/widgets?color` the query would be `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 will provide 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`. 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 would be `/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 defining the service being implemented. The value is stored as a `org.xml.sax.InputSource`.
`WSDL_SERVICE`[a]	Specifies the qualified name of the `wsdl:service` element defining the service being implemented. The value is stored as a `QName`.
`WSDL_PORT`[a]	Specifies the qualified name of the `wsdl:port` element defining the endpoint used to access the service. The value is stored as a `QName`.

Base Class	
Property Name	**Description**
WSDL_INTERFACE[a]	Specifies the qualified name of the wsdl:portType element defining the service being implemented. The value is stored as a QName.
WSDL_OPERATION[a]	Specifies the qualified name of the wsdl:operation element corresponding 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.
WSDL_DESCRIPTION	Specifies the WSDL document defining the service being implemented. The value is stored as a org.xml.sax.InputSource.
WSDL_SERVICE	Specifies the qualified name of the wsdl:service element defining the service being implemented. The value is stored as a QName.
WSDL_PORT	Specifies the qualified name of the wsdl:port element defining 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 defining the service being implemented. The value is stored as a QName.
WSDL_OPERATION	Specifies the qualified name of the wsdl:operation element corresponding 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 a Integer.

Base Class	
Property Name	**Description**
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 would be /demo/widgets.
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 was made to http://cxf.apache.org/demo/widgets?color the query would be 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	

Base Class	
Property Name	**Description**
JMS_SERVER_HEADERS	Contains the JMS message headers. For more information see Working with JMS Message Properties.

[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 inbound message from the last operation invocation 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 62, “The `getRequestContext()` Method”, returns the request context as a `Map` object. The returned `Map` object can be used to directly manipulate the contents of the context.

Example 62. The `getRequestContext()` Method

```
Map<String, Object> getRequestContext();
```

`getResponseContext()`

The `getResponseContext()`, shown in Example 63, “The `getResponseContext()` Method”, 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 remote invocation in the current thread.

Example 63. The `getResponseContext()` Method

```
Map<String, Object> getResponseContext();
```

Since proxy objects implement the `BindingProvider` interface, a `BindingProvider` object can be obtained by casting the 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 64, “Getting a Consumer's Request Context” shows code for obtaining the request context for a proxy.

Example 64. 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>` object. The maps have keys String and values of arbitrary type. Use `java.util.Map.get()` to access an entry in the hash map of response context properties.

To retrieve a particular context property, *`ContextPropertyName`*, use the code shown in Example 65, “Reading a Response Context Property”.

Example 65. 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>` object. The map has keys of String and values of arbitrary type. To set a property in the context you use the `java.util.Map.put()` method.

Tip

While you can set properties in both the request 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 66, “Setting a Request Context Property” changes the address of the target service provider by setting the value of the `BindingProvider.ENDPOINT_ADDRESS_PROPERTY`.

Example 66. 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/widgets");

// 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 13. 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 wishes to maintain session information. The value is stored as a `Boolean`.
`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.

[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

Table of Contents

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` do the following:

1. Obtain the context as described in Obtaining a context.
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 67, "Getting JMS Message Headers in a Service Implementation" shows code for getting the JMS message headers from a service's message context:

Example 67. 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.HelloWorldPortType",
                           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_SERVER_HEADERS);
       ...
```

```
    }
     ...
}
```

Getting JMS Message Header Properties in a Consumer

Once a message has been 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 will wait 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.
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 68, “Getting the JMS Headers from a Consumer Response Header” shows code for getting the JMS message header properties from a consumer's response context.

Example 68. 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 68, “Getting the JMS Headers from a Consumer Response Header” 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 14, "JMS Header Properties" lists the standard properties in the JMS header that you can inspect.

Table 14. 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 69, "Reading the JMS Header Properties" shows code for inspecting some of the JMS properties using the response context.

Example 69. 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 69, “Reading the JMS Header Properties” 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 will need to 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 15, “Settable JMS Header Properties” lists the properties in the JMS header that you can set using the consumer endpoint's request context.

Table 15. 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 wish to set using the appropriate setter methods from Table 15, “Settable JMS Header Properties”.
3. Set the values into 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 into 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` of `org.apache.cxf.transports.jms.context.JMSPropertyType`. 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 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 will wait 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 to consumer to wait as the second argument.

Example

Example 70, "Setting JMS Properties using the Request Context" shows code for setting some of the JMS properties using the request context.

Example 70. 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);

11  requestContext.put(JMSConstants.CLIENT_REQUEST_HEADERS, requestHdr);

12  requestContext.put(JMSConstants.CLIENT_RECEIVE_TIMEOUT, new Long(1000));
}
```

The code in Example 70, “Setting JMS Properties using the Request Context” 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.
11 Sets the JMS message header values into the request context.
12 Sets the client receive timeout property to 1 second.

Index

Symbols

A

W