Progress FUSE

FUSE™ ESB

Using the FUSE™ Services Framework Binding Component

[DRAFT]

Version 4.1
April 2009

PROGRESS SOFTWARE

Using the FUSE™ Services Framework Binding Component

Version 4.1

Publication date 22 Jul 2009
Copyright © 2001-2009 Progress Software Corporation and/or its subsidiaries or affiliates.

Legal Notices

Progress Software Corporation and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from Progress Software Corporation, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

Progress, IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix;, FUSE, and Making Software Work Together are trademarks or registered trademarks of Progress Software Corporation and/or its subsidiaries in the US and other countries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the US and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate Progress Software Corporation makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Progress Software Corporation shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of Progress Software Corporation. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. Progress Software Corporation assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction to the FUSE Services Framework Binding Component

The FUSE Services Framework binding component allows you to create SOAP/HTTP and SOAP/JMS endpoints.

Overview

The FUSE Services Framework binding component provides connectivity to external endpoints using either SOAP/HTTP or SOAP/JMS. The endpoints are defined using WSDl files that contain FUSE Services Framework specific extensions for defining the transport. In addition, you can add FUSE Services Framework-based Spring configuration to use the advanced features.

It allows for the creation of two types of endpoint:

consumer endpoint

A consumer endpoint listens for messages on a specified address. When it receives a message it sends it to the NMR for delivery to the appropriate endpoint. If the message is part of a two-way exchange, then the consumer endpoint is also responsible for returning the response to the external endpoint.

For information about configuring consumer endpoints see "Consumer Endpoints" on page 79.

provider endpoint

A provider endpoint receives messages from the NMR. It then packages the message as a SOAP message and sends it to the specified external address. If the message is part of a two-way message exchange, the provider endpoint waits for the response from the external endpoint. The provider endpoint will then direct the response back to the NMR.

For information about configuring provider endpoints see "Provider Endpoints" on page 87.

Key features

The FUSE Services Framework binding component has the following features:

- HTTP support
- JMS 1.1 support

- SOAP 1.1 support
- SOAP 1.2 support
- MTOM support
- Support for all MEPs as consumers or providers
- SSL support
- WS-Security support
- WS-Policy support
- WS-RM support
- WS-Addressing support

Steps for working with the FUSE Services Framework binding component

Using the FUSE Services Framework binding component to expose SOAP endpoints usually involves the following steps:

1. Defining the contract for your endpoint in WSDL.

   See Part I: on page 15.

2. Configuring the endpoint and packaging it into a service unit.

   See Part II: on page 73.

3. Bundling the service unit into a service assembly for deployment into the FUSE ESB container.

More information

For more information about using FUSE Services Framework to create SOAP endpoints see the FUSE Services Framework library.[1].

[1] http://fusesource.com/documentation/fuse-service-framework-documentation

Part I. Defining an Endpoint in WSDL

Endpoints are defined in WSDL 1.1 documents. The WSDL contract specifies the messages, operations, and the interfaces exposed by the endpoint. It also defines the transport used by the endpoint.

Chapter 2. Introducing WSDL Contracts

WSDL documents define services using Web Service Description Language and a number of possible extensions. The documents have a logical part and a concrete part. The abstract part of the contract defines the service in terms of implementation neutral data types and messages. The concrete part of the document defines how an endpoint implementing a service will interact with the outside world.

The recommended approach to design services is to define your services in WSDL and XML Schema before writing any code. When hand-editing WSDL documents you must make sure that the document is valid, as well as correct. To do this you must have some familiarity with WSDL. You can find the standard on the W3C web site, www.w3.org[1].

[1] http://www.w3.org/TR/wsdl

WSDL Elements

A WSDL document is made up of the following elements:

- `definitions` — The root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced in the WSDL document.
- `types` — The XML Schema definitions for the data units that form the building blocks of the messages used by a service. For information about defining data types see "Defining Logical Data Units" on page 21.
- `message` — The description of the messages exchanged during invocation of a services operations. These elements define the arguments of the operations making up your service. For information on defining messages see "Defining Logical Messages Used by a Service" on page 41.
- `portType` — A collection of `operation` elements describing the logical interface of a service. For information about defining port types see "Defining Your Logical Interfaces" on page 45.
- `operation` — The description of an action performed by a service. Operations are defined by the messages passed between two endpoints when the operation is invoked. For information on defining operations see "Operations" on page 46.
- `binding` — The concrete data format specification for an endpoint. A `binding` element defines how the abstract messages are mapped into the concrete data format used by an endpoint. This element is where specifics such as parameter order and return values are specified.
- `service` — A collection of related `port` elements. These elements are repositories for organizing endpoint definitions.
- `port` — The endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combined with the definition of transport details, and they define the physical endpoint on which a service is exposed.

Structure of a WSDL Document

A WSDL document is, at its simplest, a collection of elements contained within a root `definition` element. These elements describe a service and how an endpoint implementing that service is accessed.

A WSDL document has two distinct parts:

- An abstract part that defines the service in implementation neutral terms
- A concrete part that defines how an endpoint implementing the service is exposed on a network

The logical part

The logical part of a WSDL document contains the `types`, the `message`, and the `portType` elements. It describes the service's interface and the messages exchanged by the service. Within the `types` element, XML Schema is used to define the structure of the data that makes up the messages. A number of `message` elements are used to define the structure of the messages used by the service. The `portType` element contains one or more `operation` elements that define the messages sent by the operations exposed by the service.

The concrete part

The concrete part of a WSDL document contains the `binding` and the `service` elements. It describes how an endpoint that implements the service connects to the outside world. The `binding` elements describe how the data units described by the `message` elements are mapped into a concrete, on-the-wire data format, such as SOAP. The `service` elements contain one or more `port` elements which define the endpoints implementing the service.

Designing a contract

To design a WSDL contract for your services you must perform the following steps:

1. Define the data types used by your services.
2. Define the messages used by your services.
3. Define the interfaces for your services.
4. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
5. Define the transport details for each of the services.

Chapter 3. Defining Logical Data Units

When describing a service in a WSDL contract complex data types are defined as logical units using XML Schema.

When defining a service, the first thing you must consider is how the data used as parameters for the exposed operations is going to be represented. Unlike applications that are written in a programming language that uses fixed data structures, services must define their data in logical units that can be consumed by any number of applications. This involves two steps:

1. Breaking the data into logical units that can be mapped into the data types used by the physical implementations of the service

2. Combining the logical units into messages that are passed between endpoints to carry out the operations

This chapter discusses the first step. "Defining Logical Messages Used by a Service" on page 41 discusses the second step.

Mapping Data into Logical Data Units

The interfaces used to implement a service define the data representing operation parameters as XML documents. If you are defining an interface for a service that is already implemented, you must translate the data types of the implemented operations into discreet XML elements that can be assembled into messages. If you are starting from scratch, you must determine the building blocks from which your messages are built, so that they make sense from an implementation standpoint.

Available type systems for defining service data units

According to the WSDL specification, you can use any type system you choose to define data types in a WSDL contract. However, the W3C specification states that XML Schema is the preferred canonical type system for a WSDL document. Therefore, XML Schema is the intrinsic type system in FUSE Services Framework.

XML Schema as a type system

XML Schema is used to define how an XML document is structured. This is done by defining the elements that make up the document. These elements can use native XML Schema types, like xsd:int, or they can use types that are defined by the user. User defined types are either built up using combinations of XML elements or they are defined by restricting existing types. By combining type definitions and element definitions you can create intricate XML documents that can contain complex data.

When used in WSDL XML Schema defines the structure of the XML document that holds the data used to interact with a service. When defining the data units used by your service, you can define them as types that specify the structure of the message parts. You can also define your data units as elements that make up the message parts.

Considerations for creating your data units

You might consider simply creating logical data units that map directly to the types you envision using when implementing the service. While this approach works, and closely follows the model of building RPC-style applications, it is not necessarily ideal for building a piece of a service-oriented architecture.

The Web Services Interoperability Organization's WS-I basic profile provides a number of guidelines for defining data units and can be accessed at http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES. In addition, the W3C also provides the following guidelines for using XML Schema to represent data types in WSDL documents:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.

Adding Data Units to a Contract

Depending on how you choose to create your WSDL contract, creating new data definitions requires varying amounts of knowledge. The FUSE Services Framework GUI tools provide a number of aids for describing data types using XML Schema. Other XML editors offer different levels of assistance. Regardless of the editor you choose, it is a good idea to have some knowledge about what the resulting contract should look like.

Procedure

Defining the data used in a WSDL contract involves the following steps:

1. Determine all the data units used in the interface described by the contract.
2. Create a `types` element in your contract.
3. Create a `schema` element, shown in Example 3.1 on page 24, as a child of the `type` element.

   The `targetNamespace` attribute specifies the namespace under which new data types are defined. The remaining entries should not be changed.

Example 3.1. Schema Entry for a WSDL Contract

```
<schema targetNamespace="http://schemas.iona.com/bank.idl"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

4. For each complex type that is a collection of elements, define the data type using a `complexType` element. See "Defining Data Structures" on page 30.
5. For each array, define the data type using a `complexType` element. See "Defining Arrays" on page 34.
6. For each complex type that is derived from a simple type, define the data type using a `simpleType` element. See "Defining Types by Restriction" on page 37.
7. For each enumerated type, define the data type using a `simpleType` element. See "Defining Enumerated Types" on page 39.

8. For each element, define it using an `element` element. See "Defining Elements" on page 40.

XML Schema Simple Types

If a message part is going to be of a simple type it is not necessary to create a type definition for it. However, the complex types used by the interfaces defined in the contract are defined using simple types.

Entering simple types

XML Schema simple types are mainly placed in the `element` elements used in the types section of your contract. They are also used in the `base` attribute of `restriction` elements and `extension` elements.

Simple types are always entered using the xsd prefix. For example, to specify that an element is of type int, you would enter xsd:int in its `type` attribute as shown in Example 3.2 on page 26.

Example 3.2. Defining an Element with a Simple Type

```
<element name="simpleInt" type="xsd:int" />
```

Supported XSD simple types

FUSE Services Framework supports the following XML Schema simple types:

- xsd:string
- xsd:normalizedString
- xsd:int
- xsd:unsignedInt
- xsd:long
- xsd:unsignedLong
- xsd:short
- xsd:unsignedShort
- xsd:float
- xsd:double
- xsd:boolean
- xsd:byte

- xsd:unsignedByte
- xsd:integer
- xsd:positiveInteger
- xsd:negativeInteger
- xsd:nonPositiveInteger
- xsd:nonNegativeInteger
- xsd:decimal
- xsd:dateTime
- xsd:time
- xsd:date
- xsd:QName
- xsd:base64Binary
- xsd:hexBinary
- xsd:ID
- xsd:token
- xsd:language
- xsd:Name
- xsd:NCName
- xsd:NMTOKEN
- xsd:anySimpleType
- xsd:anyURI
- xsd:gYear
- xsd:gMonth
- xsd:gDay

- xsd:gYearMonth
- xsd:gMonthDay

Defining Complex Data Types

XML Schema provides a flexible and powerful mechanism for building complex data structures from its simple data types. You can create data structures by creating a sequence of elements and attributes. You can also extend your defined types to create even more complex types.

In addition to building complex data structures, you can also describe specialized types such as enumerated types, data types that have a specific range of values, or data types that need to follow certain patterns by either extending or restricting the primitive types.

Defining Data Structures

In XML Schema, data units that are a collection of data fields are defined using `complexType` elements. Specifying a complex type requires three pieces of information:

1. The name of the defined type is specified in the `name` attribute of the `complexType` element.
2. The first child element of the `complexType` describes the behavior of the structure's fields when it is put on the wire. See "Complex type varieties" on page 30.
3. Each of the fields of the defined structure are defined in `element` elements that are grandchildren of the `complexType` element. See "Defining the parts of a structure" on page 31.

For example, the structure shown in Example 3.3 on page 30 is be defined in XML Schema as a complex type with two elements.

Example 3.3. Simple Structure

```
struct personalInfo
{
  string name;
  int age;
};
```

Example 3.4 on page 30 shows one possible XML Schema mapping for the structure shown in Example 3.3 on page 30.

Example 3.4. A Complex Type

```
<complexType name="personalInfo">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </sequence>
</complexType>
```

Complex type varieties

XML Schema has three ways of describing how the fields of a complex type are organized when represented as an XML document and passed on the wire. The first child element of the `complexType` element determines which

variety of complex type is being used. Table 3.1 on page 31 shows the elements used to define complex type behavior.

Table 3.1. Complex Type Descriptor Elements

Element	Complex Type Behavior
`sequence`	All the complex type's fields must be present and they must be in the exact order they are specified in the type definition.
`all`	All of the complex type's fields must be present but they can be in any order.
`choice`	Only one of the elements in the structure can be placed in the message.

If a `sequence` element, an `all` element, or a `choice` is not specified, then a `sequence` is assumed. For example, the structure defined in Example 3.4 on page 30 generates a message containing two elements: `name` and `age`.

If the structure is defined using a `choice` element, as shown in Example 3.5 on page 31, it generates a message with either a `name` element or an `age` element.

Example 3.5. Simple Complex Choice Type

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

Defining the parts of a structure

You define the data fields that make up a structure using `element` elements. Every `complexType` element should contain at least one `element` element. Each `element` element in the `complexType` element represents a field in the defined data structure.

To fully describe a field in a data structure, `element` elements have two required attributes:

- The `name` attribute specifies the name of the data field and it must be unique within the defined complex type.

- The `type` attribute specifies the type of the data stored in the field. The type can be either one of the XML Schema simple types, or any named complex type that is defined in the contract.

In addition to `name` and `type`, `element` elements have two other commonly used optional attributes: `minOcurrs` and `maxOccurs`. These attributes place bounds on the number of times the field occurs in the structure. By default, each field occurs only once in a complex type. Using these attributes, you can change how many times a field must or can appear in a structure. For example, you can define a field, `previousJobs`, that must occur at least three times, and no more than seven times, as shown in Example 3.6 on page 32.

Example 3.6. Simple Complex Type with Occurrence Constraints

```
<complexType name="personalInfo>
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
    <element name="previousJobs" type="xsd:string:
             minOccurs="3" maxOccurs="7"/>
  </all>
</complexType>
```

You can also use the `minOccurs` to make the `age` field optional by setting the `minOccurs` to zero as shown in Example 3.7 on page 32. In this case `age` can be omitted and the data will still be valid.

Example 3.7. Simple Complex Type with `minOccurs` Set to Zero

```
<complexType name="personalInfo>
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

Defining attributes

In XML documents attributes are contained in the element's tag. For example, in the `complexType` element `name` is an attribute. They are specified using the `attribute` element. It comes after the `all`, `sequence`, or `choice` element and are a direct child of the `complexType` element. Example 3.8 on page 33 shows a complex type with an attribute.

Example 3.8. Complex Type with an Attribute

```
<complexType name="personalInfo>
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string"
             minOccurs="3" maxOccurs="7"/>
  </all>
  <attribute name="age" type="xsd:int" use="optional" />
</complexType>
```

The `attribute` element has three attributes:

- `name` — A required attribute that specifies the string identifying the attribute.
- `type` — Specifies the type of the data stored in the field. The type can be one of the XML Schema simple types.
- `use` — Specifies if the attribute is required or optional. Valid values are `required` or `optional`.

If you specify that the attribute is optional you can add the optional attribute `default`. The `default` attribute allows you to specify a default value for the attribute.

Defining Arrays

FUSE Services Framework supports two methods for defining arrays in a contract. The first is define a complex type with a single element whose `maxOccurs` attribute has a value greater than one. The second is to use SOAP arrays. SOAP arrays provide added functionality such as the ability to easily define multi-dimensional arrays and to transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are a special case of a sequence complex type. You simply define a complex type with a single element and specify a value for the `maxOccurs` attribute. For example, to define an array of twenty floating point numbers you use a complex type similar to the one shown in Example 3.9 on page 34.

Example 3.9. Complex Type Array

```
<complexType name="personalInfo>
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

You can also specify a value for the `minOccurs` attribute.

SOAP arrays

SOAP arrays are defined by deriving from the SOAP-ENC:Array base type using the `wsdl:arrayType` element. The syntax for this is shown in Example 3.10 on page 34.

Example 3.10. Syntax for a SOAP Array derived using `wsdl:arrayType`

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *`TypeName`* specifies the name of the newly-defined array type. *`ElementType`* specifies the type of the elements in the array. *`ArrayBounds`* specifies the number of dimensions in the array. To specify a single dimension array use `[]`; to specify a two-dimensional array use either `[][]` or `[,]`.

For example, the SOAP Array, SOAPStrings, shown in Example 3.11 on page 35, defines a one-dimensional array of strings. The `wsdl:arrayType` attribute specifies the type of the array elements, xsd:string, and the number of dimensions, with `[]` implying one dimension.

Example 3.11. Definition of a SOAP Array

```
<complexType name="SOAPStrings">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="xsd:string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in Example 3.12 on page 35.

Example 3.12. Syntax for a SOAP Array derived using an Element

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
                 maxOccurs="unbounded"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

When using this syntax, the element's `maxOccurs` attribute must always be set to `unbounded`.

Defining Types by Extension

Like most major coding languages, XML Schema allows you to create data types that inherit some of their elements from other data types. This is called defining a type by extension. For example, you could create a new type called `alienInfo`, that extends the `personalInfo` structure defined in Example 3.4 on page 30 by adding a new element called `planet`.

Types defined by extension have four parts:

1. The name of the type is defined by the `name` attribute of the `complexType` element.
2. The `complexContent` element specifies that the new type will have more than one element.

Note

If you are only adding new attributes to the complex type, you can use a `simpleContent` element.

3. The type from which the new type is derived, called the *base* type, is specified in the `base` attribute of the `extension` element.
4. The new type's elements and attributes are defined in the `extension` element, the same as they are for a regular complex type.

For example, `alienInfo` is defined as shown in Example 3.13 on page 36.

Example 3.13. Type Defined by Extension

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="personalInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Defining Types by Restriction

XML Schema allows you to create new types by restricting the possible values of an XML Schema simple type. For example, you can define a simple type, `SSN`, which is a string of exactly nine characters. New types defined by restricting simple types are defined using a `simpleType` element.

The definition of a type by restriction requires three things:

1. The name of the new type is specified by the `name` attribute of the `simpleType` element.
2. The simple type from which the new type is derived, called the *base type*, is specified in the `restriction` element. See "Specifying the base type" on page 37.
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the `restriction` element. See "Defining the restrictions" on page 37.

Specifying the base type

The base type is the type that is being restricted to define the new type. It is specified using a `restriction` element. The `restriction` element is the only child of a `simpleType` element and has one attribute, `base`, that specifies the base type. The base type can be any of the XML Schema simple types.

For example, to define a new type by restricting the values of an xsd:int you use a definition like the one shown in Example 3.14 on page 37.

Example 3.14. int as Base Type

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

Defining the restrictions

The rules defining the restrictions placed on the base type are called *facets*. Facets are elements with one attribute, `value`, that defines how the facet is enforced. The available facets and their valid `value` settings depend on the base type. For example, xsd:string supports six facets, including:

- length
- minLength
- maxLength
- pattern
- whitespace
- enumeration

Each facet element is a child of the `restriction` element.

Example

Example 3.15 on page 38 shows an example of a simple type, `SSN`, which represents a social security number. The resulting type is a string of the form `xxx-xx-xxxx`. <SSN>032-43-9876<SSN> is a valid value for an element of this type, but <SSN>032439876</SSN> is not.

Example 3.15. SSN Simple Type Description

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

Defining Enumerated Types

Enumerated types in XML Schema are a special case of definition by restriction. They are described by using the `enumeration` facet which is supported by all XML Schema primitive types. As with enumerated types in most modern programming languages, a variable of this type can only have one of the specified values.

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in Example 3.16 on page 39.

Example 3.16. Syntax for an Enumeration

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value"/>
    <enumeration value="Case2Value"/>
    ...
    <enumeration value="CaseNValue"/>
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

Example

For example, an XML document with an element defined by the enumeration `widgetSize`, shown in Example 3.17 on page 39, would be valid if it contained <widgetSize>big</widgetSize>, but it would not be valid if it contained <widgetSize>big,mungo</widgetSize>.

Example 3.17. widgetSize Enumeration

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

Defining Elements

Elements in XML Schema represent an instance of an element in an XML document generated from the schema. The most basic element consists of a single `element` element. Like the `element` element used to define the members of a complex type, they have three attributes:

- `name` — A required attribute that specifies the name of the element as it appears in an XML document.
- `type` — Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. This attribute can be omitted if the type has an in-line definition.
- `nillable` — Specifies whether an element can be omitted from a document entirely. If `nillable` is set to `true`, the element can be omitted from any document generated using the schema.

An element can also have an *in-line* type definition. In-line types are specified using either a `complexType` element or a `simpleType` element. Once you specify if the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data. In-line type definitions are discouraged because they are not reusable.

Chapter 4. Defining Logical Messages Used by a Service

A service is defined by the messages exchanged when its operations are invoked. In a WSDL contract these messages are defined using `message` *element. The messages are made up of one or more parts that are defined using* `part` *elements.*

A service's operations are defined by specifying the logical messages that are exchanged when an operation is invoked. These logical messages define the data that is passed over a network as an XML document. They contain all of the parameters that are a part of a method invocation.

Logical messages are defined using the `message` element in your contracts. Each logical message consists of one or more parts, defined in `part` elements.

Tip

While your messages can list each parameter as a separate part, the recommended practice is to use only a single part that encapsulates the data needed for the operation.

Messages and parameter lists

Each operation exposed by a service can have only one input message and one output message. The input message defines all of the information the service receives when the operation is invoked. The output message defines all of the data that the service returns when the operation is completed. Fault messages define the data that the service returns when an error occurs.

In addition, each operation can have any number of fault messages. The fault messages define the data that is returned when the service encounters an error. These messages usually have only one part that provides enough information for the consumer to understand the error.

Message design for integrating with legacy systems

If you are defining an existing application as a service, you must ensure that each parameter used by the method implementing the operation is represented in a message. You must also ensure that the return value is included in the operation's output message.

One approach to defining your messages is RPC style. When using RPC style, you define the messages using one part for each parameter in the method's

parameter list. Each message part is based on a type defined in the `types` element of the contract. Your input message contains one part for each input parameter in the method. Your output message contains one part for each output parameter, plus a part to represent the return value, if needed. If a parameter is both an input and an output parameter, it is listed as a part for both the input message and the output message.

RPC style message definition is useful when service enabling legacy systems that use transports such as Tibco or CORBA. These systems are designed around procedures and methods. As such, they are easiest to model using messages that resemble the parameter lists for the operation being invoked. RPC style also makes a cleaner mapping between the service and the application it is exposing.

Message design for SOAP services

While RPC style is useful for modeling existing systems, the service's community strongly favors the wrapped document style. In wrapped document style, each message has a single part. The message's part references a wrapper element defined in the `types` element of the contract. The wrapper element has the following characteristics:

- It is a complex type containing a sequence of elements. For more information see "Defining Complex Data Types" on page 29.
- If it is a wrapper for an input message:
  - It has one element for each of the method's input parameters.
  - Its name is the same as the name of the operation with which it is associated.
- If it is a wrapper for an output message:
  - It has one element for each of the method's output parameters and one element for each of the method's inout parameters.
  - Its first element represents the method's return parameter.
  - Its name would be generated by appending `Response` to the name of the operation with which the wrapper is associated.

Message naming

Each message in a contract must have a unique name within its namespace. It is recommended that you use the following naming conventions:

- Messages should only be used by a single operation.
- Input message names are formed by appending `Request` to the name of the operation.
- Output message names are formed by appending `Response` to the name of the operation.
- Fault message names should represent the reason for the fault.

Message parts

Message parts are the formal data units of the logical message. Each part is defined using a `part` element, and is identified by a `name` attribute and either a `type` attribute or an `element` attribute that specifies its data type. The data type attributes are listed in Table 4.1 on page 43.

Table 4.1. Part Data Type Attributes

Attribute	Description
`element="`*`elem_name`*`"`	The data type of the part is defined by an element called *`elem_name`*.
`type="`*`type_name`*`"`	The data type of the part is defined by a type called *`type_name`*.

Messages are allowed to reuse part names. For instance, if a method has a parameter, *`foo`*, that is passed by reference or is an in/out, it can be a part in both the request message and the response message, as shown in Example 4.1 on page 43.

Example 4.1. Reused Part

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
<message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
<message>
```

Example

For example, imagine you had a server that stored personal information and provided a method that returned an employee's data based on the employee's ID number. The method signature for looking up the data is similar to Example 4.2 on page 44.

Example 4.2. personalInfo lookup Method

```
personalInfo lookup(long empId)
```

This method signature can be mapped to the RPC style WSDL fragment shown in Example 4.3 on page 44.

Example 4.3. RPC WSDL Message Definitions

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo"/>
<message/>
```

It can also be mapped to the wrapped document style WSDL fragment shown in Example 4.4 on page 44.

Example 4.4. Wrapped Document WSDL Message Definitions

```
<types>
  <schema ...>
  ...
  <element name="personalLookup">
    <complexType>
      <sequence>
        <element name="empID" type="xsd:int" />
      </sequence>
    </complexType>
  </element>
  <element name="personalLookupResponse">
    <complexType>
      <sequence>
        <element name="return" type="personalInfo" />
      </sequence>
    </complexType>
  </element>
  </schema>
</types>
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse>
  <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
```

Chapter 5. Defining Your Logical Interfaces

Logical service interfaces are defined using the `portType` element.

Logical service interfaces are defined using the WSDL `portType` element. The `portType` element is a collection of abstract operation definitions. Each operation is defined by the input, output, and fault messages used to complete the transaction the operation represents. When code is generated to implement the service interface defined by a `portType` element, each operation is converted into a method containing the parameters defined by the input, output, and fault messages specified in the contract.

Process

To define a logical interface in a WSDL contract you must do the following:

1. Create a `portType` element to contain the interface definition and give it a unique name. See "Port types" on page 45.
2. Create an `operation` element for each operation defined in the interface. See "Operations" on page 46.
3. For each operation, specify the messages used to represent the operation's parameter list, return type, and exceptions. See "Operation messages" on page 46.

Port types

A WSDL `portType` element is the root element in a logical interface definition. While many Web service implementations map `portType` elements directly to generated implementation objects, a logical interface definition does not specify the exact functionality provided by the the implemented service. For example, a logical interface named `ticketSystem` can result in an implementation that either sells concert tickets or issues parking tickets.

The `portType` element is the unit of a WSDL document that is mapped into a binding to define the physical data used by an endpoint exposing the defined service.

Each `portType` element in a WSDL document must have a unique name, which is specified using the `name` attribute, and is made up of a collection of

operations, which are described in `operation` elements. A WSDL document can describe any number of port types.

Operations

Logical operations, defined using WSDL `operation` elements, define the interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation defined within a `portType` element must have a unique name, specified using the `name` attribute. The `name` attribute is required to define an operation.

Operation messages

Logical operations are made up of a set of elements representing the logical messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in Table 5.1 on page 46.

Table 5.1. Operation Message Elements

Element	Description
`input`	Specifies the message the client endpoint sends to the service provider when a request is made. The parts of this message correspond to the input parameters of the operation.
`output`	Specifies the message that the service provider sends to the client endpoint in response to a request. The parts of this message correspond to any operation parameters that can be changed by the service provider, such as values passed by reference. This includes the return value of the operation.
`fault`	Specifies a message used to communicate an error condition between the endpoints.

An operation is required to have at least one `input` or one `output` element. An operation can have both `input` and `output` elements, but it can only have one of each. Operations are not required to have any `fault` elements, but can, if required, have any number of `fault` elements.

The elements have the two attributes listed in Table 5.2 on page 47.

Table 5.2. Attributes of the Input and Output Elements

Attribute	Description
`name`	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
`message`	Specifies the abstract message that describes the data being sent or received. The value of the `message` attribute must correspond to the `name` attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the `name` attribute for all `input` and `output` elements; WSDL provides a default naming scheme based on the enclosing operation's name. If only one element is used in the operation, the element name defaults to the name of the operation. If both an `input` and an `output` element are used, the element name defaults to the name of the operation with either `Request` or `Response` respectively appended to the name.

Return values

Because the `operation` element is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value it will be mapped into the `output` element as the last part of that message.

Example

For example, you might have an interface similar to the one shown in Example 5.1 on page 47.

Example 5.1. personalInfo lookup interface

```
interface personalInfoLookup
{
  personalInfo lookup(in int empID)
  raises(idNotFound);
}
```

This interface can be mapped to the port type in Example 5.2 on page 47.

Example 5.2. personalInfo lookup port type

```
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalLookupResponse"/>
```

```
<message/>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound"/>
<message/>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest"/>
    <output name="return" message="personalLookupResponse"/>
    <fault name="exception" message="idNotFoundException"/>
  </operation>
</portType>
```

Chapter 6. Using HTTP

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is the assumed transport for most WS- specifications and is integral to RESTful architectures.*

Adding a Basic HTTP Endpoint

Overview

There are three ways of specifying an HTTP endpoint's address depending on the payload format you are using.

- SOAP 1.1 uses the standardized `soap:address` element.
- SOAP 1.2 uses the `soap12:address` element.
- All other payload formats use the `http:address element.`

SOAP 1.1

When you are sending SOAP 1.1 messages over HTTP you must use the SOAP 1.1 `address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL. The SOAP 1.1 `address` element is defined in the namespace `http://schemas.xmlsoap.org/wsdl/soap/`.

Example 6.1 on page 50 shows a `port` element used to send SOAP 1.1 messages over HTTP.

Example 6.1. SOAP 1.1 Port Element

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  ...
  <service name="SOAP11Service">
    <port binding="SOAP11Binding" name="SOAP11Port">
      <soap:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

SOAP 1.2

When you are sending SOAP 1.2 messages over HTTP you must use the SOAP 1.2 `address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL. The SOAP 1.2 `address` element is defined in the namespace `http://schemas.xmlsoap.org/wsdl/soap12/`.

Example 6.2 on page 51 shows a `port` element used to send SOAP 1.2 messages over HTTP.

Example 6.2. SOAP 1.2 Port Element

```
<definitions ...
             xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ... >
  <service name="SOAP12Service">
    <port binding="SOAP12Binding" name="SOAP12Port">
      <soap12:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

Other messages types

When your messages are mapped to any payload format other than SOAP you must use the HTTP `address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL. The HTTP `address` element is defined in the namespace `http://schemas.xmlsoap.org/wsdl/http/`.

Example 6.3 on page 51 shows a `port` element used to send an XML message.

Example 6.3. HTTP Port Element

```
<definitions ...
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ... >
  <service name="HTTPService">
    <port binding="HTTPBinding" name="HTTPPort">
      <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

Consumer Configuration

Namespace

The WSDL extension elements used to configure an HTTP consumer endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the line shown in Example 6.4 on page 52 to the `definitions` element of your endpoint's WSDL document.

Example 6.4. HTTP Consumer WSDL Element's Namespace

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

Configuring the endpoint

The `http-conf:client` element is used to specify the connection properties of an HTTP consumer in a WSDL document. The `http-conf:client` element is a child of the WSDL `port` element. The attributes are described in Table 6.1 on page 52.

Table 6.1. HTTP Consumer Configuration Attributes

Attribute	Description
`ConnectionTimeout`	Specifies the amount of time, in milliseconds, that the consumer attempts to establish a connection before it times out. The default is `30000`. `0` specifies that the consumer will continue to send the request indefinitely.
`ReceiveTimeout`	Specifies the amount of time, in milliseconds, that the consumer will wait for a response before it times out. The default is `30000`. `0` specifies that the consumer will wait indefinitely.
`AutoRedirect`	Specifies if the consumer will automatically follow a server issued redirection. The default is `false`.
`MaxRetransmits`	Specifies the maximum number of times a consumer will retransmit a request to satisfy a redirect. The default is `-1` which specifies that unlimited retransmissions are allowed.
`AllowChunking`	Specifies whether the consumer will send requests using chunking. The default is `true` which specifies that the consumer will use chunking when sending requests. Chunking cannot be used if either of the following are true:

Attribute	Description
	• `http-conf:basicAuthSupplier` is configured to provide credentials preemptively. • `AutoRedirect` is set to `true`. In both cases the value of `AllowChunking` is ignored and chunking is disallowed.
`Accept`	Specifies what media types the consumer is prepared to handle. The value is used as the value of the HTTP Accept property. The value of the attribute is specified using multipurpose internet mail extensions (MIME) types.
`AcceptLanguage`	Specifies what language (for example, American English) the consumer prefers for the purpose of receiving a response. The value is used as the value of the HTTP AcceptLanguage property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.
`AcceptEncoding`	Specifies what content encodings the consumer is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP AcceptEncoding property.
`ContentType`	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType property. The default is `text/xml`. For web services, this should be set to `text/xml`. If the client is sending HTML form data to a CGI script, this should be set to `application/x-www-form-urlencoded`. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to `application/octet-stream`.
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP Host property. This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).
`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • `Keep-Alive` — Specifies that the consumer wants the connection kept open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it.

Attribute	Description
	• close(default) — Specifies that the connection to the server is closed after each request/response sequence.
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a consumer to a service provider. See "Consumer Cache Control Directives" on page 55.
Cookie	Specifies a static cookie to be sent with all requests.
BrowserType	Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based on the client that is sending the request.
Referer	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property. This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. If the AutoRedirect attribute is set to true and the request is redirected, any value specified in the Referer attribute is overridden. The value of the HTTP Referer property is set to the URL of the service that redirected the consumer's original request.
DecoupledEndpoint	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate provider->consumer connection. For more information on using decoupled endpoints see, "Using the HTTP Transport in Decoupled Mode" on page 60. You must configure both the consumer endpoint and the service provider endpoint to use WS-Addressing for the decoupled endpoint to work.
ProxyServer	Specifies the URL of the proxy server through which requests are routed.
ProxyServerPort	Specifies the port number of the proxy server through which requests are routed.
ProxyServerType	Specifies the type of proxy server used to route requests. Valid values are: • HTTP(default)

Attribute	Description
	• SOCKS

Consumer Cache Control Directives

Table 6.2 on page 55 lists the cache control directives supported by an HTTP consumer.

Table 6.2. `http-conf:client` Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store either any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that is still fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

Example

Example 6.5 on page 56 shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 6.5. WSDL to Configure an HTTP Consumer Endpoint

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

Provider Configuration

Namespace

The WSDL extension elements used to configure an HTTP provider endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. To use the HTTP configuration elements you must add the line shown in Example 6.6 on page 57 to the `definitions` element of your endpoint's WSDL document.

Example 6.6. HTTP Provider WSDL Element's Namespace

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

Configuring the endpoint

The `http-conf:server` element is used to specify the connection properties of an HTTP service provider in a WSDL document. The `http-conf:server` element is a child of the WSDL `port` element. The attributes are described in Table 6.3 on page 57.

Table 6.3. HTTP Service Provider Configuration Attributes

Attribute	Description
`ReceiveTimeout`	Sets the length of time, in milliseconds, the service provider attempts to receive a request before the connection times out. The default is `30000`. `0` specifies that the provider will not timeout.
`SuppressClientSendErrors`	Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. The default is `false`; exceptions are thrown on encountering errors.
`SuppressClientReceiveErrors`	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a consumer. The default is `false`; exceptions are thrown on encountering errors.
`HonorKeepAlive`	Specifies whether the service provider honors requests for a connection to remain open after a response has been sent. The default is `false`; keep-alive requests are ignored.
`RedirectURL`	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to `302` and the status description

Attribute	Description
	is set to `Object Moved`. The value is used as the value of the HTTP RedirectURL property.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a service provider to a consumer. See "Service Provider Cache Control Directives" on page 58.
`ContentLocation`	Sets the URL where the resource being sent in a response is located.
`ContentType`	Specifies the media type of the information being sent in a response. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType location.
`ContentEncoding`	Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`. This value is used as the value of the HTTP ContentEncoding property. The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. FUSE Services Framework performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.
`ServerType`	Specifies what type of server is sending the response. Values take the form *`program-name/version`*; for example, `Apache/1.2.5`.

Service Provider Cache Control Directives

Table 6.4 on page 58 lists the cache control directives supported by an HTTP service provider.

Table 6.4. `http-conf:server` Cache Control Directives

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

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

Example

Example 6.7 on page 59 shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 6.7. WSDL to Configure an HTTP Service Provider Endpoint

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

Using the HTTP Transport in Decoupled Mode

Overview

In normal HTTP request/response scenarios, the request and the response are sent using the same HTTP connection. The service provider processes the request and responds with a response containing the appropriate HTTP status code and the contents of the response. In the case of a successful request, the HTTP status code is set to `200`.

In some instances, such as when using WS-RM or when requests take an extended period of time to execute, it makes sense to decouple the request and response message. In this case the service providers sends the consumer a `202 Accepted` response to the consumer over the back-channel of the HTTP connection on which the request was received. It then processes the request and sends the response back to the consumer using a new decoupled server->client HTTP connection. The consumer runtime receives the incoming response and correlates it with the appropriate request before returning to the application code.

Configuring decoupled interactions

Using the HTTP transport in decoupled mode requires that you do the following:

1. Configure the consumer to use WS-Addressing.

   See "Configuring an endpoint to use WS-Addressing" on page 60.

2. Configure the consumer to use a decoupled endpoint.

   See "Configuring the consumer" on page 61.

3. Configure any service providers that the consumer interacts with to use WS-Addressing.

   See "Configuring an endpoint to use WS-Addressing" on page 60.

Configuring an endpoint to use WS-Addressing

Specify that the consumer and any service provider with which the consumer interacts use WS-Addressing.

You can specify that an endpoint uses WS-Addressing in one of two ways:

- Adding the `wswa:UsingAddressing` element to the endpoint's WSDL `port` element as shown in Example 6.8 on page 61.

Example 6.8. Activating WS-Addressing using WSDL

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wswa:UsingAddressing xmlns:wswa="http://www.w3.org/2005/02/addressing/wsdl"/>
  </port>
</service>
...
```

- Adding the WS-Addressing policy to the endpoint's WSDL `port` element as shown in Example 6.9 on page 61.

Example 6.9. Activating WS-Addressing using a Policy

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy">
      <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
  </port>
</service>
...
```

Note

The WS-Addressing policy supersedes the `wswa:UsingAddressing` WSDL element.

Configuring the consumer

Configure the consumer endpoint to use a decoupled endpoint using the `DecoupledEndpoint` attribute of the `http-conf:conduit` element.

Example 6.10 on page 62 shows the configuration for setting up the endpoint defined in Example 6.8 on page 61 to use use a decoupled endpoint. The consumer now receives all responses at `http://widgetvendor.net/widgetSellerInbox`.

Example 6.10. Configuring a Consumer to Use a Decoupled HTTP Endpoint

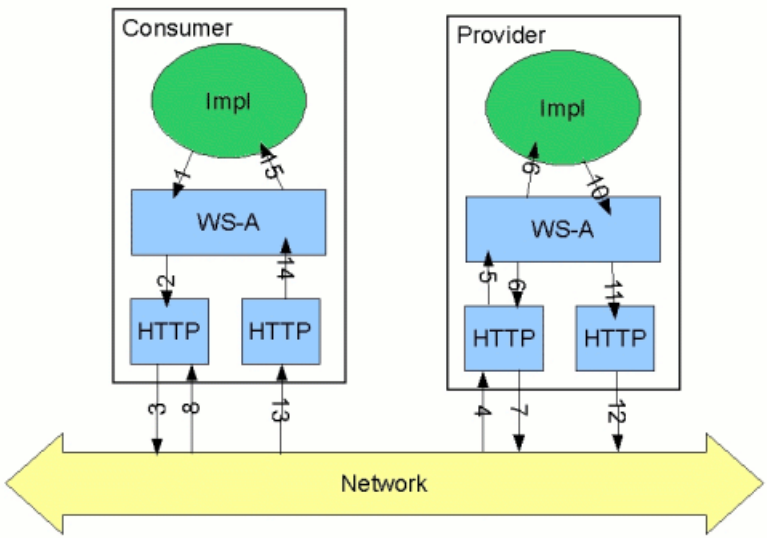
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                         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


  <http:conduit name="{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
    <http:client DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

How messages are processed

Using the HTTP transport in decoupled mode adds extra layers of complexity to the processing of HTTP messages. While the added complexity is transparent to the implementation level code in an application, it might be important to understand what happens for debugging reasons.

Figure 6.1 on page 63 shows the flow of messages when using HTTP in decoupled mode.

Figure 6.1. Message Flow in for a Decoupled HTTP Transport



A request starts the following process:

1. The consumer implementation invokes an operation and a request message is generated.
2. The WS-Addressing layer adds the WS-A headers to the message.

   When a decoupled endpoint is specified in the consumer's configuration, the address of the decoupled endpoint is placed in the WS-A ReplyTo header.
3. The message is sent to the service provider.

4. The service provider receives the message.
5. The request message from the consumer is dispatched to the provider's WS-A layer.
6. Because the WS-A ReplyTo header is not set to anonymous, the provider sends back a message with the HTTP status code set to `202`, acknowledging that the request has been received.
7. The HTTP layer sends a `202 Accepted` message back to the consumer using the original connection's back-channel.
8. The consumer receives the `202 Accepted` reply on the back-channel of the HTTP connection used to send the original message.

   When the consumer receives the `202 Accepted` reply, the HTTP connection closes.
9. The request is passed to the service provider's implementation where the request is processed.
10. When the response is ready, it is dispatched to the WS-A layer.
11. The WS-A layer adds the WS-Addressing headers to the response message.
12. The HTTP transport sends the response to the consumer's decoupled endpoint.
13. The consumer's decoupled endpoint receives the response from the service provider.
14. The response is dispatched to the consumer's WS-A layer where it is correlated to the proper request using the WS-A RelatesTo header.
15. The correlated response is returned to the client implementation and the invoking call is unblocked.

Chapter 7. Using JMS

The JMS is a standards based messaging system that is widely used in enterprise Java applications.

Basic Configuration

WSDL Namespace

The WSDL extensions for defining a JMS endpoint are defined in the namespace `http://cxf.apache.org/transports/jms`. In order to use the JMS extensions you will need to add the line shown in Example 7.1 on page 66 to the definitions element of your contract.

Example 7.1. JMS Extension Namespace

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

The address element

The basic configuration for a JMS endpoint is done by using a `jms:address` element as the child of your service's `port` element. The `jms:address` element's attributes are listed in Table 7.1 on page 66. The `jms:address` element uses a `jms:JMSNamingProperties` child element to specify additional information needed to connect to a JNDI provider.

Table 7.1. JMS Endpoint Attributes

Attribute	Description
`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jmsDestinationName`	Specifies the JMS name of the JMS destination to which requests are sent.
`jmsReplyDestinationName`	Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see "Using a Named Reply Destination" on page 69.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which requests are sent.
`jndiReplyDestinationName`	Specifies the JNDI name bound to the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see "Using a Named Reply Destination" on page 69.
`connectionUserName`	Specifies the user name to use when connecting to a JMS broker.

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

The JMSNamingProperties element

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

The following is a list of common JNDI properties that can be set:

1. `java.naming.factory.initial`
2. `java.naming.provider.url`
3. `java.naming.factory.object`
4. `java.naming.factory.state`
5. `java.naming.factory.url.pkgs`
6. `java.naming.dns.url`
7. `java.naming.authoritative`
8. `java.naming.batchsize`
9. `java.naming.referral`
10. `java.naming.security.protocol`
11. `java.naming.security.authentication`
12. `java.naming.security.principal`
13. `java.naming.security.credentials`

14. `java.naming.language`

15. `java.naming.applet`

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

Example

Example 7.2 on page 68 shows an example of a JMS WSDL `port` specification.

Example 7.2. JMS WSDL Port Specification

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

Using a Named Reply Destination

Overview

By default, FUSE Services Framework endpoints using JMS create a temporary queue for sending replies back and forth. If you prefer to use named queues, you can configure the queue used to send replies as part of an endpoint's JMS configuration.

Setting the reply destination name

You specify the reply destination using either the `jmsReplyDestinationName` attribute or the `jndiReplyDestinationName` attribute in the endpoint's JMS configuration. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the `ReplyTo` field of all outgoing requests. A service endpoint will use the value of the `jndiReplyDestinationName` attribute as the location for placing replies if there is no destination specified in the request's `ReplyTo` field.

Example

Example 7.3 on page 69 shows the configuration for a JMS client endpoint.

Example 7.3. JMS Consumer Specification Using a Named Reply Queue

```
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
                 jndiConnectionFactoryName="myConnectionFactory"
                 jndiDestinationName="myDestination"
                 jndiReplyDestinationName="myReplyDestination" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.apache.cxf.transport.jms.MyInitialContextFactory"
/>
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </jms:conduit>
```

JMS Consumer Configuration

Specifying the message type

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

Table 7.2. JMS Client WSDL Extensions

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

Example

Example 7.4 on page 70 shows the WSDL for configuring a JMS consumer endpoint.

Example 7.4. WSDL for a JMS Consumer Endpoint

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

JMS Provider Configuration

Overview

JMS provider endpoints have a number of behaviors that are configurable. These include:

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

Configuring the endpoint

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

Table 7.3. JMS Provider Endpoint WSDL Extensions

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

[a]Currently,setting the `transactional` attribute to `true` is not supported by the runtime.

Example

Example 7.5 on page 71 shows the WSDL for configuring a JMS provider endpoint.

Example 7.5. WSDL for a JMS Provider Endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
```

```
                jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:server messageSelector="cxf_message_selector"
                useMessageIDAsCorrelationID="true"
                transactional="true"
                durableSubscriberName="cxf_subscriber" />
  </port>
</service>
```

Part II. Configuring and Packaging Endpoints

Endpoints exposed by the FUSE Services Framework binding component are configured in a service unit's `xbean.xml` file. The endpoints are then packaged into a service unit that can be deployed to FUSE ESB.

Chapter 8. Introduction to the FUSE Services Framework Binding Component

Endpoints being deployed using the FUSE Services Framework binding component are packaged into a service unit. The service unit will container the WSDL document defining the endpoint's interface and a configuration file that sets-up the endpoint's runtime behavior.

Contents of a file component service unit

A service unit that configures the FUSE Services Framework binding component will contain the following artifacts:

`xbean.xml`

The `xbean.xml` file contains the XML configuration for the endpoint defined by the service unit. The contents of this file are the focus of this guide.

Note

The service unit can define more than one endpoint.

WSDL file
: The WSDL file defines the endpoint the interface exposes.

Spring configuration file
: The Spring configuration file contains configuration for the FUSE Services Framework runtime.

`meta-inf/jbi.xml`

The `jbi.xml` file is the JBI descriptor for the service unit. Example 8.1 on page 75 shows a JBI descriptor for a FUSE Services Framework binding component service unit.

Example 8.1. JBI Descriptor for a FUSE Services Framework Binding Component Service Unit

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false" />
</jbi>
```

For information on using the Maven tooling to package endpoints into a JBI service unit see Appendix C: on page 147.

OSGi Packaging

You can package FUSE Services Framework binding component endpoints in an OSGi bundle. To do so you need to make two minor changes:

- you will need to include an OSGi bundle manifest in the `META-INF` folder of the bundle.
- You need to add the following to your service unit's configuration file:

```
<bean class="org.apache.servicemix.common.osgi.EndpointExporter" />
```

Important

When you deploy FUSE Services Framework binding component endpoints in an OSGi bundle, the resulting endpoints are deployed as a JBI service unit.

For more information on using the OSGi packaging see Appendix D on page 163.

Namespace

The elements used to configure FUSE Services Framework binding component endpoints are defined in the http://servicemix.apache.org/cxfbc/1.0 namespace. You will need to add a namespace declaration similar to the one in Example 8.2 on page 76 to your `xbeans.xml` file's `beans` element.

Example 8.2. Namespace Declaration for Using FUSE Services Framework Binding Component Endpoints

```
<beans ...
       xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
</beans>
```

In addition, you need to add the schema location to the Spring `beans` element's `xsi:schemaLocation` as shown in Example 8.3 on page 76.

Example 8.3. Schema Location for Using FUSE Services Framework Binding Component Endpoints

```
<beans ...
       xsi:schemaLocation="...
```

```
http://servicemix.apache.org/cxfbc/1.0 http://service
mix.apache.org/cxfbc/1.0/servicemix-cxfbc.xsd
...">
  ...
</beans>
```

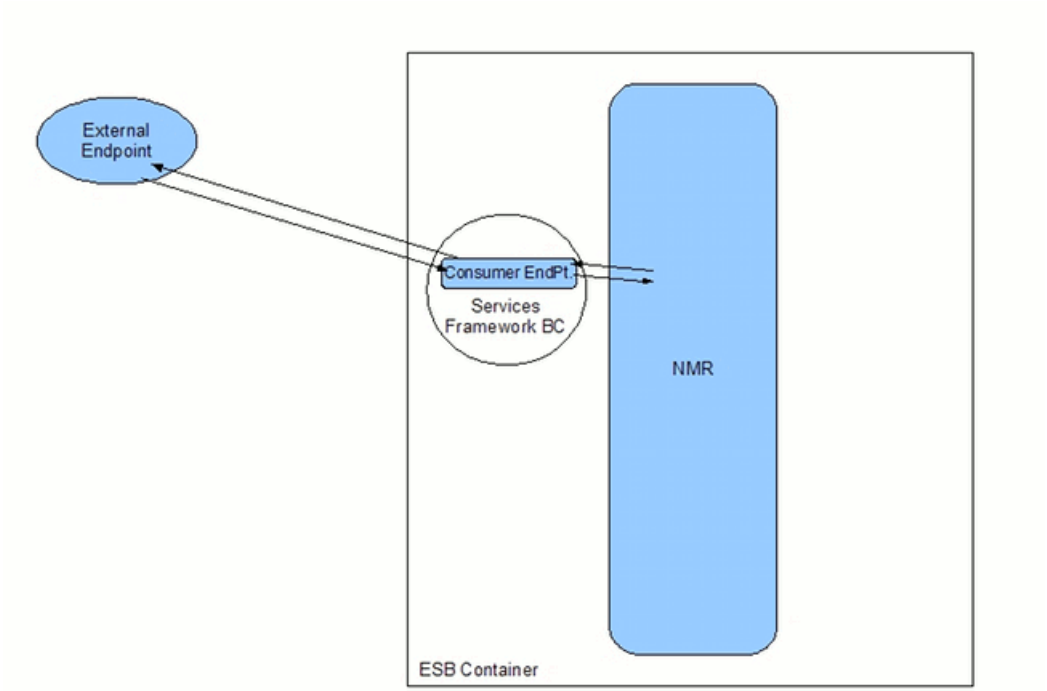
Chapter 9. Consumer Endpoints

A consumer endpoint listens for requests from external endpoints and delivers responses back to the requesting endpoint. It is configured using a single XML element that specifies the WSDL document defining the endpoint.

Overview

Consumer endpoints play the role of consumer from the vantage point of other endpoints running inside of the ESB. However, from outside of the ESB a consumer endpoint plays the role of a service provider. As shown in Figure 9.1 on page 80, consumer endpoints listen from incoming requests from external endpoints. When it receives a request, the consumer passes it off to the NMR fro delivery to endpoint that will process the request. If a response is generated, the consumer endpoint delivers the response back to the external endpoint.

Figure 9.1. Consumer Endpoint



Important

Because consumer endpoint's behave like service providers to external endpoints, you configure the runtime behavior of the transport using the provider-specific WSDL entries.

Procedure

To configure a consumer endpoint do the following:

1. Add a `consumer` element to your `xbean.xml` file.
2. Add a `wsdl` attribute to the `consumer` element.

See "Specifying the WSDL" on page 81.

3. If your WSDL defines more than one service, you will need to specify a value for the `service` attribute.

   See "Specifying the endpoint details" on page 82.

4. If the service you choose defines more than one endpoint, you will need to specify a value for the `endpoint` attribute.

   See "Specifying the endpoint details" on page 82.

5. Specify the details for the target of the requests received by the endpoint.

   See "Specifying the target endpoint" on page 84.

6. If your endpoint is going to be receiving binary attachments set its `mtomEnabled` attribute to `true`.

   See "Using MTOM to Process Binary Content" on page 93.

7. If your endpoint does not need to process the JBI wrapper set its `useJbiWrapper` attribute to `false`.

   See "Working with the JBI Wrapper" on page 95.

8. If you are using any of the advanced features, such as WS-Addressing or WS-Policy, specify a value for the `busCfg` attribute.

   See Part III on page 101.

Specifying the WSDL

The `wsdl` attribute is the only required attribute to configure a consumer endpoint. It specifies the location of the WSDL document that defines the endpoint being exposed. The path used is relative to the top-level of the exploded service unit.

Tip

If the WSDL document defines a single service with a single endpoint, then you do not require any additional information to expose a consumer endpoint.

Example 9.1 on page 82 shows the minimal configuration for a consumer endpoint.

Example 9.1. Minimal Consumer Endpoint Configuration

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl" />

  ...
</beans>
```

For information on creating a WSDL document see Part I: on page 15.

Specifying the endpoint details

If the endpoint's WSDL document defines a single service with a single endpoint, the ESB can easily determine which endpoint to use. It will use the values from the WSDL document to specify the service name, endpoint name and interface name for the instantiated endpoint.

However, if the endpoint's WSDL document defines multiple services or if it defines multiple endpoints for a service, you will need to provide the consumer endpoint with additional information so that it can determine the proper definition to use. What information you need to provide depends on the complexity of the WSDL document. You may need to supply values for both the service name and the endpoint name, or you may only have to supply one of these values.

If the WSDL document contains more than one `service` element you will need to specify a value for the consumer's `service` attribute. The value of the consumer's `service` attribute is the QName of the WSDL `service` element that defines the desired service in the WSDL document. For example, if you wanted your endpoint to use the WidgetSalesService in the WSDL shown in Example 9.2 on page 82 you would use the configuration shown in Example 9.3 on page 83.

Example 9.2. WSDL with Two Services

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
```

```
  </service>

  <service name="WidgetInventoryService">
    <port binding="WidgetInventoryBinding" name="WidgetInventoryPort">
      <soap:address location="http://widget.inventory.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

Example 9.3. Consumer Endpoint with a Defined Service Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  service="widgets:WidgetSalesService" />

  ...
</beans>
```

If the WSDL document's service definition contains more than one endpoint, then you will need to provide a value for the consumer's `endpoint` attribute. The value of the `endpoint` attribute corresponds to the value of the WSDL `port` element's `name` attribute. For example, if you wanted your endpoint to use the WidgetEasternSalesPort in the WSDL shown in Example 9.4 on page 83 you would use the configuration shown in Example 9.5 on page 84.

Example 9.4. Service with Two Endpoints

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetWesternSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
    <port binding="WidgetSalesBinding" name="WidgetEasternSalesPort">
      <jms:address jndiConnectionFactoryName="ConnectionFactory"
                   jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
        <jms:JMSNamingProperty name="java.naming.factory.initial"
                               value="org.activemq.jndi.ActiveMQInitialContextFactory" />
        <jms:JMSNamingProperty name="java.naming.provider.url"
                               value="tcp://localhost:61616" />
      </jms:address>
```

```
    </port>
  </service>
  ...
<definitions>
```

Example 9.5. Consumer Endpoint with a Defined Endpoint Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  endpoint="WidgetEasternSalesService" />
  ...
</beans>
```

Specifying the target endpoint

The consumer endpoint will determine the target endpoint in the following manner:

1. If you explicitly specify an endpoint using both the `targetService` attribute and the `targetEndpoint` attribute, the ESB will use that endpoint.
2. If you only specify a value for the `targetService` attribute, the ESB will attempt to find an appropriate endpoint on the specified service.
3. If you specify an the name of an interface that can accept the message using the `targetInterface` attribute, the ESB will attempt to locate an endpoint that implements the specified interface and direct the messages to it.
4. If you do not use any of the target attributes, the ESB will use the values used in configuring the endpoint's service name and endpoint name to determine the target endpoint.

Example 9.6 on page 84 shows the configuration for a consumer endpoint that specifies the target endpoint to use.

Example 9.6. Consumer Endpoint Configuration Specifying a Target Endpoint

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
```

```
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  targetEndpoint="WidgetSalesTargetPort"
                  targetService="widgets:WidgetSalesTargetService" />

  ...
</beans>
```

Important

If you specify values for more than one of the target attributes, the consumer endpoint will use the most specific information.

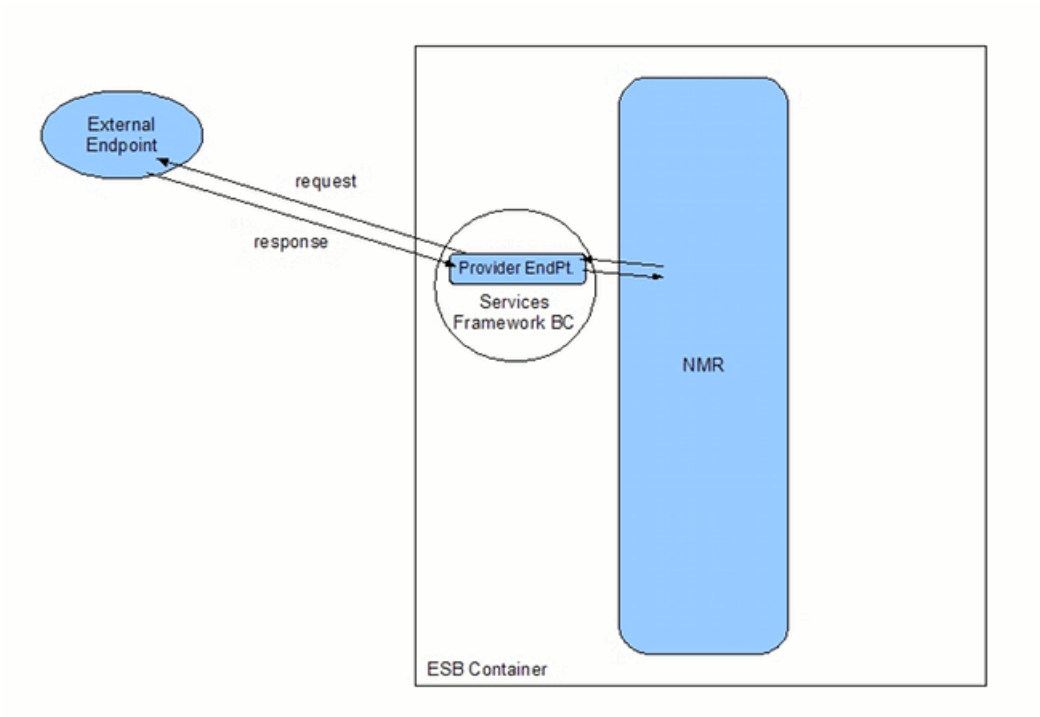
Chapter 10. Provider Endpoints

A provider endpoint sends requests to external endpoints and waits for the response. It is configured using a single XML element that specifies the WSDL document defining the endpoint.

Overview

Provider endpoints play the role of service provider from the vantage point of other endpoints running inside of the ESB. However, from outside of the ESB a provider endpoint plays the role of a consumer. As shown in Figure 10.1 on page 87, provider endpoints make requests on external endpoints. When it receives the response, the provider endpoint returns it back to the NMR.

Figure 10.1. Provider Endpoint



Important

Because provider endpoint's behave like consumers to external endpoints, you configure the runtime behavior of the transport using the consumer-specific WSDL entries.

Procedure

To configure a provider endpoint do the following:

1. Add a `provider` element to your `xbean.xml` file.
2. Add a `wsdl` attribute to the `provider` element.

   See "Specifying the WSDL" on page 89.
3. If your WSDL defines more than one service, you will need to specify a value for the `service` attribute.

   See "Specifying the endpoint details" on page 89.
4. If the service you choose defines more than one endpoint, you will need to specify a value for the `endpoint` attribute.

   See "Specifying the endpoint details" on page 89.
5. If your endpoint is going to be receiving binary attachments set its `mtomEnabled` attribute to `true`.

   See "Using MTOM to Process Binary Content" on page 93.
6. If your endpoint does not need to process the JBI wrapper set its `useJbiWrapper` attribute to `false`.

   See "Working with the JBI Wrapper" on page 95.
7. If you are using any of the advanced features, such as WS-Addressing or WS-Policy, specify a value for the `busCfg` attribute.

See Part III on page 101.

Specifying the WSDL

The `wsdl` attribute is the only required attribute to configure a provider endpoint. It specifies the location of the WSDL document that defines the endpoint being exposed. The path used is relative to the top-level of the exploded service unit.

Tip

If the WSDL document defines a single service with a single endpoint, then you do not require any additional information to expose a provider endpoint.

Example 10.1 on page 89 shows the minimal configuration for a provider endpoint.

Example 10.1. Minimal Provider Endpoint Configuration

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
  <cxfbc:provider wsdl="/wsdl/widget.wsdl" />

  ...
</beans>
```

For information on creating a WSDL document see Part I: on page 15.

Specifying the endpoint details

If the endpoint's WSDL document defines a single service with a single endpoint, the ESB can easily determine which endpoint to use. It will use the values from the WSDL document to specify the service name, endpoint name and interface name for the instantiated endpoint.

However, if the endpoint's WSDL document defines multiple services or if it defines multiple endpoints for a service, you will need to provide the consumer endpoint with additional information so that it can determine the proper definition to use. What information you need to provide depends on the complexity of the WSDL document. You may need to supply values for both the service name and the endpoint name, or you may only have to supply one of these values.

If the WSDL document contains more than one `service` element you will need to specify a value for the provider's `service` attribute. The value of the

provider's `service` attribute is the QName of the WSDL `service` element that defines the desired service in the WSDL document. For example, if you wanted your endpoint to use the WidgetInventoryService in the WSDL shown in Example 10.2 on page 90 you would use the configuration shown in Example 10.3 on page 90.

Example 10.2. WSDL with Two Services

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
  </service>

  <service name="WidgetInventoryService">
    <port binding="WidgetInventoryBinding" name="WidgetInventoryPort">
      <soap:address location="http://widget.inventory.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

Example 10.3. Provider Endpoint with a Defined Service Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:provider wsdl="/wsdl/widget.wsdl"
                  service="widgets:WidgetInventoryService" />

  ...
</beans>
```

If the WSDL document's service definition contains more than one endpoint, then you will need to provide a value for the provider's `endpoint` attribute. The value of the `endpoint` attribute corresponds to the value of the WSDL `port` element's `name` attribute. For example, if you wanted your endpoint to use the WidgetWesternSalesPort in the WSDL shown in Example 10.4 on page 91 you would use the configuration shown in Example 10.5 on page 91.

Example 10.4. Service with Two Endpoints

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetWesternSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
    <port binding="WidgetSalesBinding" name="WidgetEasternSalesPort">
      <jms:address jndiConnectionFactoryName="ConnectionFactory"
                   jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
        <jms:JMSNamingProperty name="java.naming.factory.initial"
                               value="org.activemq.jndi.ActiveMQInitialContextFactory" />
        <jms:JMSNamingProperty name="java.naming.provider.url"
                               value="tcp://localhost:61616" />
      </jms:address>
    </port>
  </service>
  ...
<definitions>
```

Example 10.5. Provider Endpoint with a Defined Endpoint Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:provider wsdl="/wsdl/widget.wsdl"
                  endpoint="WidgetWesternSalesService" />
  ...
</beans>
```

Chapter 11. Using MTOM to Process Binary Content

Enabling MTOM support allows your endpoints to consume and produce messages that contain binary data.

Overview

SOAP Message Transmission Optimization Mechanism (MTOM) specifies an optimized method for sending binary data as part of a SOAP message using the XML-binary Optimized Packaging (XOP) packages for transmitting binary data. The FUSE Services Framework binding supports the use of MTOM to send and receive binary data. MTOM support is enabled on an endpoint by endpoint basis.

Configuring an endpoint to support MTOM

As shown in Example 11.1 on page 93, you configure an endpoint to support MTOM by setting its `mtomEnabled` attribute to true.

Example 11.1. Configuring an Endpoint to Use MTOM

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ...>

  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  mtomEnabled="true" />

  ...
</beans>
```

Chapter 12. Working with the JBI Wrapper

By default, all FUSE Services Framework binding component endpoints expect SOAP messages to be inside of the JBI wrapper. You can turn off the extra processing if it is not required.

Overview

There are instances when a JBI component cannot consume a native SOAP message. For instance, SOAP headers pose difficulty for JBI components. The JBI specification defines a JBI wrapper that can be used to make SOAP messages, or any message defined in WSDL 1.1, conform to the expectations of a JBI component.

For the sake of compatibility, all endpoints exposed by the FUSE Services Framework binding component will check for the JBI wrapper. If it is present the endpoint will unwrap the messages. If you are positive that your endpoints will never receive messages that use the JBI wrapper, you can turn off the extra processing.

Turning of JBI wrapper processing

If you are sure your endpoint will not receive messages using the JBI wrapper you can set its `useJbiWrapper` attribute to `false`. This instructs the endpoint to disable the processing of the JBI wrapper. If the endpoint does receive a message that uses the JBI wrapper, it will fail to process the message and generate an error.

Example

Example 12.1 on page 95 shows a configuration fragment for configuring a consumer that does not process the JBI wrapper.

Example 12.1. Configuring a Consumer to Not Use the JBI Wrapper

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  useJbiWrapper="false" />
  ...
</beans>
```

Chapter 13. Using Message Interceptors

You can use low-level message interceptors to process messages before they are delivered to your endpoint's service implementation.

Overview

Interceptors are a low-level pieces of code that process messages as they are passed between the message channel and service's implementation. They have access to the raw message data and can be used to process SOAP action entries, process security tokens, or correlate messages. Interceptors are called in a chain and you can configure what interceptors are used at a number of points along the chain.

Configuring an endpoint's interceptor chain

A FUSE Services Framework binding component endpoint's interceptor chain has four points at which you can insert an interceptor:

in interceptors
: On consumer endpoints the *in interceptors* process messages when they are received from the external endpoint.

  On provider endpoints the *in interceptors* process messages when they are received from the NMR.

in fault interceptors
: The *in fault interceptors* process fault messages that are generated before the service implementation gets called.

out interceptors
: On consumer endpoints the *out interceptors* process messages as they pass from the service implementation to the external endpoint.

  On provider endpoints the *out interceptors* process messages as they pass from the service implementation to the NMR.

out fault interceptors
: The *out fault interceptors* process fault messages that are generated by the service implementation or by an out interceptor.

An endpoint's interceptor chain is configured using children of its `consumer` element or `provider` element. Table 13.1 on page 98 lists the elements used to configure an endpoint's interceptor chain.

Table 13.1. Elements Used to Configure an Endpoint's Interceptor Chain

Name	Description
`inInterceptors`	Specifies a list of interceptors that process incoming messages.
`inFaultInterceptors`	Specifies a list of interceptors that process incoming fault messages.
`outInterceptors`	Specifies a list of interceptors that process outgoing messages.
`outFaultInterceptors`	Specifies a list of interceptors that process outgoing fault messages.

Example 13.1 on page 98 shows a consumer endpoint configured to use the FUSE Services Framework logging interceptors.

Example 13.1. Configuring an Interceptor Chain

```
<cxfbc:consumer ...>
  ...
  <cxfbc:inInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingInInterceptor" />
  </cxfbc:inInterceptors>
  <cxfbc:outInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor" />
  </cxfbc:outInterceptors>
  <cxfbc:inFaultInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingInInterceptor" />
  </cxfbc:inFaultInterceptors>
  <cxfbc:outFaultInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor" />
  </cxfbc:outFaultInterceptors>
</cxfbc:consumer>
```

Implementing an interceptor

You can implement a custom interceptor by extending the `org.apache.cxf.phase.AbstractPhaseInterceptor` class or one of its sub-classes. Extending `AbstractPhaseInterceptor` provides you with access to the generic message handling APIs used by FUSE Services Framework. Extending one of the sub-classes provides you with more specific

APIs. For example, extending the `AbstractSoapInterceptor` class allows your interceptor to work directly with the SOAP APIs.

More information

For more information about writing FUSE Services Framework interceptors see the Apache CXF documentation[1].

[1] http://cwiki.apache.org/CXF20DOC/interceptors.html

Part III. Configuring the CXF Transport Runtimes

To take advantage of some of the features of the FUSE Services Framework transports you need to configure the FUSE Services Framework's runtime. You do this by configuring your endpoint to pass configuration information to the runtime using the `busCfg` *attribute.*

Chapter 14. Configuring the Endpoints to Load FUSE Services Framework Runtime Configuration

Both consumers and providers use the `busCfg` attribute to configure the endpoint to load FUSE Services Framework runtime configuration. Its value points to a FUSE Services Framework configuration file.

Specifying the configuration to load

You instruct an endpoint to load FUSE Services Framework runtime configuration using the `busCfg` attribute. Both the `provider` element and the `consumer` element accept this attribute. The attribute's value is the path to a file containing configuration information used by the FUSE Services Framework runtime. This path is relative to the location of the endpoint's `xbean.xml` file.

Tip

The FUSE Services Framework configuration file should be stored in the endpoint's service unit.

Each endpoint uses a separate FUSE Services Framework runtime. If your service unit creates multiple endpoints, each endpoint can load its own FUSE Services Framework runtime configuration.

Example

Example 14.1 on page 103 shows the configuraiton for a provider endpoint that loads a FUSE Services Framework configuration file called `jms-config.xml`.

Example 14.1. Provider Endpoint that Loads FUSE Services Framework Runtime Configuration

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:greeter="http://cxf.apache.org/jms_greeter"
       xmlns:test="http://test">

  <cxfbc:provider wsdl="classpath:jms_greeter.wsdl"
                  service="greeter:JMSGreeterService"
                  endpoint="GreeterPort"
                  interfaceName="greeter:JMSGreeterPortType"
                  useJBIWrapper="false"
```

```
                    busCfg="./jms-config.xml" />

</beans>
```

Chapter 15. JMS Runtime Configuration

The FUSE Services Framework JMS runtime is highly configurable.

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

- JMS session pool configuration on page 106
- Consumer specific configuration on page 107
- Provider specific configuration on page 108

JMS Session Pool Configuration

Overview

The JMS configuration allows you to specify the number of JMS sessions an endpoint will keep in a pool.

Configuration element

You use the `jms:sessionPool` element to specify the session pool configuration for a JMS endpoint. The `jms:sessionPool` element is a child of both the `jms:conduit` element and the `jms:destination` element.

The `jms:sessionPool` element's attributes, listed in Table 15.1 on page 106, specify the high and low water marks for the endpoint's JMS session pool.

Table 15.1. Attributes for Configuring the JMS Session Pool

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

Example

Example 15.1 on page 106 shows an example of configuring the session pool for a FUSE Services Framework JMS provider endpoint.

Example 15.1. JMS Session Pool Configuration

```
...
<jms:destination name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-destination>

  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:sessionPool lowWaterMark="10"
                   highWaterMark="5000" />
  ...
</jms:destination>
...
```

Consumer Specific Runtime Configuration

Overview

The JMS consumer configuration allows you to specify two runtime behaviors:

- the number of milliseconds the consumer will wait for a response.
- the number of milliseconds a request will exist before the JMS broker can remove it.

Configuration element

You configure consumer runtime behavior using the `jms:clientConfig` element. The `jms:clientConfig` element is a child of the `jms:conduit` element. It has two attributes that are used to specify the configurable runtime properties of a consumer endpoint.

Configuring the response timeout interval

You specify the interval, in milliseconds, a consumer endpoint will wait for a response before timing out using the `jms:clientConfig` element's `clientReceiveTimeout` attribute. The default timeout interval is 2000.

Configure the request time to live

You specify the interval, in milliseconds, that a request can remain unreceived before the JMS broker can delete it using the `jms:clientConfig` element's `messageTimeToLive` attribute. The default time to live interval is 0 which specifies that the request has an infinite time to live.

Example

Example 15.2 on page 107 shows a configuration fragment that sets the consumer endpoint's request lifetime to 500 milliseconds and its timeout value to 500 milliseconds.

Example 15.2. JMS Consumer Endpoint Runtime Configuration

```
...
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:clientConfig clientReceiveTimeout="500"
                    messageTimeToLive="500" />
  ...
</jms:conduit>
...
```

Provider Specific Runtime Configuration

Overview

The provider specific configuration allows you to specify to runtime behaviors:

- the amount of time a response message can remain unreceived before the JMS broker can delete it.
- the client identifier used when creating and accessing durable subscriptions.

Configuration element

You configure provider runtime behavior using the `jms:serverConfig` element. The `jms:serverConfig` element is a child of the `jms:destination` element. It has two attributes that are used to specify the configurable runtime properties of a provider endpoint.

Configuring the response time to live

The `jms:serverConfig` element's `messageTimeToLive` attribute specifies the amount of time, in milliseconds, that a response can remain unread before the JMS broker is allowed to delete it. The default is `0` which specifies that the message can live forever.

Configuring the durable subscriber identifier

The `jms:serverConfig` element's `durableSubscriptionClientId` attribute specifies the client identifier the endpoint uses to create and access durable subscriptions.

Example

Example 15.3 on page 108 shows a configuration fragment that sets the provider endpoint's response lifetime to 500 milliseconds and its durable subscription client identifier to `jms-test-id`.

Example 15.3. Provider Endpoint Runtime Configuration

```
...
<jms:destination name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-destination">

  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:serverConfig messageTimeToLive="500"
                    durableSubscriptionClientId="jms-test-id" />
  ...
</jms:destination>
...
```

Chapter 16. Configuring the Jetty Runtime

The Jetty instance used to implement the HTTP server runtime has a number of configurable properties.

Overview

The Jetty runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Jetty runtime.

Namespace

The elements used to configure the Jetty runtime are defined in the namespace `http://cxf.apache.org/transports/http-jetty/configuration`. It is commonly referred to using the prefix httpj. In order to use the Jetty configuration elements you must add the lines shown in Example 16.1 on page 109 to the `beans` element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 16.1. Jetty Runtime Configuration Namespace

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

The engine-factory element

The `httpj:engine-factory` element is the root element used to configure the Jetty runtime used by an application. It has a single required attribute, `bus`, whose value is the name of the `Bus` that manages the Jetty instances being configured.

Tip

The value is typically `cxf` which is the name of the default `Bus` instance.

The `httpj:engine-factory` element has three children that contain the information used to configure the HTTP ports instantiated by the Jetty runtime factory. The children are described in Table 16.1 on page 110.

Table 16.1. Elements for Configuring a Jetty Runtime Factory

Element	Description
`httpj:engine`	Specifies the configuration for a particular Jetty runtime instance. See "The `engine` element" on page 110.
`httpj:identifiedTLSServerParameters`	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, `id`, that specifies a unique identifier by which the property set can be referred.
`httpj:identifiedThreadingParameters`	Specifies a reusable set of properties for controlling a Jetty instance's thread pool. It has a single attribute, `id`, that specifies a unique identifier by which the property set can be referred. See "Configuring the thread pool" on page 111.

The engine element

The `httpj:engine` element is used to configure specific instances of the Jetty runtime. It has a single attribute, `port`, that specifies the number of the port being managed by the Jetty instance.

Tip

You can specify a value of `0` for the `port` attribute. Any threading properties specified in an `httpj:engine` element with its `port` attribute set to `0` are used as the configuration for all Jetty listeners that are not explicitly configured.

Each `httpj:engine` element can have two children: one for configuring security properties and one for configuring the Jetty instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent `httpj:engine-factory` element.

The child elements used to provide the configuration properties are described in Table 16.2 on page 111.

Table 16.2. Elements for Configuring a Jetty Runtime Instance

Element	Description
`httpj:tlsServerParameters`	Specifies a set of properties for configuring the security used for the specific Jetty instance.
`httpj:tlsServerParametersRef`	Refers to a set of security properties defined by a `identifiedTLSServerParameters` element. The `id` attribute provides the id of the referred `identifiedTLSServerParameters` element.
`httpj:threadingParameters`	Specifies the size of the thread pool used by the specific Jetty instance. See "Configuring the thread pool" on page 111.
`httpj:threadingParametersRef`	Refers to a set of properties defined by a `identifiedThreadingParameters` element. The `id` attribute provides the id of the referred `identifiedThreadingParameters` element.

Configuring the thread pool

You can configure the size of a Jetty instance's thread pool by either:

- Specifying the size of the thread pool using a `identifiedThreadingParameters` element in the `engine-factory` element. You then refer to the element using a `threadingParametersRef` element.
- Specifying the size of the of the thread pool directly using a `threadingParameters` element.

The `threadingParameters` has two attributes to specify the size of a thread pool. The attributes are described in Table 16.3 on page 111.

Note

The `httpj:identifiedThreadingParameters` element has a single child `threadingParameters` element.

Table 16.3. Attributes for Configuring a Jetty Thread Pool

Attribute	Description
`minThreads`	Specifies the minimum number of threads available to the Jetty instance for processing requests.

Attribute	Description
maxThreads	Specifies the maximum number of threads available to the Jetty instance for processing requests.

Example

Example 16.2 on page 112 shows a configuration fragment that configures a Jetty instance on port number 9001.

Example 16.2. Configuring a Jetty Instance

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

  <httpj:engine-factory bus="cxf">
    <httpj:identifiedTLSServerParameters id="secure">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                      file="certs/cherry.jks"/>
      </sec:keyManagers>
    </httpj:identifiedTLSServerParameters>

    <httpj:engine port="9001">
      <httpj:tlsServerParametersRef id="secure" />
      <httpj:threadingParameters minThreads="5"
                                 maxThreads="15" />
    </httpj:engine>
  </httpj:engine-factory>
 </beans>
```

Chapter 17. Deploying WS-Addressing

FUSE Services Framework supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the FUSE Services Framework runtime environment.

Introduction to WS-Addressing

Overview

WS-Addressing is a specification that allows services to communicate addressing information in a transport neutral way. It consists of two parts:

- A structure for communicating a reference to a Web service endpoint
- A set of Message Addressing Properties (MAP) that associate addressing information with a particular message

Supported specifications

FUSE Services Framework supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification.

Further information

For detailed information on WS-Addressing, see the 2004/08 submission at http://www.w3.org/Submission/ws-addressing/.

WS-Addressing Interceptors

Overview

In FUSE Services Framework, WS-Addressing functionality is implemented as interceptors. The FUSE Services Framework runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the WS-Addressing interceptors are added to the application's interceptor chain, any WS-Addressing information included with a message is processed.

WS-Addressing Interceptors

The WS-Addressing implementation consists of two interceptors, as described in Table 17.1 on page 115.

Table 17.1. WS-Addressing Interceptors

Interceptor	Description
`org.apache.cxf.ws.addressing.MAPAggregator`	A logical interceptor responsible for aggregating the Message Addressing Properties (MAPs) for outgoing messages.
`org.apache.cxf.ws.addressing.soap.MAPCodec`	A protocol-specific interceptor responsible for encoding and decoding the Message Addressing Properties (MAPs) as SOAP headers.

Enabling WS-Addressing

Overview

To enable WS-Addressing the WS-Addressing interceptors must be added to the inbound and outbound interceptor chains. This is done in one of the following ways:

- FUSE Services Framework Features
- RMAssertion and WS-Policy Framework
- Using Policy Assertion in a WS-Addressing Feature

Adding WS-Addressing as a Feature

WS-Addressing can be enabled by adding the WS-Addressing feature to the client and the server configuration as shown in Example 17.1 on page 116 and Example 17.2 on page 116 respectively.

Example 17.1. client.xml—Adding WS-Addressing Feature to Client Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans ht
tp://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:client ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:client>
</beans>
```

Example 17.2. server.xml—Adding WS-Addressing Feature to Server Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans http://www.spring
framework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Configuring WS-Addressing Attributes

Overview

The FUSE Services Framework WS-Addressing feature element is defined in the namespace http://cxf.apache.org/ws/addressing. It supports the two attributes described in Table 17.2 on page 118.

Table 17.2. WS-Addressing Attributes

Attribute Name	Value
`allowDuplicates`	A boolean that determines if duplicate MessageIDs are tolerated. The default setting is `true`.
`usingAddressingAdvisory`	A boolean that indicates if the presence of the `UsingAddressing` element in the WSDL is advisory only; that is, its absence does not prevent the encoding of WS-Addressing headers.

Configuring WS-Addressing attributes

Configure WS-Addressing attributes by adding the attribute and the value you want to set it to the WS-Addressing feature in your server or client configuration file. For example, the following configuration extract sets the `allowDublicates` attribute to `false` on the server endpoint:

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing"
...>
    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing allowDuplicates="false"/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Using a WS-Policy assertion embedded in a feature

In Example 17.3 on page 118 an addressing policy assertion to enable non-anonymous responses is embedded in the `policies` element.

Example 17.3. Using the Policies to Configure WS-Addressing

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:policy="http://cxf.apache.org/policy-config"
```

```
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
        xmlns:jaxws="http://cxf.apache.org/jaxws"
        xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
                    createdFromAPI="true">
        <jaxws:features>
            <policy:policies>
                <wsp:Policy xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                    <wsam:Addressing>
                        <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                        </wsp:Policy>
                    </wsam:Addressing>
                </wsp:Policy>
            <policy:policies>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Chapter 18. Enabling Reliable Messaging

FUSE Services Framework supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in FUSE Services Framework.

Introduction to WS-RM

Overview

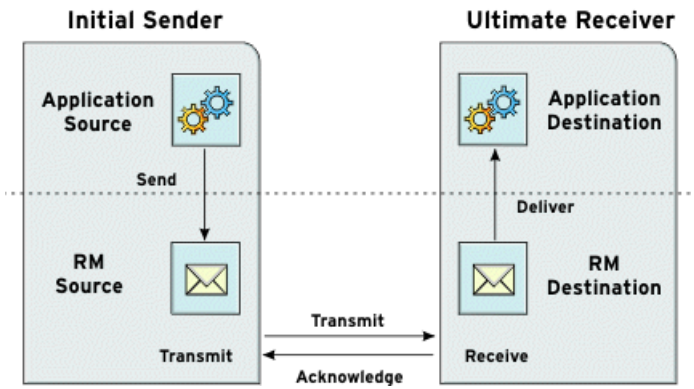
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

For example, WS-RM can be used to ensure that the correct messages have been delivered across a network exactly once, and in the correct order.

How WS-RM works

WS-RM ensures the reliable delivery of messages between a source and a destination endpoint. The source is the initial sender of the message and the destination is the ultimate receiver, as shown in Figure 18.1 on page 122.

Figure 18.1. Web Services Reliable Messaging



The flow of WS-RM messages can be described as follows:

1. The RM source sends a `CreateSequence` protocol message to the RM destination. This contains a reference for the endpoint that receives acknowledgements (the `wsrm:AcksTo` endpoint).

2. The RM destination sends a `CreateSequenceResponse` protocol message back to the RM source. This message contains the sequence ID for the RM sequence session.

3. The RM source adds an RM `Sequence` header to each message sent by the application source. This header contains the sequence ID and a unique message ID.
4. The RM source transmits each message to the RM destination.
5. The RM destination acknowledges the receipt of the message from the RM source by sending messages that contain the RM `SequenceAcknowledgement` header.
6. The RM destination delivers the message to the application destination in an exactly-once-in-order fashion.
7. The RM source retransmits a message that it has not yet received an acknowledgement.

   The first retransmission attempt is made after a base retransmission interval. Successive retransmission attempts are made, by default, at exponential back-off intervals or, alternatively, at fixed intervals. For more details, see "Configuring WS-RM" on page 130.

This entire process occurs symmetrically for both the request and the response message; that is, in the case of the response message, the server acts as the RM source and the client acts as the RM destination.

WS-RM delivery assurances

WS-RM guarantees reliable message delivery in a distributed environment, regardless of the transport protocol used. Either the source or the destination endpoint logs an error if reliable delivery can not be assured.

Supported specifications

FUSE Services Framework supports the 2005/02 version of the WS-RM specification, which is based on the WS-Addressing 2004/08 specification.

Further information

For detailed information on WS-RM, see the specification at http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf.

WS-RM Interceptors

Overview

In FUSE Services Framework, WS-RM functionality is implemented as interceptors. The FUSE Services Framework runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the application's interceptor chain includes the WS-RM interceptors, the application can participate in reliable messaging sessions. The WS-RM interceptors handle the collection and aggregation of the message chunks. They also handle all of the acknowledgement and retransmission logic.

FUSE Services Framework WS-RM Interceptors

The FUSE Services Framework WS-RM implementation consists of four interceptors, which are described in Table 18.1 on page 124.

Table 18.1. FUSE Services Framework WS-ReliableMessaging Interceptors

Interceptor	Description
`org.apache.cxf.ws.rm.RMOutInterceptor`	Deals with the logical aspects of providing reliability guarantees for outgoing messages. Responsible for sending the `CreateSequence` requests and waiting for their `CreateSequenceResponse` responses. Also responsible for aggregating the sequence properties—ID and message number—for an application message.
`org.apache.cxf.ws.rm.RMInInterceptor`	Responsible for intercepting and processing RM protocol messages and `SequenceAcknowledgement` messages that are piggybacked on application messages.
`org.apache.cxf.ws.rm.soap.RMSoapInterceptor`	Responsible for encoding and decoding the reliability properties as SOAP headers.

Interceptor	Description
`org.apache.cxf.ws.rm.RetransmissionInterceptor`	Responsible for creating copies of application messages for future resending.

Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, when intercepting the first application message on the outbound interceptor chain, the `RMOutInterceptor` sends a `CreateSequence` request and waits to process the original application message until it receives the `CreateSequenceResponse` response. In addition, the WS-RM interceptors add the sequence headers to the application messages and, on the destination side, extract them from the messages. It is not necessary to make any changes to your application code to make the exchange of messages reliable.

For more information on how to enable WS-RM, see "Enabling WS-RM" on page 126.

Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default FUSE Services Framework attempts to maximize the lifetime of a sequence, thus reducing the overhead incurred by the out-of-band WS-RM protocol messages. To enforce the use of a separate sequence per application message configure the WS-RM source's sequence termination policy (setting the maximum sequence length to `1`).

For more information on configuring WS-RM behavior, see "Configuring WS-RM" on page 130.

Enabling WS-RM

Overview

To enable reliable messaging, the WS-RM interceptors must be added to the interceptor chains for both inbound and outbound messages and faults. Because the WS-RM interceptors use WS-Addressing, the WS-Addressing interceptors must also be present on the interceptor chains.

You can ensure the presence of these interceptors in one of two ways:

- Explicitly, by adding them to the dispatch chains using Spring beans
- Implicitly, using WS-Policy assertions, which cause the FUSE Services Framework runtime to transparently add the interceptors on your behalf.

Spring beans—explicitly adding interceptors

To enable WS-RM add the WS-RM and WS-Addressing interceptors to the FUSE Services Framework bus, or to a consumer or service endpoint using Spring bean configuration. This is the approach taken in the WS-RM sample that is found in the *InstallDir*/samples/ws_rm directory. The configuration file, ws-rm.cxf, shows the WS-RM and WS-Addressing interceptors being added one-by-one as Spring beans (see Example 18.1 on page 126).

Example 18.1. Enabling WS-RM Using Spring Beans

```
<?xml version="1.0" encoding="UTF-8"?>
❶<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
   beans http://www.springframework.org/schema/beans/spring-beans.xsd">
❷   <bean id="mapAggregator" class="org.apache.cxf.ws.addressing.MAPAggregator"/>
   <bean id="mapCodec" class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
❸   <bean id="rmLogicalOut" class="org.apache.cxf.ws.rm.RMOutInterceptor">
        <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
        <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmCodec" class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>
   <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
❹       <property name="inInterceptors">
          <list>
              <ref bean="mapAggregator"/>
              <ref bean="mapCodec"/>
              <ref bean="rmLogicalIn"/>
```

```
                <ref bean="rmCodec"/>
            </list>
        </property>
❺       <property name="inFaultInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
❻       <property name="outInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
❼       <property name="outFaultInterceptors">
            <list>
                <ref bean="mapAggregator">
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
    </bean>
</beans>
```

The code shown in Example 18.1 on page 126 can be explained as follows:

❶ A FUSE Services Framework configuration file is a Spring XML file. You must include an opening Spring `beans` element that declares the namespaces and schema files for the child elements that are encapsulated by the `beans` element.

❷ Configures each of the WS-Addressing interceptors—`MAPAggregator` and `MAPCodec`. For more information on WS-Addressing, see "Deploying WS-Addressing" on page 113.

❸ Configures each of the WS-RM interceptors—`RMOutInterceptor`, `RMInInterceptor`, and `RMSoapInterceptor`.

❹ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.

❺ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.

❻ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.
❼ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound faults.

WS-Policy framework—implicitly adding interceptors

The WS-Policy framework provides the infrastructure and APIs that allow you to use WS-Policy. It is compliant with the November 2006 draft publications of the Web Services Policy 1.5—Framework[1] and Web Services Policy 1.5—Attachment[2] specifications.

To enable WS-RM using the FUSE Services Framework WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. Example 18.2 on page 128 shows a reference bean nested within a `jaxws:feature` element. The reference bean specifies the `AddressingPolicy`, which is defined as a separate element within the same configuration file.

Example 18.2. Configuring WS-RM using WS-Policy

```
<jaxws:client>
    <jaxws:features>
      <ref bean="AddressingPolicy"/>
    </jaxws:features>
</jaxws:client>
<wsp:Policy wsu:Id="AddressingPolicy" xmlns:wsam="http://www.w3.org/2007/02/address
ing/metadata">
    <wsam:Addressing>
      <wsp:Policy>
        <wsam:NonAnonymousResponses/>
      </wsp:Policy>
    </wsam:Addressing>
</wsp:Policy>
```

2. Add a reliable messaging policy to the `wsdl:service` element—or any other WSDL element that can be used as an attachment point for policy or policy reference elements—to your WSDL file, as shown in Example 18.3 on page 129.

[1] http://www.w3.org/TR/2006/WD-ws-policy-20061117/
[2] http://www.w3.org/TR/2006/WD-ws-policy-attach-20061117/

Example 18.3. Adding an RM Policy to Your WSDL File

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

Configuring WS-RM

You can configure WS-RM by:

- Setting FUSE Services Framework-specific attributes that are defined in the FUSE Services Framework WS-RM manager namespace, http://cxf.apache.org/ws/rm/manager.
- Setting standard WS-RM policy attributes that are defined in the `http://schemas.xmlsoap.org/ws/2005/02/rm/policy` namespace.

Configuring FUSE Services Framework-Specific WS-RM Attributes

Overview

To configure the FUSE Services Framework-specific attributes, use the `rmManager` Spring bean. Add the following to your configuration file:

- The http://cxf.apache.org/ws/rm/manager namespace to your list of namespaces.
- An `rmManager` Spring bean for the specific attribute that your want to configure.

Example 18.4 on page 131 shows a simple example.

Example 18.4. Configuring FUSE Services Framework-Specific WS-RM Attributes

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
     xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager http://cxf.apache.org/schemas/configuration/wsrm-man
ager.xsd">
...
<wsrm-mgr:rmManager>
<!--
  ...Your configuration goes here
-->
</wsrm-mgr:rmManager>
```

Children of the rmManager Spring bean

Table 18.2 on page 131 shows the child elements of the `rmManager` Spring bean, defined in the http://cxf.apache.org/ws/rm/manager namespace.

Table 18.2. Children of the rmManager Spring Bean

Element	Description
`RMAssertion`	An element of type RMAssertion
`deliveryAssurance`	An element of type DeliveryAssuranceType that describes the delivery assurance that should apply
`sourcePolicy`	An element of type SourcePolicyType that allows you to configure details of the RM source

Element	Description
`destinationPolicy`	An element of type DestinationPolicyType that allows you to configure details of the RM destination

Example

For an example, see "Maximum unacknowledged messages threshold" on page 139.

Configuring Standard WS-RM Policy Attributes

Overview

You can configure standard WS-RM policy attributes in one of the following ways:

- "RMAssertion in rmManager Spring bean"
- "Policy within a feature"
- "WSDL file"
- "External attachment"

WS-Policy RMAssertion Children

Table 18.3 on page 133 shows the elements defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace:

Table 18.3. Children of the WS-Policy RMAssertion Element

Name	Description
`InactivityTimeout`	Specifies the amount of time that must pass without receiving a message before an endpoint can consider an RM sequence to have been terminated due to inactivity.
`BaseRetransmissionInterval`	Sets the interval within which an acknowledgement must be received by the RM Source for a given message. If an acknowledgement is not received within the time set by the `BaseRetransmissionInterval`, the RM Source will retransmit the message.
`ExponentialBackoff`	Indicates the retransmission interval will be adjusted using the commonly known exponential backoff algorithm (Tanenbaum). For more information, see *Computer Networks*, Andrew S. Tanenbaum, Prentice Hall PTR, 2003.
`AcknowledgementInterval`	In WS-RM, acknowledgements are sent on return messages or sent stand-alone. If a return message is not available to send an acknowledgement, an RM Destination can wait for up to the acknowledgement interval before sending a

Name	Description
	stand-alone acknowledgement. If there are no unacknowledged messages, the RM Destination can choose not to send an acknowledgement.

More detailed reference information

For more detailed reference information, including descriptions of each element's sub-elements and attributes, please refer to http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd.

RMAssertion in rmManager Spring bean

You can configure standard WS-RM policy attributes by adding an `RMAssertion` within a FUSE Services Framework `rmManager` Spring bean. This is the best approach if you want to keep all of your WS-RM configuration in the same configuration file; that is, if you want to configure FUSE Services Framework-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 18.5 on page 134 shows:

- A standard WS-RM policy attribute, `BaseRetransmissionInterval`, configured using an `RMAssertion` within an `rmManager` Spring bean.
- An FUSE Services Framework-specific RM attribute, `intraMessageThreshold`, configured in the same configuration file.

Example 18.5. Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
    <wsrm-mgr:destinationPolicy>
        <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
    </wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>
```

Policy within a feature

You can configure standard WS-RM policy attributes within features, as shown in Example 18.6 on page 135.

Example 18.6. Configuring WS-RM Attributes as a Policy within a Feature

```
<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd">
    <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort" created
FromAPI="true">
        <jaxws:features>
               <wsp:Policy>
                   <wsrm:RMAssertion xmlns:wsrm="http://schem
as.xmlsoap.org/ws/2005/02/rm/policy">
                     <wsrm:AcknowledgementInterval Milliseconds="200" />
                   </wsrm:RMAssertion>
                   <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/address
ing/metadata">
                      <wsp:Policy>
                           <wsam:NonAnonymousResponses/>
                      </wsp:Policy>
                   </wsam:Addressing>
              </wsp:Policy>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

WSDL file

If you use the WS-Policy framework to enable WS-RM, you can configure standard WS-RM policy attributes in a WSDL file. This is a good approach if you want your service to interoperate and use WS-RM seamlessly with consumers deployed to other policy-aware Web services stacks.

For an example, see "WS-Policy framework—implicitly adding interceptors" on page 128 where the base retransmission interval is configured in the WSDL file.

External attachment

You can configure standard WS-RM policy attributes in an external attachment file. This is a good approach if you cannot, or do not want to, change your WSDL file.

Example 18.7 on page 136 shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 18.7. Configuring WS-RM in an External Attachment

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy" xmlns:wsa="ht
tp://www.w3.org/2005/08/addressing">
    <wsp:PolicyAttachment>
        <wsp:AppliesTo>
           <wsa:EndpointReference>
                <wsa:Address>http://localhost:9020/SoapContext/GreeterPort</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:Policy>
            <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsp:Policy/>
            </wsam:Addressing>
          <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">

                <wsrmp:BaseRetransmissionInterval Milliseconds="30000"/>
            </wsrmp:RMAssertion>
        </wsp:Policy>
    </wsp:PolicyAttachment>
</attachments>/
```

WS-RM Configuration Use Cases

Overview

This subsection focuses on configuring WS-RM attributes from a use case point of view. Where an attribute is a standard WS-RM policy attribute, defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace, only the example of setting it in an `RMAssertion` within an `rmManager` Spring bean is shown. For details of how to set such attributes as a policy within a feature; in a WSDL file, or in an external attachment, see "Configuring Standard WS-RM Policy Attributes" on page 133.

The following use cases are covered:

- "Base retransmission interval"
- "Exponential backoff for retransmission"
- "Acknowledgement interval"
- "Maximum unacknowledged messages threshold"
- "Maximum length of an RM sequence"
- "Message delivery assurance policies"

Base retransmission interval

The `BaseRetransmissionInterval` element specifies the interval at which an RM source retransmits a message that has not yet been acknowledged. It is defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd schema file. The default value is 3000 milliseconds.

Example 18.8 on page 137 shows how to set the WS-RM base retransmission interval.

Example 18.8. Setting the WS-RM Base Retransmission Interval

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
```

```
</wsrm-mgr:rmManager>
</beans>
```

Exponential backoff for retransmission

The `ExponentialBackoff` element determines if successive retransmission attempts for an unacknowledged message are performed at exponential intervals.

The presence of the `ExponentialBackoff` element enables this feature. An exponential backoff ratio of `2` is used by default.

Example 18.9 on page 138 shows how to set the WS-RM exponential backoff for retransmission.

Example 18.9. Setting the WS-RM Exponential Backoff Property

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff="4"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Acknowledgement interval

The `AcknowledgementInterval` element specifies the interval at which the WS-RM destination sends asynchronous acknowledgements. These are in addition to the synchronous acknowledgements that it sends on receipt of an incoming message. The default asynchronous acknowledgement interval is `0` milliseconds. This means that if the `AcknowledgementInterval` is not configured to a specific value, acknowledgements are sent immediately (that is, at the first available opportunity).

Asynchronous acknowledgements are sent by the RM destination only if both of the following conditions are met:

- The RM destination is using a non-anonymous `wsrm:acksTo` endpoint.
- The opportunity to piggyback an acknowledgement on a response message does not occur before the expiry of the acknowledgement interval.

Example 18.10 on page 139 shows how to set the WS-RM acknowledgement interval.

Example 18.10. Setting the WS-RM Acknowledgement Interval

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Maximum unacknowledged messages threshold

The `maxUnacknowledged` attribute sets the maximum number of unacknowledged messages that can accrue per sequence before the sequence is terminated.

Example 18.11 on page 139 shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 18.11. Setting the WS-RM Maximum Unacknowledged Message Threshold

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Maximum length of an RM sequence

The `maxLength` attribute sets the maximum length of a WS-RM sequence. The default value is `0`, which means that the length of a WS-RM sequence is unbound.

When this attribute is set, the RM endpoint creates a new RM sequence when the limit is reached, and after receiving all of the acknowledgements for the previously sent messages. The new message is sent using a newsequence.

Example 18.12 on page 139 shows how to set the maximum length of an RM sequence.

Example 18.12. Setting the Maximum Length of a WS-RM Message Sequence

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
```

```
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Message delivery assurance policies

You can configure the RM destination to use the following delivery assurance policies:

- `AtMostOnce` — The RM destination delivers the messages to the application destination only once. If a message is delivered more than once an error is raised. It is possible that some messages in a sequence may not be delivered.
- `AtLeastOnce` — The RM destination delivers the messages to the application destination at least once. Every message sent will be delivered or an error will be raised. Some messages might be delivered more than once.
- `InOrder` — The RM destination delivers the messages to the application destination in the order that they are sent. This delivery assurance can be combined with the `AtMostOnce` or `AtLeastOnce` assurances.

Example 18.13 on page 140 shows how to set the WS-RM message delivery assurance.

Example 18.13. Setting the WS-RM Message Delivery Assurance Policy

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:deliveryAssurance>
        <wsrm-mgr:AtLeastOnce />
    </wsrm-mgr:deliveryAssurance>
</wsrm-mgr:reliableMessaging>
</beans>
```

Configuring WS-RM Persistence

Overview

The FUSE Services Framework WS-RM features already described in this chapter provide reliability for cases such as network failures. WS-RM persistence provides reliability across other types of failure such as an RM source or an RM destination crash.

WS-RM persistence involves storing the state of the various RM endpoints in persistent storage. This enables the endpoints to continue sending and receiving messages when they are reincarnated.

FUSE Services Framework enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, FUSE Services Framework includes Derby for out-of-the-box deployment. In addition, the persistent store is also exposed using a Java API.

Important

WS-RM persistence is supported for oneway calls only, and it is disabled by default.

How it works

FUSE Services Framework WS-RM persistence works as follows:

- At the RM source endpoint, an outgoing message is persisted before transmission. It is evicted from the persistent store after the acknowledgement is received.
- After a recovery from crash, it recovers the persisted messages and retransmits until all the messages have been acknowledged. At that point, the RM sequence is closed.
- At the RM destination endpoint, an incoming message is persisted, and upon a successful store, the acknowledgement is sent. When a message is successfully dispatched, it is evicted from the persistent store.

- After a recovery from a crash, it recovers the persisted messages and dispatches them. It also brings the RM sequence to a state where new messages are accepted, acknowledged, and delivered.

Enabling WS-persistence

To enable WS-RM persistence, you must specify the object implementing the persistent store for WS-RM. You can develop your own or you can use the JDBC based store that comes with FUSE Services Framework.

The configuration shown in Example 18.14 on page 142 enables the JDBC-based store that comes with FUSE Services Framework.

Example 18.14. Configuration for the Default WS-RM Persistence Store

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

Configuring WS-persistence

The JDBC-based store that comes with FUSE Services Framework supports the properties shown in Table 18.4 on page 142.

Table 18.4. JDBC Store Properties

Attribute Name	Type	Default Setting
driverClassName	String	`org.apache.derby.jdbc.EmbeddedDriver`
userName	String	null
passWord	String	null
url	String	`jdbc:derby:rmdb;create=true`

The configuration shown in Example 18.15 on page 142 enables the JDBC-based store that comes with FUSE Services Framework, while setting the driverClassName and url to non-default values.

Example 18.15. Configuring the JDBC Store for WS-RM Persistence

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName" value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

Appendix A. Consumer Endpoint Properties

The attributes described in Table A.1 on page 143 are used to configure a consumer endpoint.

Table A.1. Consumer Endpoint Attributes

Name	Type	Description	Required
`wsdl`	String	Specifies the location of the WSDL defining the endpoint.	yes
`service`	QName	Specifies the service name of the proxied endpoint. This corresponds to WSDL `service` element's `name` attribute.	no[a]
`endpoint`	String	Specifies the endpoint name of the proxied endpoint. This corresponds to WSDL `port` element's `name` attribute.	no[b]
`interfaceName`	QName	Specifies the interface name of the proxied endpoint. This corresponds to WSDL `portType` element's `name` attribute.	no
`targetService`	QName	Specifies the service name of the target endpoint.	no (defaults to the value of the `service` attribute)
`targetEndpoint`	String	Specifies the endpoint name of the target endpoint.	no (defaults to the value of the `endpoint` attribute)
`targetInterfaceName`	QName	Specifies the interface name of the target endpoint.	no
`busCfg`	String	Specifies the location of a spring configuration file used for FUSE Services Framework bus initialization.	no
`mtomEnabled`	boolean	Specifies if MTOM / attachment support is enabled.	no (defaults to `false`)
`useJbiWrapper`	boolean	Specifies if the JBI wrapper is sent in the body of the message.	no (defaults to `true`)

Name	Type	Description	Required
`timeout`	int	Specifies the number of seconds to wait for a response.	no (defaults to `10`

[a]If the WSDL defining the service has more than one `service` element, this attribute is required.

[b]If the service being used defines more than one endpoint, this attribute is required.

Appendix B. Provider Endpoint Properties

The attributes described in Table B.1 on page 145 are used to configure a provider endpoint.

Table B.1. Provider Endpoint Attributes

Attribute	Type	Description	Required
`wsdl`	String	Specifies the location of the WSDL defining the endpoint.	yes
`service`	QName	Specifies the service name of the exposed endpoint.	no[a]
`endpoint`	String	Specifies the endpoint name of the exposed endpoint.	no[b]
`locationURI`	URI	Specifies the URL of the target service.	no[cd]
`interfaceName`	QName	Specifies the interface name of the exposed jbi endpoint.	no
`busCfg`	String	Specifies the location of the spring configuration file used for FUSE Services Framework bus initialization.	no
`mtomEnabled`	boolean	Specifies if MTOM / attachment support is enabled.	no (defaults to `false`)
`useJbiWrapper`	boolean	Specifies if the JBI wrapper is sent in the body of the message.	no (defaults to `true`)

[a] If the WSDL defining the service has more than one `service` element, this attribute is required.

[b] If the service being used defines more than one endpoint, this attribute is required.

[c] If specified, the value of this attribute overrides the HTTP address specified in the WSDL contract.

[d] This attribute is ignored if the endpoint uses a JMS address in the WSDL.

Appendix C. Using the Maven JBI Tooling

Packaging application components so that they conform the JBI specification is a cumbersome job. FUSE ESB includes tooling that automates the process of packaging you applications and creating the required JBI descriptors.

FUSE ESB provides a Maven plug-in and a number of Maven archetypes that make developing, packaging, and deploying JBI artifacts easier. The tooling provides you with a number of benefits. These benefits include:

- automatic generation of JBI descriptors
- dependency checking

Because FUSE ESB only allows you to deploy service assemblies, you will need to do the following when using the Maven JBI tooling:

1. Set up a top-level project on page 148 to build all of the service units and the final service assembly.
2. Create a project for each of your service units. on page 153.
3. Create a project for the service assembly on page 159.

Setting Up a FUSE ESB JBI Project

Overview

When working with the FUSE ESB JBI Maven tooling, you will want to create a top-level project that can build all of the service units and package them into a service assembly. Using a top-level project for this purpose has several advantages. It allows you to control the dependencies for all of the parts of an application in a central location. It limits the number of times you need to specify the proper repositories to load. It also gives you a central location from which to build and deploy the application.

The top-level project is responsible for assembling the application. It will use the Maven assembly plug-in and list your service units and the service assembly as modules of the project.

Directory structure

Your top-level project will contain the following directories:

- a source directory containing the information needed by the Maven assembly plug-in
- a directory to hold the service assembly project
- at least one directory containing a service unit project

Tip

You will need a project folder for each service unit that is to be included in the generated service assembly.

Setting up the Maven tools

In order to use the FUSE ESB JBI Maven tooling, you add the elements shown in Example C.1 on page 148 to your top-level POM file.

Example C.1. POM Elements for Using FUSE ESB Tooling

```
...
<pluginRepositories>
  <pluginRepository>
    <id>fusesource.m2</id>
    <name>FUSE Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/maven2</url>
    <snapshots>
      <enabled>false</enabled>
```

```
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>
<repositories>
  <repository>
    <id>fusesource.m2</id>
    <name>FUSE Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/maven2</url>
    <snapshots>
       <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name>FUSE Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/maven2-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>
  ...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.servicemix.tooling</groupId>
      <artifactId>jbi-maven-plugin</artifactId>
      <version>servicemix-version</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
  ...
```

These elements point Maven to the correct repositories to download the FUSE ESB Maven tooling and load the plug-in that implements the tooling.

Listing the subprojects

Your top-level POM lists all of the service units and the service assembly that will be generated as modules. The modules are contained in a `modules`

element. The `modules` element contains one `module` element for each service unit in the assembly. You will also need a `module` element for the service assembly.

The modules should be listed in the order in which they are built. This means that the service assembly module should be listed after all of the service unit modules.

Example JBI Project POM

Example C.2 on page 150 shows a top-level pom for a project that contains a single service unit.

Example C.2. Top-Level POM for a FUSE ESB JBI Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.widgets</groupId>
    <artifactId>demos</artifactId>
    <version>1.0</version>
  </parent>

  <groupId>com.widgets.demo</groupId>
  <artifactId>cxf-wsdl-first</artifactId>
  <name>CXF WSDL Fisrt Demo</name>
  <packaging>pom</packaging>

  <pluginRepositories> ❶
    <pluginRepository>
      <id>fusesource.m2</id>
      <name>FUSE Open Source Community Release Repository</name>
      <url>http://repo.fusesource.com/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <enabled>true</enabled>
      </releases>
    </pluginRepository>
  </pluginRepositories>
  <repositories>
    <repository>
      <id>fusesource.m2</id>
      <name>FUSE Open Source Community Release Repository</name>
```

```
    <url>http://repo.fusesource.com/maven2</url>
    <snapshots>
       <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name>FUSE Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/maven2-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>

<modules> ❷
  <module>wsdl-first-cxfse-su</module>
   <module>wsdl-first-cxf-sa</module>
</modules>

<build>
  <plugins>
    <plugin> ❸
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
       <version>2.1</version>
       <inherited>false</inherited>
         <executions>
           <execution>
              <id>src</id>
              <phase>package</phase>
              <goals>
                <goal>single</goal>
              </goals>
              <configuration>
                <descriptors>
                  <descriptor>src/main/assembly/src.xml</descriptor>
                </descriptors>
               </configuration>
            </execution>
         </executions>
       </plugin>
       <plugin> ❹
```

```
            <groupId>org.apache.servicemix.tooling</groupId>
            <artifactId>jbi-maven-plugin</artifactId>
            <extensions>true</extensions>
          </plugin>
    </plugins>
  </build>
</project>
```

The POM shown in Example C.2 on page 150 does the following:

❶ Configures Maven to use the FUSE repositories for loading the FUSE ESB plug-ins.

❷ Lists the sub-projects used for this application. The `wsdl-first-cxfse-su` module is the module for the service unit. The `wsdl-first-cxf-sa` module is the module for the service assembly

❸ Configures the Maven assembly plug-in.

❹ Loads the FUSE ESB JBI plug-in.

A Service Unit Project

Overview

Each service unit in the service assembly needs to be its own project. These projects are placed at the same level as the service assembly project. The contents of a service unit's project depends on the component at which the service unit is targeted. At a minimum, a service unit project will contain a POM and an XML configuration file.

Seeding a project using a Maven artifact

FUSE ESB provides Maven artifacts for a number of service unit types. You can use them to seed a project with the **smx-arch** command. As shown in Example C.3 on page 153, the **smx-arch** command takes three arguments. The `groupId` value and the `artifactId` values correspond to the project's group ID and artifact ID.

Example C.3. Maven Archetype Command for Service Units

`smx-arch` su *`suArchetypeName`* ["-DgroupId=*`my.group.id`*"] ["-DartifactId=*`my.artifact.id`*"]

Important

The double quotes(") are required when using the `-DgroupId` argument and the `-DartifactId` argument.

The *`suArchetypeName`* specifies the type of service unit to seed. Table C.1 on page 153 lists the possible values and describes what type of project will be seeded.

Table C.1. Service Unit Archetypes

Name	Description
camel	Creates a project for using the FUSE Mediation Router service engine.
cxf-se	Creates a project for developing a Java-first service using the FUSE Services Framework service engine.
cxf-se-wsdl-first	Creates a project for developing a WSDL-first service using the FUSE Services Framework service engine.
cxf-bc	Creates an endpoint project targeted at the FUSE Services Framework binding component.

Name	Description
http-consumer	Creates a consumer endpoint project targeted at the HTTP binding component.
http-provider	Creates a provider endpoint project targeted at the HTTP binding component.
jms-consumer	Creates a consumer endpoint project targeted at the JMS binding component. See *Using the JMS Binding Component*.
jms-provider	Creates a provider endpoint project targeted at the JMS binding component. See *Using the JMS Binding Component*.
file-poller	Creates a polling (consumer) endpoint project targeted at the file binding component. See "Using Poller Endpoints" in *Using the File Binding Component*.
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component. See "Using Sender Endpoints" in *Using the File Binding Component*.
ftp-poller	Creates a polling (consumer) endpoint project targeted at the FTP binding component.
ftp-sender	Creates a sender (provider) endpoint project targeted at the FTP binding component.
jsr181-annotated	Creates a project for developing an annotated Java service to be run by the JSR181 service engine. [a]
jsr181-wsdl-first	Creates a project for developing a WSDL generated Java service to be run by the JSR181 service engine.[a]
saxon-xquery	Create a project for executing xquery statements using the Saxon service engine.
saxon-xslt	Create a project for executing XSLT scripts using the Saxon service engine.
eip	Creates a project for using the EIP service engine. [b]
lwcontainer	Create a project for deploying functionality into the lightweight container. [c]
bean	Creates a project for deploying a POJO to be executed by the bean service engine.

Name	Description
ode	Create a project for deploying a BPEL process into the ODE service engine.

[a]The JSR181 has been deprecated. The FUSE Services Framework service engine has superseded it.

[b]The EIP service engine has been deprecated. The FUSE Mediation Router service engine has superseded it.

[c]The lightweight container has been deprecated.

Contents of a project

The contents of your service unit project change from service unit to service unit. Different components require different configuration. Some components, such as the FUSE Services Framework service engine, require that you include Java classes.

At a minimum, a service unit project will contain two things:

- a POM file that configures the JBI plug-in to create a service unit
- an XML configuration file stored in `src/main/resources`

  For many of the components the XML configuration file is called `xbean.xml`. The FUSE Mediation Router component uses a file called `camel-context.xml`.

Configuring the Maven plug-in

You configure the Maven plug-in to package the results of the project build as a service unit by changing the value of the project's `packaging` element to `jbi-service-unit` as shown in Example C.4 on page 155.

Example C.4. Configuring the Maven Plug-in to Build a Service Unit

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  ...
  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxfse-wsdl-first-su</artifactId>
  <name>CXF WSDL Fisrt Demo :: SE Service Unit</name>
  <packaging>jbi-service-unit</packaging>
```

```
  ...
</project>
```

Specifying the target components

In order to properly fill in the metadata required for packaging a service unit, the Maven plug-in needs to be told what component, or components, the service unit is targeting. If your service unit only has a single component dependency, you can specify it in one of two ways:

- list the targeted component as a dependency
- add a `componentName` property specifying the targeted component

If your service unit has more than one component dependency you need to configure the project as follows:

1. Add a `componentName` property specifying the targeted component.
2. Add the remaining components to the list dependencies.

Example C.5 on page 156 shows configuration for a service unit targeting the FUSE Services Framework binding component.

Example C.5. Specifying the Target Components for a Service Unit

```
...
<dependencies>
  <dependency>
    <groupId>org.apache.servicemix</groupId>
    <artifactId>servicemix-cxf-bc</artifactId>
    <version>3.3.1.0-fuse</version>[1]
  </dependency>
>/dependencies>
...
```

The advantage of using the Maven dependency mechanism is that it allows Maven to check if the targeted component is deployed in the container. If one of the components is not deployed, FUSE ESB will not hold off deploying the service unit until all of the required components are deployed.

[1] You replace this with the version of FUSE Services Framework you are using.

Tip

A message identifying the missing component(s) is typically written to the log.

If your service unit targets is not available as a Maven artifact, you can specify the targeted component using the `componentName` element. This element is added to the standard Maven properties block and specifies the name of a targeted component. Example C.6 on page 157 shows how to use the `componentName` element to specify the target component.

Example C.6. Specifying the Target Components for a Service Unit

```
...
<properties>
  <componentName>servicemix-bean</componentName>
>/properties>
...
```

When you use the `componentName` element Maven does not check to see if the component is installed. Maven also cannot download the required component.

Example

Example C.7 on page 157 shows the POM file for a project building a service unit targeted to the FUSE Services Framework binding component.

Example C.7. POM for a Service Unit Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent> ❶
        <groupId>com.widgets.demo</groupId>
        <artifactId>cxf-wsdl-first</artifactId>
        <version>1.0</version>
    </parent>

  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxfse-wsdl-first-su</artifactId>
  <name>CXF WSDL Fisrt Demo :: SE Service Unit</name>
  <packaging>jbi-service-unit</packaging> ❷
```

```
  <dependencies> ❸
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-cxf-bc</artifactId>
      <version>3.3.1.0-fuse</version>
    </dependency>
  >/dependencies>

  <build>
    <plugins>
      <plugin> ❹
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

The POM in Example C.7 on page 157 does the following:

❶ Specifies that it is a part of the top-level project described in Example C.2 on page 150.

❷ Specifies that this project builds a service unit.

❸ Specifies that the service unit targets the FUSE Services Framework binding component.

❹ Specifies that the FUSE ESB Maven plug-in is to be used.

A Service Assembly Project

Overview

FUSE ESB requires that all service units be bundled into a service assembly before they can be deployed into a container. The FUSE ESB Maven plug-in will collect all of the service units to be bundled and the metadata needed for packaging. It will then build a service assembly containing the service units.

Seeding a project using a Maven artifact

FUSE ESB provides a Maven artifact for seeding a service assembly project. You can seed a project with the **smx-arch** command. As shown in Example C.8 on page 159, the **smx-arch** command takes two arguments. The `groupId` value and the `artifactId` values correspond to the project's group ID and artifact ID.

Example C.8. Maven Archetype Command for Service Assemblies

`smx-arch` sa ["-DgroupId=*my.group.id*"] ["-DartifactId=*my.artifact.id*"]

Important

The double quotes(") are required when using the `-DgroupId` argument and the `-DartifactId` argument.

Contents of a project

A service assembly project typically only contains the POM file used by Maven.

Configuring the Maven plug-in

You configure the Maven plug-in to package the results of the project build as a service assembly by changing the value of the project's `packaging` element to `jbi-service-assembly` as shown in Example C.9 on page 159.

Example C.9. Configuring the Maven Plug-in to Build a Service Assembly

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  ...
  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxf-wsdl-first-sa</artifactId>
  <name>CXF WSDL Fisrt Demo :: Service Assembly</name>
  <packaging>jbi-service-assembly</packaging>
```

```
  ...
</project>
```

Specifying the target components

The Maven plug-in needs to be told what service units are being bundled into the service assembly. You do this by specifying the service units as a dependencies using the standard Maven `dependencies` element. You add a `dependency` child element for each service unit. Example C.10 on page 160 shows configuration for a service assembly that bundles two service units.

Example C.10. Specifying the Target Components for a Service Unit

```
...
<dependencies>
  <dependency>
    <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
    <artifactId>cxfse-wsdl-first-su</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
    <artifactId>cxfbc-wsdl-first-su</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
...
```

Example

Example C.11 on page 160 shows the POM file for a project building a service assembly.

Example C.11. POM for a Service Assembly Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent> ❶
        <groupId>com.widgets.demo</groupId>
        <artifactId>cxf-wsdl-first</artifactId>
        <version>1.0</version>
    </parent>

  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
```

```
  <artifactId>cxf-wsdl-first-sa</artifactId>
  <name>CXF WSDL Fisrt Demo ::  Service Assemby</name>
  <packaging>jbi-service-assembly</packaging> ❷

  <dependencies> ❸
    <dependency>
      <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
      <artifactId>cxfse-wsdl-first-su</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
      <artifactId>cxfbc-wsdl-first-su</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin> ❹
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

The POM in Example C.11 on page 160 does the following:

❶ Specifies that it is a part of the top-level project described in Example C.2 on page 150.

❷ Specifies that this project builds a service assembly.

❸ Specifies the service units the service assembly bundles.

❹ Specifies that the FUSE ESB Maven plug-in is to be used.

Appendix D. Using the Maven OSGi Tooling

Manually creating a bundle, or a collection of bundles, for a large project can be cumbersome. The Maven bundle plug-in makes the job easier by automating the process and providing a number of shortcuts for specifying the contents of the bundle manifest.

The FUSE ESB OSGi tooling uses the Maven bundle plug-in[1] from Apache Felix. The bundle plug-in is based on the **bnd**[2] tool from Peter Kriens. It automates the construction of OSGi bundle manifests by introspecting the contents of the classes being packaged in the bundle. Using the knowledge of the classes contained in the bundle, the plug-in can calculate the proper values to populate the Import-Packages and the Export-Package properties in the bundle manifest. The plug-in also has default values that are used for other required properties in the bundle manifest.

To use the bundle plug-in you will need to do the following:

1. Add the bundle plug-in to your project's POM file.
2. Configure the plug-in to correctly populate your bundle's manifest.

[1] http://cwiki.apache.org/FELIX/apache-felix-maven-bundle-plugin-bnd.html
[2] http://www.aqute.biz/Code/Bnd

Setting Up a FUSE ESB OSGi Project

Overview

A Maven project for building an OSGi bundle can be a simple single level project. It does not require any sub-projects. It does, however, require that you do the following:

1. Add the bundle plug-in to your POM.
2. Instruct Maven to package the results as an OSGi bundle.

Tip

There are several Maven archetypes to set up your project with the appropriate settings.

Directory structure

A project that constructs an OSGi bundle can be a single level project. It only requires that you have a top-level POM file and a `src` folder. As in all Maven projects, you place all Java source code in the `src/java` folder. You place any non-Java resources into the `src/resources` folder.

Non-Java resources include Spring configuration files, JBI endpoint configuration files, WSDL contracts, etc.

Note

FUSE ESB OSGi projects that use FUSE Services Framework, FUSE Mediation Router, or another Spring configured bean also include a `beans.xml` file located in the `src/resources/META-INF/spring` folder.

Adding a bundle plug-in

Before you can use the bundle plug-in you must add a dependency on Apache Felix. After you add the dependency, you can add the bundle plug-in to the plug-in portion of the POM.

Example D.1 on page 165 shows the POM entries required to add the bundle plug-in to your project.

Example D.1. Adding an OSGi Bundle Plug-in to a POM

```
...
<dependencies>
  <dependency> ❶
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin> ❷
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName> ❸
          <Import-Package>*,org.apache.camel.osgi</Import-Package> ❹
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package> ❺
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example D.1 on page 165 do the following:

❶ Adds the dependency on Apache Felix.

❷ Adds the bundle plug-in to your project.

❸ Configures the plug-in to use the project's artifact ID as the bundle's symbolic name.

❹ Configures the plug-in to include all Java packages imported by the bundled classes and also import the `org.apache.camel.osgi` package.

❺ Configures the plug-in to bundle the listed class, but not include them in the list of exported packages.

Note

You should edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see "Configuring a Bundle Plug-in" on page 169.

Activating a bundle plug-in

To instruct Maven to use the bundle plug-in, you instruct it to package the results of the project as a bundle. You do this by setting the POM file's `packaging` element to `bundle`.

Useful Maven archetypes

There are several Maven archetypes to generate a project that is preconfigured to use the bundle plug-in:

- "Spring OSGi archetype"
- "FUSE Services Framework code-first archetype"
- "FUSE Services Framework wsdl-first archetype"
- "FUSE Mediation Router archetype"

Spring OSGi archetype

The Spring OSGi archetype creates a generic project for building an OSGi project using Spring DM:

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.springframework.osgi
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=1.12
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

FUSE Services Framework code-first archetype

The FUSE Services Framework code-first archetype creates a project for building a service from Java:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

FUSE Services Framework wsdl-first archetype

The FUSE Services Framework wsdl-first archetype creates a project for creating a service from WSDL:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

FUSE Mediation Router archetype

The FUSE Mediation Router archetype creates a project for building a route that is deployed into FUSE ESB:

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-camel-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Configuring a Bundle Plug-in

Overview

A bundle plug-in requires very little information to function. All of the required properties have default settings to generate a valid OSGi bundle.

While you can create a valid bundle using just the default values, you will likely want to modify some of the values. You can specify most of the properties inside the plug-in's `instructions` element.

Configuration properties

Some of the commonly used configuration properties are:

- Bundle-SymbolicName
- Bundle-Name
- Bundle-Version
- Export-Package
- Private-Package
- Import-Package

Setting a bundle's symbolic name

By default, the bundle plug-in sets the value for the Bundle-SymbolicName property to *groupId*+ "." + *artifactId*, with the following exceptions:

- If *groupId* has only one section (no dots), the first package name with classes is returned.

  For example, if the groupId is `commons-logging:commons-logging`, the bundle's symbolic name is `org.apache.commons.logging`.

- If *artifactId* is equal to the last section of *groupId*, then *groupId* is used.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven`, the bundle's symbolic name is `org.apache.maven`.

- If *artifactId* starts with the last section of *groupId*, that portion is removed.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven-core`, the bundle's symbolic name is `org.apache.maven.core`.

To specify your own value for the bundle's symbolic name, add a `Bundle-SymbolicName` child in the plug-in's `instructions` element, as shown in Example D.2.

Example D.2. Setting a Bundle's Symbolic Name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to `${pom.name}`.

To specify your own value for the bundle's name, add a `Bundle-Name` child to the plug-in's `instructions` element, as shown in Example D.3.

Example D.3. Setting a Bundle's Name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's version

By default, a bundle's version is set to `${pom.version}`. Any dashes (`-`) are replaced with dots (`.`).

To specify your own value for the bundle's version, add a `Bundle-Version` child to the plug-in's `instructions` element, as shown in Example D.4.

Example D.4. Setting a Bundle's Version

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying exported packages

By default, the OSGi manifest's `Export-Package` list is populated by all of the packages in your project's class path that match the pattern *`Bundle-SymbolicName`*`.*`. These packages are also included in the bundle.

Important

If you use a `Private-Package` element in your plug-in configuration and do not specify a list of packages to export, the default behavior is to assume that no packages are exported. Only the packages listed in the `Private-Package` element are included in the bundle and none of them are exported.

The default behavior can result in very large packages as well as exporting packages that should be kept private. To change the list of exported packages you can add a `Export-Package` child to the plug-in's `instructions` element.

The `Export-Package` element specifies a list of packages that are to be included in the bundle and be exported. The package names can be specified using the `*` wildcard. For example, the entry `com.fuse.demo.*`, includes all packages on the project's classpath that start with `com.fuse.demo`.

You can specify packages to be excluded be prefixing the entry with `!`. For example, the entry, `!com.fuse.demo.private`, excludes the package `com.fuse.demo.private`.

When attempting to exclude packages, the order of entries in the list is important. The list is processed in order from the start and subsequent contradicting entries are ignored.

For example, to include all packages starting with `com.fuse.demo` except the package `com.fuse.demo.private`, list the packages in the following way:

```
!com.fuse.demo.private,com.fuse.demo.*
```

However, if you list the packages as:

```
com.fuse.demo.*,!com.fuse.demo.private
```

Then `com.fuse.demo.private` is included in the bundle because it matches the first pattern.

Specifying private packages

By default, all packages included in a bundle are exported. You can include packages in the bundle without exporting them. To specify a list of packages to be included in a bundle, but not exported, add a `Private-Package` child to the plug-in's `instructions` element.

The `Private-Package` element works similarly to the `Export-Package` element. You specify a list of packages to be included in the bundle. The bundle plug-in uses the list to find all classes on the project's classpath to be included in the bundle. These packages are packaged in the bundle, but not exported.

Important

If a package matches an entry in both the `Private-Package` element and the `Export-Package` element, the `Export-Package` element takes precedent. The package is added to the bundle and exported.

Example D.5 shows the configuration for including a private package in a bundle

Example D.5. Including a Private Package in a Bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying imported packages

By default, the bundle plug-in populates the OSGi manifest's Import-Package property with a list of all the packages referred to by the contents of the bundle and not included in the bundle.

While the default behavior is typically sufficient for most projects, you might find instances where you want to import packages that are not automatically added to the list. The default behavior can also result in unwanted packages being imported.

To specify a list of packages to be imported by the bundle, add a `Import-Package` child to the plug-in's `instructions` element. The syntax for the package list is the same as for both the `Export-Package` and `Private-Package` elements.

Important

When you use the `Import-Package` element, the plug-in does not automatically scan the bundle's contents to determine if there are any required imports. To ensure that the contents of the bundle are scanned, you must place `*` as the last entry in the package list.

Example D.6 shows the configuration for including a private package in a bundle

Example D.6. Specifying the Packages Imported by a Bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws,
         javax.wsdl,
         org.apache.cxf.bus,
         org.apache.cxf.bus.spring,
         org.apache.cxf.bus.resource,
         org.apache.cxf.configuration.spring,
         org.apache.cxf.resource,
         org.springframework.beans.factory.config,
         *
     </Import-Package>
     ...
   </instructions>
  </configuration>
</plugin>
```

More information

For more information on configuring a bundle plug-in, see:

- Apache Felix documentation[3]
- Peter Kriens' aQute Software Consultancy web site[4]

[3] http://cwiki.apache.org/FELIX/apache-felix-maven-bundle-plugin-bnd.html
[4] http://www.aqute.biz/Code/Bnd

Index

I

J

L

M

N

O

P

R

S

T

U

W

X