Red Hat JBoss Fuse 6.0

Using the Apache CXF Binding Component

Implementing Web services

Last Updated: 2017-10-13

Red Hat JBoss Fuse 6.0 Using the Apache CXF Binding Component

Implementing Web services

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide provides an overview of the JBI CXF binding component; describes how to define endpoints in WSDL, how to configure and package endpoints, and how to configure the CXF runtime; describes the properties of consumer and provider endpoints; and describes how to use the Maven tooling.

Table of Contents

CHAPTER 1. INTRODUCTION TO THE APACHE CXF BINDING COMPONENT

Abstract

The Apache CXF binding component allows you to create SOAP/HTTP and SOAP/JMS endpoints.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The Apache CXF binding component provides connectivity to external endpoints using either SOAP/HTTP or SOAP/JMS. The endpoints are defined using WSDI files that contain Apache CXF specific extensions for defining the transport. In addition, you can add Apache CXF-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 Chapter 9, *Consumer Endpoints*.

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 Chapter 10, *Provider Endpoints*.

KEY FEATURES

The Apache CXF 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 APACHE CXF BINDING COMPONENT

Using the Apache CXF binding component to expose SOAP endpoints usually involves the following steps:

1. Defining the contract for your endpoint in WSDL.

   See Part I, “Defining an Endpoint in WSDL” .

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

   See Part II, “Configuring and Packaging Endpoints” .

3. Bundling the service unit into a service assembly for deployment into the Red Hat JBoss Fuse container.

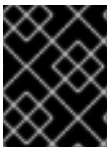
MORE INFORMATION

For more information about using Apache CXF to create SOAP endpoints see the Apache CXF documentation.

PART I. DEFINING AN ENDPOINT IN WSDL

Abstract

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.



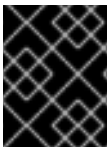
IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

CHAPTER 2. INTRODUCING WSDL CONTRACTS

Abstract

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.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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.

2.1. 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:

- A logical 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.

2.2. 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 Chapter 3, *Defining Logical Data Units*.
- **`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 Chapter 4, *Defining Logical Messages Used by a Service*.
- **`portType`** — A collection of **`operation`** elements describing the logical interface of a service. For information about defining port types see Chapter 5, *Defining Your Logical Interfaces*.
- **`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 the section called "Operations".
- **`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.

2.3. 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

Abstract

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

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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. Chapter 4, *Defining Logical Messages Used by a Service* discusses the second step.

3.1. 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 Apache CXF.

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.

3.2. 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 Apache CXF 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, "Schema entry for a WSDL contract" , 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 Section 3.4.1, "Defining data structures".
5. For each array, define the data type using a **complexType** element. See Section 3.4.2, "Defining arrays".
6. For each complex type that is derived from a simple type, define the data type using a **simpleType** element. See Section 3.4.4, "Defining types by restriction".
7. For each enumerated type, define the data type using a **simpleType** element. See Section 3.4.5, "Defining enumerated types".

8. For each element, define it using an **element** element. See Section 3.5, "Defining elements".

3.3. 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, "Defining an element with a simple type".

Example 3.2. Defining an element with a simple type

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

Supported XSD simple types

Apache CXF 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

3.4. 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.

3.4.1. 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 the section called "Complex type varieties".
3. Each of the fields of the defined structure are defined in **element** elements that are grandchildren of the **complexType** element. See the section called "Defining the parts of a structure".

For example, the structure shown in Example 3.3, "Simple Structure" 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, "A complex type" shows one possible XML Schema mapping for the structure shown in Example 3.3, "Simple Structure".

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, "Complex type descriptor elements" 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.

Element	Complex Type Behavior
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, "A complex type" generates a message containing two elements: **name** and **age**.

If the structure is defined using a **choice** element, as shown in Example 3.5, "Simple complex choice type", 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, "Simple complex type with occurrence constraints" .

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, "Simple complex type with minOccurs set to zero" . 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, "Complex type with an attribute" 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.

3.4.2. Defining arrays

Apache CXF 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, "Complex type array".

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, "Syntax for a SOAP array derived using wsdl:arrayType".

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, "Definition of a SOAP array", 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, “Syntax for a SOAP array derived using an element” .

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**.

3.4.3. 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, “A complex type” 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, “Type defined by extension” .

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>
```

3.4.4. 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 the section called "Specifying the base type".
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the **restriction** element. See the section called "Defining the restrictions".

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, "Using int as the base type".

Example 3.14. Using int as the 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, "SSN simple type description" 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>
```

3.4.5. 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, "Syntax for an enumeration".

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, "widgetSize enumeration", 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>
```

3.5. 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

Abstract

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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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 Section 3.4, "Defining complex data types".
- 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, "Part data type attributes".

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*.

Attribute	Description
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, “Reused part” .

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, “personalInfo lookup method” .

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, “RPC WSDL message definitions”.

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, “Wrapped document WSDL message definitions”.

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

Abstract

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

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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 the section called “Port types”.
2. Create an **operation** element for each operation defined in the interface. See the section called “Operations”.
3. For each operation, specify the messages used to represent the operation’s parameter list, return type, and exceptions. See the section called “Operation messages”.

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, "Operation message elements" .

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, "Attributes of the input and output elements" .

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, "personalInfo lookup interface".

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, "personalInfo lookup port type" .

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

Abstract

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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

6.1. 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, "SOAP 1.1 Port Element" 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, "SOAP 1.2 Port Element" 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, "HTTP Port Element" 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>
```

6.2. 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, “HTTP Consumer WSDL Element's Namespace” 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, “HTTP Consumer Configuration Attributes” .

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.

Attribute	Description
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: • **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**.

Attribute	Description
`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. • **`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 the section called "Consumer Cache Control Directives".
`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.

Attribute	Description
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, Section 6.4, "Using the HTTP Transport in Decoupled Mode". 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) • **SOCKS**

Consumer Cache Control Directives

Table 6.2, "**http-conf:client** Cache Control Directives" 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, “WSDL to Configure an HTTP Consumer Endpoint” 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>
```

6.3. 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, "HTTP Provider WSDL Element's Namespace" 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, "HTTP Service Provider Configuration Attributes".

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.

Attribute	Description
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 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 the section called "Service Provider Cache Control Directives".
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. Apache CXF 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, "**http-conf:server** Cache Control Directives" 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 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, "WSDL to Configure an HTTP Service Provider Endpoint" 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>
```

6.4. 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 the section called "Configuring an endpoint to use WS-Addressing".

2. Configure the consumer to use a decoupled endpoint.

   See the section called "Configuring the consumer".

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

   See the section called "Configuring an endpoint to use WS-Addressing".

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, “Activating WS-Addressing using WSDL”.

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, “Activating WS-Addressing using a Policy”.

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, “Configuring a Consumer to Use a Decoupled HTTP Endpoint” shows the configuration for setting up the endpoint defined in Example 6.8, “Activating WS-Addressing using WSDL” 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"
```

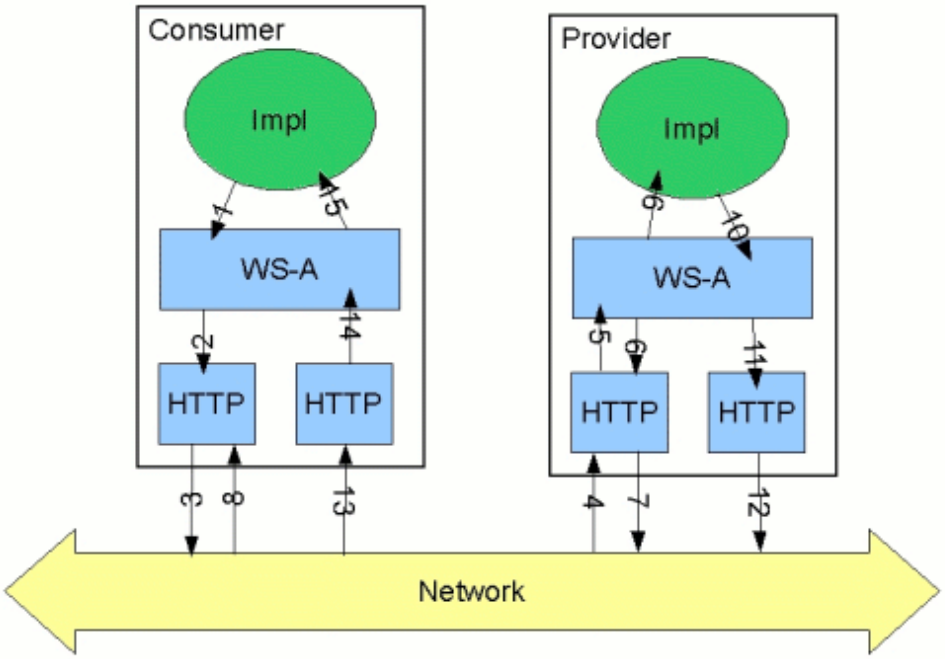
```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:http="http://cxf.apache.org/transports/http/configuration"

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, “Message Flow in for a Decoupled HTTP Transport” 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

Abstract

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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

7.1. USING SOAP/JMS

Apache CXF implements the W3C standard SOAP/JMS transport. This standard is intended to provide a more robust alternative to SOAP/HTTP services. Apache CXF applications using this transport should be able to interoperate with applications that also implement the SOAP/JMS standard. The transport is configured directly in an endpoint's WSDL.

7.1.1. Basic configuration

Overview

The SOAP over JMS protocol is defined by the World Wide Web Consortium(W3C) as a way of providing a more reliable transport layer to the customary SOAP/HTTP protocol used by most services. The Apache CXF implementation is fully compliant with the specification and should be compatible with any framework that is also compliant.

This transport uses JNDI to find the JMS destinations. When an operation is invoked, the request is packaged as a SOAP message and sent in the body of a JMS message to the specified destination.

To use the SOAP/JMS transport:

1. Specify that the transport type is SOAP/JMS.
2. Specify the target destination using a JMS URI.
3. Optionally, configure the JNDI connection.
4. Optionally, add additional JMS configuration.

Specifying the JMS transport type

You configure a SOAP binding to use the JMS transport when specifying the WSDL binding. You set the **`soap:binding`** element's **`transport`** attribute to **`http://www.w3.org/2010/soapjms/`**. Example 7.1, “SOAP over JMS binding specification” shows a WSDL binding that uses SOAP/JMS.

Example 7.1. SOAP over JMS binding specification

```
<wsdl:binding ... >
  <soap:binding style="document"
```

```
                transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

Specifying the target destination

You specify the address of the JMS target destination when specifying the WSDL port for the endpoint. The address specification for a SOAP/JMS endpoint uses the same **soap:address** element and attribute as a SOAP/HTTP endpoint. The difference is the address specification. JMS endpoints use a JMS URI as defined in the URI Scheme for JMS 1.0 . Example 7.2, "JMS URI syntax" shows the syntax for a JMS URI.

Example 7.2. JMS URI syntax

```
jms:variant:destination?options
```

Table 7.1, "JMS URI variants" describes the available variants for the JMS URI.

Table 7.1. JMS URI variants

Variant	Description
jndi	Specifies that the destination is a JNDI name for the target destination. When using this variant, you must provide the configuration for accessing the JNDI provider.
topic	Specifies that the destination is the name of the topic to be used as the target destination. The string provided is passed into **Session.createTopic()** to create a representation of the destination.
queue	Specifies that the destination is the name of the queue to be used as the target destination. The string provided is passed into **Session.createQueue()** to create a representation of the destination.

The *options* portion of a JMS URI are used to configure the transport and are discussed in Section 7.1.2, "JMS URIs".

Example 7.3, "SOAP/JMS endpoint address" shows the WSDL port entry for a SOAP/JMS endpoint whose target destination is looked up using JNDI.

Example 7.3. SOAP/JMS endpoint address

```
<wsdl:port ... >
  ...
  <soap:address
```

```
location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

For working with SOAP/JMS services in Java see chapter "Using SOAP over JMS" in "Developing Applications Using JAX-WS".

Configuring JNDI and the JMS transport

The SOAP/JMS provides several ways to configure the JNDI connection and the JMS transport:

- Using the JMS URI
- Using WSDL extensions

7.1.2. JMS URIs

Overview

When using SOAP/JMS, a JMS URI is used to specify the endpoint's target destination. The JMS URI can also be used to configure JMS connection by appending one or more options to the URI. These options are detailed in the IETF standard, URI Scheme for Java Message Service 1.0 . They can be used to configure the JNDI system, the reply destination, the delivery mode to use, and other JMS properties.

Syntax

As shown in Example 7.2, “JMS URI syntax” , you can append one or more options to the end of a JMS URI by separating them from the destination's address with a question mark(**?**). Multiple options are separated by an ampersand(**&**). Example 7.4, “Syntax for JMS URI options” shows the syntax for using multiple options in a JMS URI.

Example 7.4. Syntax for JMS URI options

```
jmsAddress?option1=value1&option2=value2&...optionN=valueN
```

JMS properties

Table 7.2, “JMS properties settable as URI options” shows the URI options that affect the JMS transport layer.

Table 7.2. JMS properties settable as URI options

Property	Default	Description

Property	Default	Description
`deliveryMode`	`PERSISTENT`	Specifies whether to use JMS `PERSISTENT` or `NON_PERSISTENT` message semantics. In the case of `PERSISTENT` delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas `NON_PERSISTENT` messages are kept in memory only.
`replyToName`		Explicitly specifies the reply destination to appear in the `JMSReplyTo` header. Setting this property is recommended for applications that have request-reply semantics because the JMS provider will assign a temporary reply queue if one is not explicitly set. The value of this property has an interpretation that depends on the variant specified in the JMS URI: • `jndi` variant—the JNDI name of the destination • `queue` or `topic` variants—the actual name of the destination
`priority`	`4`	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`timeToLive`	`0`	Time (in milliseconds) after which the message will be discarded by the JMS provider. A value of `0` represents an infinite lifetime (the default).

JNDI properties

Table 7.3, "JNDI properties settable as URI options" shows the URI options that can be used to configure JNDI for this endpoint.

Table 7.3. JNDI properties settable as URI options

Property	Description
jndiConnectionFactoryName	Specifies the JNDI name of the JMS connection factory.
jndiInitialContextFactory	Specifies the fully qualified Java class name of the JNDI provider (which must be of **javax.jms.InitialContextFactory** type). Equivalent to setting the **java.naming.factory.initial** Java system property.
jndiURL	Specifies the URL that initializes the JNDI provider. Equivalent to setting the **java.naming.provider.url** Java system property.

Additional JNDI properties

The properties, **java.naming.factory.initial** and **java.naming.provider.url**, are standard properties, which are required to initialize any JNDI provider. Sometimes, however, a JNDI provider might support custom properties in addition to the standard ones. In this case, you can set an arbitrary JNDI property by setting a URI option of the form **jndi-*PropertyName***.

For example, if you were using SUN's LDAP implementation of JNDI, you could set the JNDI property, **java.naming.factory.control**, in a JMS URI as shown in Example 7.5, “Setting a JNDI property in a JMS URI”.

Example 7.5. Setting a JNDI property in a JMS URI

```
jms:queue:FOO.BAR?jndi-
java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
```

Example

If the JMS provider is *not* already configured, it is possible to provide the requisite JNDI configuration details in the URI using options (see Table 7.3, “JNDI properties settable as URI options”). For example, to configure an endpoint to use the Apache ActiveMQ JMS provider and connect to the queue called **test.cxf.jmstransport.queue**, use the URI shown in Example 7.6, “JMS URI that configures a JNDI connection”.

Example 7.6. JMS URI that configures a JNDI connection

```
jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
?
jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContex
tFactory
&jndiConnectionFactoryName=ConnectionFactory
&jndiURL=tcp://localhost:61616
```

7.1.3. WSDL extensions

Overview

You can specify the basic configuration of the JMS transport by inserting WSDL extension elements into the contract, either at binding scope, service scope, or port scope. The WSDL extensions enable you to specify the properties for bootstrapping a JNDI **`InitialContext`**, which can then be used to look up JMS destinations. You can also set some properties that affect the behavior of the JMS transport layer.

SOAP/JMS namespace

the SOAP/JMS WSDL extensions are defined in the **`http://www.w3.org/2010/soapjms/`** namespace. To use them in your WSDL contracts add the following setting to the **`wsdl:definitions`** element:

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

WSDL extension elements

Table 7.4, “SOAP/JMS WSDL extension elements” shows all of the WSDL extension elements you can use to configure the JMS transport.

Table 7.4. SOAP/JMS WSDL extension elements

Element	Default	Description
`soapjms:jndiInitialContextFactory`		Specifies the fully qualified Java class name of the JNDI provider. Equivalent to setting the **`java.naming.factory.initial`** Java system property.
`soapjms:jndiURL`		Specifies the URL that initializes the JNDI provider. Equivalent to setting the **`java.naming.provider.url`** Java system property.
`soapjms:jndiContextParameter`		Enables you to specify an additional property for creating the JNDI **`InitialContext`**. Use the **`name`** and **`value`** attributes to specify the property.
`soapjms:jndiConnectionFactoryName`		Specifies the JNDI name of the JMS connection factory.

Element	Default	Description
`soapjms:deliveryMode`	**PERSISTENT**	Specifies whether to use JMS **PERSISTENT** or **NON_PERSISTENT** message semantics. In the case of **PERSISTENT** delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas **NON_PERSISTENT** messages are kept in memory only.
`soapjms:replyToName`		Explicitly specifies the reply destination to appear in the **JMSReplyTo** header. Setting this property is recommended for SOAP invocations that have request-reply semantics. If this property is not set the JMS provider allocates a temporary queue with an automatically generated name. The value of this property has an interpretation that depends on the variant specified in the JMS URI, as follows: • **jndi** variant—the JNDI name of the destination. • **queue** or **topic** variants—the actual name of the destination.
`soapjms:priority`	**4**	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`soapjms:timeToLive`	**0**	Time, in milliseconds, after which the message will be discarded by the JMS provider. A value of **0** represents an infinite lifetime.

Configuration scopes

The WSDL elements placement in the WSDL contract effect the scope of the configuration changes on the endpoints defined in the contract. The SOAP/JMS WSDL elements can be placed as children of either the `wsdl:binding` element, the `wsdl:service` element, or the `wsdl:port` element. The parent of the SOAP/JMS elements determine which of the following scopes the configuration is placed into.

Binding scope

You can configure the JMS transport at the *binding scope* by placing extension elements inside the

wsdl:binding element. Elements in this scope define the default configuration for all endpoints that use this binding. Any settings in the binding scope can be overridden at the service scope or the port scope.

Service scope

You can configure the JMS transport at the *service scope* by placing extension elements inside a **wsdl:service** element. Elements in this scope define the default configuration for all endpoints in this service. Any settings in the service scope can be overridden at the port scope.

Port scope

You can configure the JMS transport at the *port scope* by placing extension elements inside a **wsdl:port** element. Elements in the port scope define the configuration for this port. They override any defaults defined at the service scope or at the binding scope.

Example

Example 7.7, "WSDL contract with SOAP/JMS configuration" shows a WSDL contract for a SOAP/JMS service. It configures the JNDI layer in the binding scope, the message delivery details in the service scope, and the reply destination in the port scope.

Example 7.7. WSDL contract with SOAP/JMS configuration

```
<wsd;definitions ...
1     xmlns:soapjms="http://www.w3.org/2010/soapjms/"
    ... >
  ...
  <wsdl:binding name="JMSGreeterPortBinding"
type="tns:JMSGreeterPortType">
    ...
2     <soapjms:jndiInitialContextFactory>
        org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    ...
  </wsdl:binding>
  ...
  <wsdl:service name="JMSGreeterService">
    ...
3     <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
      <soapjms:timeToLive>60000</soapjms:timeToLive>
    ...
    <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
4       <soap:address
 location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
5       <soapjms:replyToName>
          dynamicQueues/greeterReply.queue
      </soapjms:replyToName>
      ...
    </wsdl:port>
    ...
```

```
  </wsdl:service>
  ...
</wsdl:definitions>
```

The WSDL in Example 7.7, "WSDL contract with SOAP/JMS configuration" does the following:

1. Declare the namespace for the SOAP/JMS extensions.
2. Configure the JNDI connections in the binding scope.
3. Configure the JMS delivery style to non-persistent and each message to live for one minute.
4. Specify the target destination.
5. Configure the JMS transport so that reply messages are delivered on the **greeterReply.queue** queue.

7.2. USING WSDL TO CONFIGURE JMS

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 7.8, "JMS WSDL extension namespace" to the definitions element of your contract.

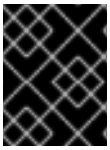
Example 7.8. JMS WSDL extension namespace

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

7.2.1. Basic JMS configuration

Overview

The JMS address information is provided using the **jms:address** element and its child, the **jms:JMSNamingProperties** element. The **jms:address** element's attributes specify the information needed to identify the JMS broker and the destination. The **jms:JMSNamingProperties** element specifies the Java properties used to connect to the JNDI service.



IMPORTANT

Information specified using the JMS feature will override the information in the endpoint's WSDL file.

Specifying the JMS address

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 used in WSDL is identical to the one used in the configuration file. Its attributes are listed in Table 7.5, "JMS endpoint attributes".

Table 7.5. 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 Section 7.3, "Using a Named Reply Destination".
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 Section 7.3, "Using a Named Reply Destination".
connectionUserName	Specifies the user name to use when connecting to a JMS broker.
connectionPassword	Specifies the password to use when connecting to a JMS broker.

The **jms:address** WSDL element uses a **jms:JMSNamingProperties** child element to specify additional information needed to connect to a JNDI provider.

Specifying JNDI properties

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 7.9, "JMS WSDL port specification" shows an example of a JMS WSDL **port** specification.

Example 7.9. 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>
```

7.2.2. JMS client configuration

Overview

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS **ByteMessage** or a JMS **TextMessage**.

When using an **ByteMessage** the consumer endpoint uses a byte[] as the method for storing data into and retrieving data from the JMS message body. When messages are sent, the message data, including any formating information, is packaged into a byte[] and placed into the message body before it is placed on the wire. When messages are received, the consumer endpoint will attempt to unmarshall the data stored in the message body as if it were packed in a byte[].

When using a **TextMessage**, the consumer endpoint uses a string as the method for storing and retrieving data from the message body. When messages are sent, the message information, including any format-specific information, is converted into a string and placed into the JMS message body. When messages are received the consumer endpoint will attempt to unmarshall the data stored in the JMS message body as if it were packed into a string.

When native JMS applications interact with Apache CXF consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the Apache CXF contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as **TextMessage**, the receiving JMS application will get a text message containing all of the SOAP envelope information.

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 7.6. 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 7.10, “WSDL for a JMS consumer endpoint” shows the WSDL for configuring a JMS consumer endpoint.

Example 7.10. 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>
```

7.2.3. 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

Specifying the configuration

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 7.7. 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 7.11, “WSDL for a JMS provider endpoint” shows the WSDL for configuring a JMS provider endpoint.

Example 7.11. 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>
```

7.3. USING A NAMED REPLY DESTINATION

Overview

By default, Apache CXF 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 7.12, "JMS Consumer Specification Using a Named Reply Queue" shows the configuration for a JMS client endpoint.

Example 7.12. JMS Consumer Specification Using a Named Reply Queue

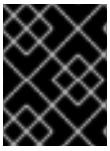
```
<jms:conduit name="
{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
                 jndiConnectionFactoryName="myConnectionFactory"
                 jndiDestinationName="myDestination"
                 jndiReplyDestinationName="myReplyDestination" >
      <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>
```

PART II. CONFIGURING AND PACKAGING ENDPOINTS

Abstract

Endpoints exposed by the Apache CXF 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 Red Hat JBoss Fuse.



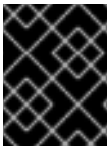
IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

CHAPTER 8. INTRODUCTION TO THE APACHE CXF BINDING COMPONENT

Abstract

Endpoints being deployed using the Apache CXF 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.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

CONTENTS OF A FILE COMPONENT SERVICE UNIT

A service unit that configures the Apache CXF 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 Apache CXF runtime.

`meta-inf/jbi.xml`

The **`jbi.xml`** file is the JBI descriptor for the service unit. Example 8.1, “JBI Descriptor for a Apache CXF Binding Component Service Unit” shows a JBI descriptor for a Apache CXF binding component service unit.

Example 8.1. JBI Descriptor for a Apache CXF Binding Component Service Unit

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

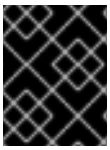
For information on using the Maven tooling to package endpoints into a JBI service unit see Appendix C, *Using the Maven JBI Tooling*.

OSGI PACKAGING

You can package Apache CXF 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 Apache CXF 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, *Using the Maven OSGi Tooling*.

NAMESPACE

The elements used to configure Apache CXF 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, "Namespace Declaration for Using Apache CXF Binding Component Endpoints" to your **xbeans.xml** file's **beans** element.

Example 8.2. Namespace Declaration for Using Apache CXF 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, "Schema Location for Using Apache CXF Binding Component Endpoints".

Example 8.3. Schema Location for Using Apache CXF Binding Component Endpoints

```
<beans ...
       xsi:schemaLocation="...
http://servicemix.apache.org/cxfbc/1.0
http://servicemix.apache.org/cxfbc/1.0/servicemix-cxfbc.xsd
...">
  ...
</beans>
```

CHAPTER 9. CONSUMER ENDPOINTS

Abstract

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.

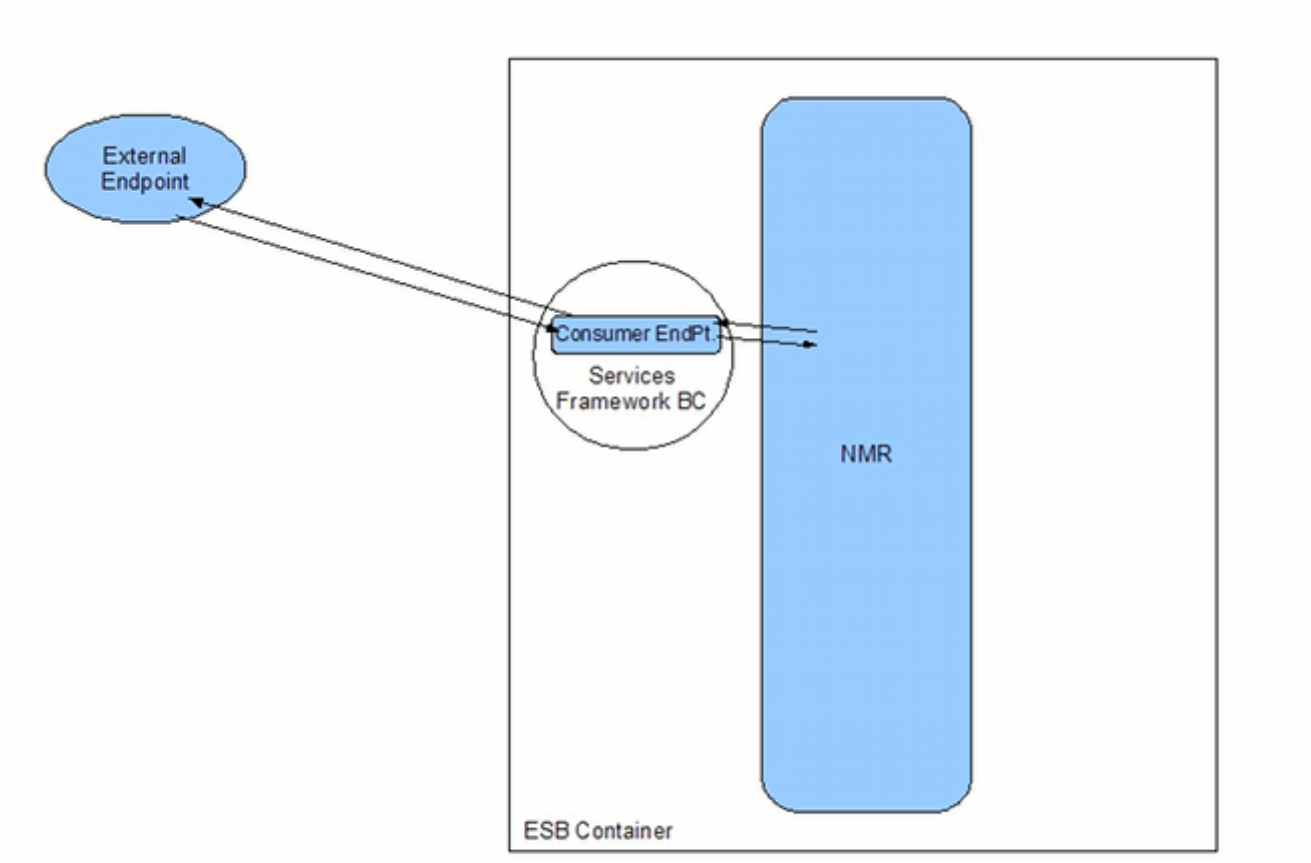
IMPORTANT

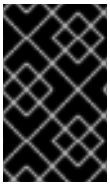
The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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, “Consumer Endpoint”, 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 the section called “Specifying the WSDL” .
3. If your WSDL defines more than one service, you will need to specify a value for the **`service`** attribute.

   See the section called “Specifying the endpoint details” .
4. If the service you choose defines more than one endpoint, you will need to specify a value for the **`endpoint`** attribute.

   See the section called “Specifying the endpoint details” .
5. Specify the details for the target of the requests received by the endpoint.

   See the section called “Specifying the target endpoint” .
6. If your endpoint is going to be receiving binary attachments set its **`mtomEnabled`** attribute to **`true`**.

   See Chapter 11, *Using MTOM to Process Binary Content*.
7. If your endpoint does not need to process the JBI wrapper set its **`useJbiWrapper`** attribute to **`false`**.

   See Chapter 12, *Working with the JBI Wrapper*.
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, “Configuring the CXF Runtime”.

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, “Minimal Consumer Endpoint Configuration” 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, “Defining an Endpoint in WSDL” .

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, “WSDL with Two Services” you would use the configuration shown in Example 9.3, “Consumer Endpoint with a Defined Service Name”.

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, "Service with Two Endpoints" you would use the configuration shown in Example 9.5, "Consumer Endpoint with a Defined Endpoint Name".

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

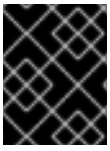
The consumer endpoint will determine the target endpoint in the following manner:

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, “Consumer Endpoint Configuration Specifying a Target Endpoint” 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>
```



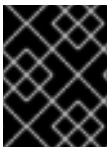
IMPORTANT

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

Abstract

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.



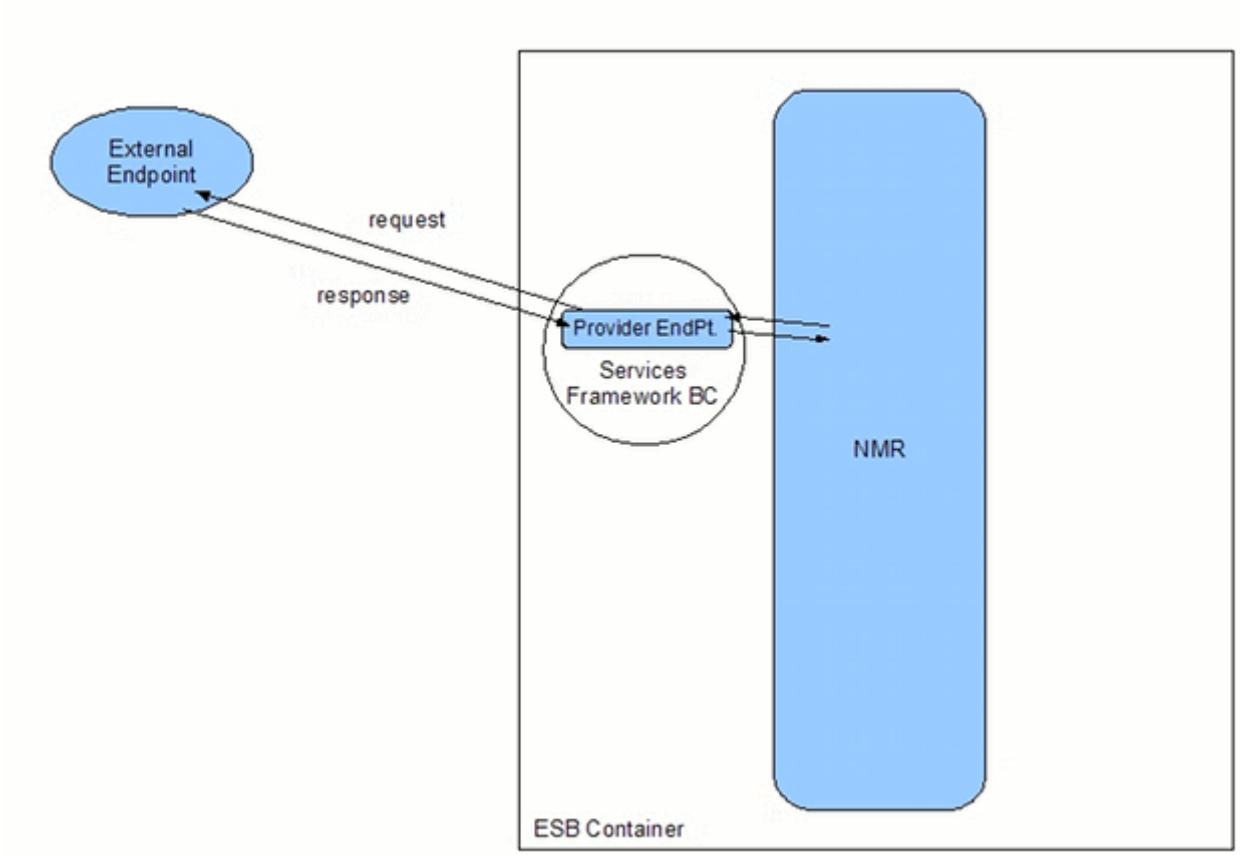
IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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, “Provider Endpoint”, 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 the section called "Specifying the WSDL".
3. If your WSDL defines more than one service, you will need to specify a value for the **service** attribute.

   See the section called "Specifying the endpoint details".
4. If the service you choose defines more than one endpoint, you will need to specify a value for the **endpoint** attribute.

   See the section called "Specifying the endpoint details".
5. If your endpoint is going to be receiving binary attachments set its **mtomEnabled** attribute to **true**.

   See Chapter 11, *Using MTOM to Process Binary Content*.
6. If your endpoint does not need to process the JBI wrapper set its **useJbiWrapper** attribute to **false**.

   See Chapter 12, *Working with the JBI Wrapper*.
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, "Configuring the CXF Runtime".

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, "Minimal Provider Endpoint Configuration" 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, “Defining an Endpoint in WSDL” .

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 provider 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, “WSDL with Two Services” you would use the configuration shown in Example 10.3, “Provider Endpoint with a Defined Service Name”.

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, "Service with Two Endpoints" you would use the configuration shown in Example 10.5, "Provider Endpoint with a Defined Endpoint Name".

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>
```

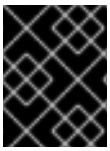
Example 10.5. Provider Endpoint with a Defined Endpoint Name

```
<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

Abstract

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



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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 Apache CXF 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, “Configuring an Endpoint to Use MTOM” , 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

Abstract

By default, all Apache CXF 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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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 Apache CXF 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, “Configuring a Consumer to Not Use the JBI Wrapper” shows a configuration fragment for configuring a consumer that does not process the JBI wrapper.

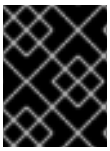
Example 12.1. Configuring a Consumer to Not Use the JBI Wrapper

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

CHAPTER 13. USING MESSAGE INTERCEPTORS

Abstract

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



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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 Apache CXF 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, “Elements Used to Configure an Endpoint's Interceptor Chain” 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, "Configuring an Interceptor Chain" shows a consumer endpoint configured to use the Apache CXF 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 Apache CXF. 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 Apache CXF interceptors see the Apache CXF documentation.

PART III. CONFIGURING THE CXF RUNTIME

Abstract

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

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

CHAPTER 14. CONFIGURING THE ENDPOINTS TO LOAD APACHE CXF RUNTIME CONFIGURATION

Abstract

Both consumers and providers use the **busCfg** attribute to configure the endpoint to load Apache CXF runtime configuration. Its value points to a Apache CXF configuration file.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

SPECIFYING THE CONFIGURATION TO LOAD

You instruct an endpoint to load Apache CXF 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 Apache CXF runtime. This path is relative to the location of the endpoint's **xbean.xml** file.

TIP

The Apache CXF configuration file should be stored in the endpoint's service unit.

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

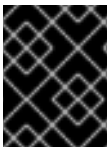
EXAMPLE

Example 14.1, “Provider Endpoint that Loads Apache CXF Runtime Configuration” shows the configuraiton for a provider endpoint that loads a Apache CXF configuration file called **jms-config.xml**.

Example 14.1. Provider Endpoint that Loads Apache CXF 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. TRANSPORT CONFIGURATION



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

15.1. USING THE JMS CONFIGURATION BEAN

Overview

To simplify JMS configuration and make it more powerful, Apache CXF uses a single JMS configuration bean to configure JMS endpoints. The bean is implemented by the **`org.apache.cxf.transport.jms.JMSConfiguration`** class. It can be used to either configure endpoint's directly or to configure the JMS conduits and destinations.

Configuration namespace

The JMS configuration bean uses the Spring p-namespace to make the configuration as simple as possible. To use this namespace you need to declare it in the configuration's root element as shown in Example 15.1, “Declaring the Spring p-namespace”.

Example 15.1. Declaring the Spring p-namespace

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ... >
  ...
</beans>
```

Specifying the configuration

You specify the JMS configuration by defining a bean of class **`org.apache.cxf.transport.jms.JMSConfiguration`**. The properties of the bean provide the configuration settings for the transport.

Table 15.1, “General JMS Configuration Properties” lists properties that are common to both providers and consumers.

Table 15.1. General JMS Configuration Properties

Property	Default	Description
`connectionFactory-ref`		Specifies a reference to a bean that defines a JMS **`ConnectionFactory`**.

Property	Default	Description
wrapInSingleConnectionFactory	**true**	Specifies whether to wrap the **ConnectionFactory** with a Spring **SingleConnectionFactory**. Doing so can improve the performance of the JMS transport when the specified connection factory does not pool connections.
reconnectOnException	**false**	Specifies whether to create a new connection in the case of an exception. This property is only used when wrapping the connection factory with a Spring **SingleConnectionFactory**.
targetDestination		Specifies the JNDI name or provider specific name of a destination.
replyDestination		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 Section 7.3, "Using a Named Reply Destination".
destinationResolver		Specifies a reference to a Spring **DestinationResolver**. This allows you to define how destination names are resolved. By default a **DynamicDestinationResolver** is used. It resolves destinations using the JMS providers features. If you reference a **JndiDestinationResolver** you can resolve the destination names using JNDI.
transactionManager		Specifies a reference to a Spring transaction manager. This allows the service to participate in JTA Transactions.

Property	Default	Description
taskExecutor		Specifies a reference to a Spring **TaskExecutor**. This is used in listeners to decide how to handle incoming messages. By default the transport uses the Spring **SimpleAsyncTaskExecutor**.
useJms11	**false**	Specifies whether JMS 1.1 features are available.
messageIdEnabled	**true**	Specifies whether the JMS transport wants the JMS broker to provide message IDs. Setting this to **false** causes the endpoint to call its message producer's **setDisableMessageID()** method with a value of **true**. The JMS broker is then given a hint that it does not need to generate message IDs or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.
messageTimestampEnabled	**true**	Specifies whether the JMS transport wants the JMS broker to provide message time stamps. Setting this to **false** causes the endpoint to call its message producer's **setDisableMessageTimestamp()** method with a value of **true**. The JMS broker is then given a hint that it does not need to generate time stamps or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.
cacheLevel	**3**	Specifies the level of caching allowed by the listener. Valid values are **0**(CACHE_NONE), **1**(CACHE_CONNECTION), **2**(CACHE_SESSION), **3**(CACHE_CONSUMER), **4**(CACHE_AUTO).

Property	Default	Description
pubSubNoLocal	**false**	Specifies whether to receive messages produced from the same connection.
receiveTimeout	**0**	Specifies, in milliseconds, the amount of time to wait for response messages. **0** means wait indefinitely.
explicitQosEnabled	**false**	Specifies whether the QoS settings like priority, persistence, and time to live are explicitly set for each message or if they are allowed to use default values.
deliveryMode	**1**	Specifies if a message is persistent. The two values are: • **1**(NON_PERSISTENT)—messages will be kept memory • **2**(PERSISTENT)—messages will be persisted to disk
priority	**4**	Specifies the message's priority for the messages. JMS priority values can range from 0 to 9. The lowest priority is 0 and the highest priority is 9.
timeToLive	**0**	Specifies, in milliseconds, the message will be available after it is sent. 0 specifies an infinite time to live.
sessionTransacted	**false**	Specifies if JMS transactions are used.
concurrentConsumers	**1**	Specifies the minimum number of concurrent consumers created by the listener.
maxConcurrentConsumers	**1**	Specifies the maximum number of concurrent consumers by listener.

Property	Default	Description
messageSelector		Specifies the string value of the selector. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
subscriptionDurable	**false**	Specifies whether the server uses durrable subscriptions.
durableSubscriptionName		Specifies the string used to register the durable subscription.
messageType	**text**	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**.
pubSubDomain	**false**	Specifies whether the target destination is a topic.
jmsProviderTibcoEms	**false**	Specifies if your JMS provider is Tibco EMS. This causes the principal in the security context to be populated from the **JMS_TIBCO_SENDER** header.
useMessageIDAsCorrelationID	**false**	Specifies whether JMS will use the message ID to correlate messages. If not, the client will set a generated correlation ID.

As shown in Example 15.2, "JMS configuration bean" , the bean's properties are specified as attributes to the **bean** element. They are all declared in the Spring **p** namespace.

Example 15.2. JMS configuration bean

```
<bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
      p:connectionFactory-ref="connectionFactory"
      p:targetDestination="dynamicQueues/greeter.request.queue"
      p:pubSubDomain="false" />
```

Applying the configuration to an endpoint

The **JMSConfiguration** bean can be applied directly to both server and client endpoints using the

Apache CXF features mechanism. To do so:

1. Set the endpoint's **address** attribute to **jms://**.
2. Add a **jaxws:feature** element to the endpoint's configuration.
3. Add a bean of type **org.apache.cxf.transport.jms.JMSConfigFeature** to the feature.
4. Set the **bean** element's **p:jmsConfig-ref** attribute to the ID of the **JMSConfiguration** bean.

Example 15.3, “Adding JMS configuration to a JAX-WS client” shows a JAX-WS client that uses the JMS configuration from Example 15.2, “JMS configuration bean”.

Example 15.3. Adding JMS configuration to a JAX-WS client

```
<jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"

serviceClass="com.example.customerservice.CustomerService">
  <jaxws:features>
    <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
          p:jmsConfig-ref="jmsConfig"/>
  </jaxws:features>
</jaxws:client>
```

Applying the configuration to the transport

The **JMSConfiguration** bean can be applied to JMS conduits and JMS destinations using the **jms:jmsConfig-ref** element. The **jms:jmsConfig-ref** element's value is the ID of the **JMSConfiguration** bean.

Example 15.4, “Adding JMS configuration to a JMS conduit” shows a JMS conduit that uses the JMS configuration from Example 15.2, “JMS configuration bean”.

Example 15.4. Adding JMS configuration to a JMS conduit

```
<jms:conduit name="
{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-
conduit">
  ...
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

15.2. CONFIGURING THE JETTY RUNTIME

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 15.5, "Jetty Runtime Configuration Namespace" 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 15.5. 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 15.2, "Elements for Configuring a Jetty Runtime Factory".

Table 15.2. Elements for Configuring a Jetty Runtime Factory

Element	Description
`httpj:engine`	Specifies the configuration for a particular Jetty runtime instance. See the section called "The **`engine`** element".

Element	Description
`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 the section called "Configuring the thread pool".

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 15.3, "Elements for Configuring a Jetty Runtime Instance".

Table 15.3. 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 the section called "Configuring the thread pool".
`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 15.4, "Attributes for Configuring a Jetty Thread Pool" .

NOTE

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

Table 15.4. Attributes for Configuring a Jetty Thread Pool

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

Example

Example 15.6, "Configuring a Jetty Instance" shows a configuration fragment that configures a Jetty instance on port number 9001.

Example 15.6. 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 16. DEPLOYING WS-ADDRESSING

Abstract

Apache CXF supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Apache CXF runtime environment.

16.1. 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

Apache CXF 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/.

16.2. WS-ADDRESSING INTERCEPTORS

Overview

In Apache CXF, WS-Addressing functionality is implemented as interceptors. The Apache CXF 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 16.1, “WS-Addressing Interceptors”.

Table 16.1. WS-Addressing Interceptors

Interceptor	Description

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.

16.3. 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:

- Apache CXF 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 16.1, “client.xml—Adding WS-Addressing Feature to Client Configuration” and Example 16.2, “server.xml—Adding WS-Addressing Feature to Server Configuration” respectively.

Example 16.1. 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
http://www.springframework.org/schema/beans/spring-beans.xsd">

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

Example 16.2. 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.springframework.org/schema/beans/spring-beans.xsd">

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

16.4. CONFIGURING WS-ADDRESSING ATTRIBUTES

Overview

The Apache CXF WS-Addressing feature element is defined in the namespace **`http://cxf.apache.org/ws/addressing`**. It supports the two attributes described in Table 16.2, "WS-Addressing Attributes".

Table 16.2. 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 16.3, "Using the Policies to Configure WS-Addressing" an addressing policy assertion to enable non-anonymous responses is embedded in the **policies** element.

Example 16.3. 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.springframework.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 17. ENABLING RELIABLE MESSAGING

Abstract

Apache CXF supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in Apache CXF.

17.1. INTRODUCTION TO WS-RM

Overview

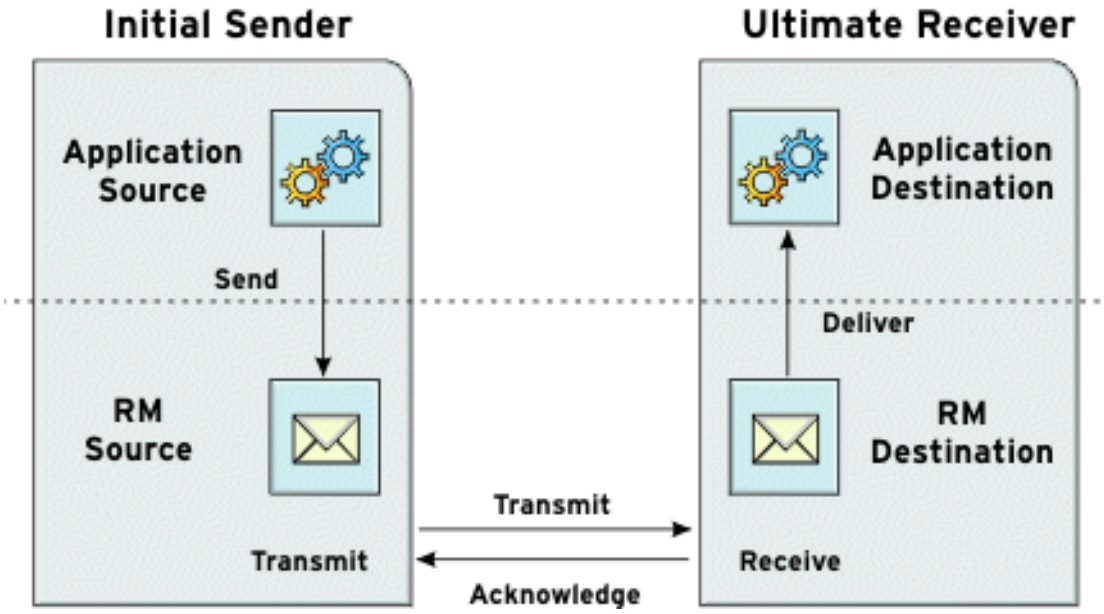
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

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 17.1, “Web Services Reliable Messaging”.

Figure 17.1. 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 Section 17.4, “Configuring WS-RM”.

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

Apache CXF 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.

17.2. WS-RM INTERCEPTORS

Overview

In Apache CXF, WS-RM functionality is implemented as interceptors. The Apache CXF 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.

Apache CXF WS-RM Interceptors

The Apache CXF WS-RM implementation consists of four interceptors, which are described in Table 17.1, “Apache CXF WS-ReliableMessaging Interceptors”.

Table 17.1. Apache CXF 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.
`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 Section 17.3, “Enabling WS-RM”.

Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Apache CXF 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 Section 17.4, “Configuring WS-RM”.

17.3. 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 Apache CXF 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 Apache CXF 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 17.1, "Enabling WS-RM Using Spring Beans").

Example 17.1. Enabling WS-RM Using Spring Beans

```
<?xml version="1.0" encoding="UTF-8"?>
1 <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">
2    <bean id="mapAggregator"
  class="org.apache.cxf.ws.addressing.MAPAggregator"/>
   <bean id="mapCodec"
class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
3    <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">
4        <property name="inInterceptors">
             <list>
               <ref bean="mapAggregator"/>
               <ref bean="mapCodec"/>
               <ref bean="rmLogicalIn"/>
               <ref bean="rmCodec"/>
           </list>
       </property>
5        <property name="inFaultInterceptors">
             <list>
               <ref bean="mapAggregator"/>
               <ref bean="mapCodec"/>
               <ref bean="rmLogicalIn"/>
               <ref bean="rmCodec"/>
           </list>
       </property>
```

```
6         <property name="outInterceptors">
              <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
7         <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 17.1, "Enabling WS-RM Using Spring Beans" can be explained as follows:

1. A Apache CXF 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.
2. Configures each of the WS-Addressing interceptors—**`MAPAggregator`** and **`MAPCodec`**. For more information on WS-Addressing, see Chapter 16, *Deploying WS-Addressing*.
3. Configures each of the WS-RM interceptors—**`RMOutInterceptor`**, **`RMInInterceptor`**, and **`RMSoapInterceptor`**.
4. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.
5. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.
6. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.
7. 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 and Web Services Policy 1.5—Attachment specifications.

To enable WS-RM using the Apache CXF WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. Example 17.2, "Configuring WS-RM using WS-Policy" 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 17.2. 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/addressing/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 17.3, “Adding an RM Policy to Your WSDL File” .

Example 17.3. 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>
```

17.4. CONFIGURING WS-RM

You can configure WS-RM by:

- Setting Apache CXF-specific attributes that are defined in the Apache CXF 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.

17.4.1. Configuring Apache CXF-Specific WS-RM Attributes

Overview

To configure the Apache CXF-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 17.4, "Configuring Apache CXF-Specific WS-RM Attributes" shows a simple example.

Example 17.4. Configuring Apache CXF-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.springframework.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager
http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd">
...
<wsrm-mgr:rmManager>
<!--
  ...Your configuration goes here
-->
</wsrm-mgr:rmManager>
```

Children of the rmManager Spring bean

Table 17.2, "Children of the rmManager Spring Bean" shows the child elements of the **rmManager** Spring bean, defined in the **http://cxf.apache.org/ws/rm/manager** namespace.

Table 17.2. 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 the section called “Maximum unacknowledged messages threshold” .

17.4.2. 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 17.3, “Children of the WS-Policy RMAssertion Element” shows the elements defined in the `http://schemas.xmlsoap.org/ws/2005/02/rm/policy` namespace:

Table 17.3. 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.

Name	Description
`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 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 Apache CXF **`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 Apache CXF-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 17.5, "Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean" shows:

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

Example 17.5. 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 17.6, "Configuring WS-RM Attributes as a Policy within a Feature".

Example 17.6. 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.springframework.org/schema/beans/spring-beans.xsd">
    <jaxws:endpoint name="
{http://cxf.apache.org/greeter_control}GreeterPort"
createdFromAPI="true">
        <jaxws:features>
               <wsp:Policy>
                   <wsrm:RMAssertion
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                     <wsrm:AcknowledgementInterval Milliseconds="200"
/>
                   </wsrm:RMAssertion>
                   <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/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 the section called "WS-Policy framework—implicitly adding interceptors" 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 17.7, "Configuring WS-RM in an External Attachment" shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 17.7. Configuring WS-RM in an External Attachment

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsa="http://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>/
```

17.4.3. 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 Section 17.4.2, "Configuring Standard WS-RM Policy Attributes".

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 17.8, "Setting the WS-RM Base Retransmission Interval" shows how to set the WS-RM base retransmission interval.

Example 17.8. 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 17.9, "Setting the WS-RM Exponential Backoff Property" shows how to set the WS-RM exponential backoff for retransmission.

Example 17.9. 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 17.10, "Setting the WS-RM Acknowledgement Interval" shows how to set the WS-RM acknowledgement interval.

Example 17.10. 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 17.11, "Setting the WS-RM Maximum Unacknowledged Message Threshold" shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 17.11. 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 17.12, "Setting the Maximum Length of a WS-RM Message Sequence" shows how to set the maximum length of an RM sequence.

Example 17.12. 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 17.13, "Setting the WS-RM Message Delivery Assurance Policy" shows how to set the WS-RM message delivery assurance.

Example 17.13. 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>
```

17.5. CONFIGURING WS-RM PERSISTENCE

Overview

The Apache CXF 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.

Apache CXF enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Apache CXF 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

Apache CXF 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 Apache CXF.

The configuration shown in Example 17.14, “Configuration for the Default WS-RM Persistence Store” enables the JDBC-based store that comes with Apache CXF.

Example 17.14. 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 Apache CXF supports the properties shown in Table 17.4, "JDBC Store Properties".

Table 17.4. 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 17.15, "Configuring the JDBC Store for WS-RM Persistence" enables the JDBC-based store that comes with Apache CXF, while setting the driverClassName and url to non-default values.

Example 17.15. 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, "Consumer Endpoint Attributes" 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 Apache CXF bus initialization.	no

Name	Type	Description	Required
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**)
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, "Provider Endpoint Attributes" 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[c][d]
`interfaceName`	QName	Specifies the interface name of the exposed jbi endpoint.	no
`busCfg`	String	Specifies the location of the spring configuration file used for Apache CXF 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

Abstract

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

Red Hat JBoss Fuse 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 including:

- automatic generation of JBI descriptors
- dependency checking

Because Red Hat JBoss Fuse 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 to build all of the service units and the final service assembly.
2. Create a project for each of your service units..
3. Create a project for the service assembly.

C.1. SETTING UP A RED HAT JBOSS FUSE JBI PROJECT

Overview

When working with the Red Hat JBoss Fuse JBI Maven tooling, you create a top-level project that can build all of the service units and then 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 provides you a central location from which to build and deploy the application.

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

Directory structure

Your top-level project contains the following directories:

- A source directory containing the information required for the Maven assembly plug-in
- A directory to store 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

To use the JBoss Fuse JBI Maven tooling, add the elements shown in Example C.1 to your top-level POM file.

Example C.1. POM elements for using Red Hat JBoss Fuse Maven tooling

```
...
<pluginRepositories>
  <pluginRepository>
    <id>fusesource.m2</id>
    <name>FuseSource Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>
<repositories>
  <repository>
    <id>fusesource.m2</id>
    <name>FuseSource Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
    <snapshots>
       <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name>FuseSource Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</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 JBoss Fuse Maven tooling and to load the plug-in that implements the tooling.

Listing the sub-projects

The top-level POM lists all of the service units and the service assembly that is 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 also need a **module** element for the service assembly.

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

Example JBI project pOM

Example C.2 shows a top-level POM for a project that contains a single service unit.

Example C.2. Top-level POM for a Red Hat JBoss Fuse 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>

1  <pluginRepositories>
     <pluginRepository>
      <id>fusesource.m2</id>
      <name>FuseSource Open Source Community Release Repository</name>
      <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
```

```
        <enabled>true</enabled>
      </releases>
    </pluginRepository>
  </pluginRepositories>
  <repositories>
    <repository>
      <id>fusesource.m2</id>
      <name>FuseSource Open Source Community Release Repository</name>
      <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
      <releases>
        <enabled>true</enabled>
      </releases>
    </repository>
    <repository>
      <id>fusesource.m2-snapshot</id>
      <name>FuseSource Open Source Community Snapshot Repository</name>
      <url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
    </repository>
  </repositories>

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

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

The top-level POM shown in Example C.2, “Top-level POM for a Red Hat JBoss Fuse JBI project” does the following:

1. Configures Maven to use the FuseSource repositories for loading the JBoss Fuse plug-ins.
2. 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
3. Configures the Maven assembly plug-in.
4. Loads the JBoss Fuse JBI plug-in.

C.2. A SERVICE UNIT PROJECT

Overview

Each service unit in the service assembly must 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 the minimum, a service unit project contains a POM and an XML configuration file.

Seeding a project using a Maven artifact

Red Hat JBoss Fuse provides Maven artifacts for a number of service unit types. They can be used to seed a project with the **smx-arch** command. As shown in Example C.3, 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 lists the possible values and describes what type of project is seeded.

Table C.1. Service unit archetypes

Name	Description
camel	Creates a project for using the Apache Camel service engine
cxf-se	Creates a project for developing a Java-first service using the Apache CXF service engine
cxf-se-wsdl-first	Creates a project for developing a WSDL-first service using the Apache CXF service engine
cxf-bc	Creates an endpoint project targeted at the Apache CXF binding component
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 chapter "Using Poller Endpoints" in "Using the File Binding Component")
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component (see chapter "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]

Name	Description
jsr181-wsdl-first	Creates a project for developing a WSDL generated Java service to be run by the JSR181 service engine [a]
saxon-xquery	Creates a project for executing xquery statements using the Saxon service engine
saxon-xslt	Creates a project for executing XSLT scripts using the Saxon service engine
eip	Creates a project for using the EIP service engine. [b]
lwcontainer	Creates 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
ode	Create a project for deploying a BPEL process into the ODE service engine

[a] The JSR181 has been deprecated. The Apache CXF service engine has superseded it.

[b] The EIP service engine has been deprecated. The Apache Camel 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 Apache CXF 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 Apache Camel 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.

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

To correctly fill in the metadata required for packaging a service unit, the Maven plug-in must 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 must 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 shows the configuration for a service unit targeting the Apache CXF 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 verify if the targeted component is deployed in the container. If one of the components is not deployed, Red Hat JBoss Fuse will not hold off deploying the service unit until all of the required components are deployed.

TIP

Typically, a message identifying the missing component(s) is written to the log.

If your service unit's targeted component 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 it specifies the name of a targeted component, as specified in Example C.6.

Example C.6. Specifying a target component 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, nor does it download the required component.

Example

Example C.7 shows the POM file for a project that is building a service unit targeted to the Apache CXF binding component.

Example C.7. POM file 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>

1     <parent>
          <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>
2  <packaging>jbi-service-unit</packaging>

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

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

The POM file in Example C.7, “POM file for a service unit project” does the following:

1. Specifies that it is a part of the top-level project shown in Example C.2, “Top-level POM for a Red Hat JBoss Fuse JBI project”
2. Specifies that this project builds a service unit
3. Specifies that the service unit targets the Apache CXF binding component
4. Specifies to use the Red Hat JBoss Fuse Maven plug-in

C.3. A SERVICE ASSEMBLY PROJECT

Overview

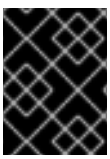
Red Hat JBoss Fuse requires that all service units are bundled into a service assembly before they can be deployed to a container. The JBoss Fuse Maven plug-in collects all of the service units to be bundled and the metadata necessary for packaging. It will then build a service assembly containing the service units.

Seeding a project using a Maven artifact

Red Hat JBoss Fuse 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, the **`smx-arch`** command takes two arguments: the **`groupId`** value and the **`artifactId`** values, which 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

T configure the Maven plug-in to package the results of the project build as a service assembly, change the value of the project's **packaging** element to **jbi-service-assembly**, as shown in Example C.9.

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 must know what service units are being bundled into the service assembly. This is done by specifying the service units as dependencies, using the standard Maven **dependencies** element. Add a **dependency** child element for each service unit. Example C.10 shows the 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 shows a POM file for a project that is 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>

(1)    <parent>
           <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>
(2)  <packaging>jbi-service-assembly</packaging>

(3)  <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>

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

The POM in Example C.11, “POM for a service assembly project” does the following:

(1) Specifies that it is a part of the top-level project shown in Example C.2, “Top-level POM for a Red Hat JBoss Fuse JBI project”

(2) Specifies that this project builds a service assembly

(3) Specifies the service units being bundled by the service assembly

(4) Specifies to use the JBoss Fuse Maven plug-in

[1] You replace this with the version of Apache CXF you are using.

APPENDIX D. USING THE MAVEN OSGI TOOLING

Abstract

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 Red Hat JBoss Fuse OSGi tooling uses the Maven bundle plug-in from Apache Felix. The bundle plug-in is based on the **bnd** 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, 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.

D.1. SETTING UP A RED HAT JBOSS FUSE 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. However, it does 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.

NOTE

There are several Maven archetypes you can use 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, and you place any non-Java resources in the `src/resources` folder.

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

NOTE

Red Hat JBoss Fuse OSGi projects that use Apache CXF, Apache Camel, 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.

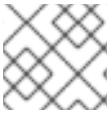
Example D.1, "Adding an OSGi bundle plug-in to a POM" shows the POM entries required to add the bundle plug-in to your project.

Example D.1. Adding an OSGi bundle plug-in to a POM

```
...
<dependencies>
1   <dependency>
      <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
2    <plugin>
       <groupId>org.apache.felix</groupId>
     <artifactId>maven-bundle-plugin</artifactId>
     <configuration>
       <instructions>
         <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
3          <Import-Package>*,org.apache.camel.osgi</Import-Package>
4            <Private-
5 Package>org.apache.servicemix.examples.camel</Private-Package>
             </instructions>
     </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example D.1, "Adding an OSGi bundle plug-in to a POM" do the following:

1. Adds the dependency on Apache Felix
2. Adds the bundle plug-in to your project
3. Configures the plug-in to use the project's artifact ID as the bundle's symbolic name
4. Configures the plug-in to include all Java packages imported by the bundled classes; also imports the org.apache.camel.osgi package
5. Configures the plug-in to bundle the listed class, but not to include them in the list of exported packages



NOTE

Edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see Section D.2, “Configuring the Bundle Plug-In”.

Activating a bundle plug-in

To have Maven use the bundle plug-in, instruct it to package the results of the project as a bundle. 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:

- the section called “Spring OSGi archetype”
- the section called “Apache CXF code-first archetype”
- the section called “Apache CXF wsdl-first archetype”
- the section called “Apache Camel archetype”

Spring OSGi archetype

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

```
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
```

Apache CXF code-first archetype

The Apache CXF code-first archetype creates a project for building a service from Java, as shown:

```
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
```

Apache CXF wsdl-first archetype

The Apache CXF wsdl-first archetype creates a project for creating a service from WSDL, as shown:

```
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
```

Apache Camel archetype

The Apache Camel archetype creates a project for building a route that is deployed into JBoss Fuse, as shown:

```
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
```

D.2. CONFIGURING THE BUNDLE PLUG-IN

Overview

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

While you can create a valid bundle using just the default values, you will probably 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 group Id 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>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to **`${project.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 **`${project.version}`**. Any dashes (**`-`**) are replaced with dots (**`.`**) and the number is padded up to four digits. For example, **`4.2-SNAPSHOT`** becomes **`4.2.0.SNAPSHOT`**.

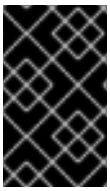
To specify your own value for the bundle's version, add a **`Bundle-Version`** child to the plug-in's **`instructions`** element, as shown in Example D.4.

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 local Java source code (under **`src/main/java`**), *except* for the deault package, **`.`**, and any packages containing **`.impl`** or **`.internal`**.



IMPORTANT

If you use a **`Private-Package`** element in your plug-in configuration and you do not specify a list of packages to export, the default behavior includes only the packages listed in the **`Private-Package`** element in the bundle. No packages are exported.

The default behavior can result in very large packages and in exporting packages that should be kept private. To change the list of exported packages you can add an **`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 that are to be exported. The package names can be specified using the **`*`** wildcard symbol. 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 excluding packages, the order of entries in the list is important. The list is processed in order from the beginning and any subsequent contradicting entries are ignored.

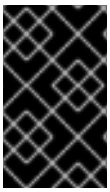
For example, to include all packages starting with com.fuse.demo except the package com.fuse.demo.private, list the packages using:

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

However, if you list the packages using 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

If you want to specify a list of packages to include in a bundle *without* exporting them, you can add a **`Private-Package`** instruction to the bundle plug-in configuration. By default, if you do not specify a **`Private-Package`** instruction, all packages in your local Java source are included in the bundle.



IMPORTANT

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

The **`Private-Package`** element works similarly to the `Export-Package` element in that 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 that are to be included in the bundle. These packages are packaged in the bundle, but not exported (unless they are also selected by the **`Export-Package`** instruction).

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

Example D.5. Including a private package in a bundle

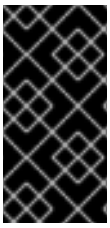
```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

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.

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 an **`Import-Package`** child to the plug-in's **`instructions`** element. The syntax for the package list is the same as for the **`Export-Package`** element and the **`Private-Package`** element.



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 an `*` as the last entry in the package list.

Example D.6 shows the configuration for specifying the packages imported by 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:

- "Managing OSGi Dependencies"
- Apache Felix documentation
- Peter Kriens' aQute Software Consultancy web site

INDEX

A

D

E

H

X