Fuse ESB

Writing WSDL Contracts

Version 4.4.1
Sept. 2011

Writing WSDL Contracts

Version 4.4.1

Updated: 06 Jun 2013

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Tables

List of Examples

Chapter 1. Introducing WSDL Contracts

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

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

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

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.

WSDL elements

A WSDL document is made up of the following elements:

- `definitions` — The root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced in the WSDL document.
- `types` — The XML Schema definitions for the data units that form the building blocks of the messages used by a service. For information about defining data types see "Defining Logical Data Units" on page 15.
- `message` — The description of the messages exchanged during invocation of a services operations. These elements define the arguments of the operations making up your service. For information on defining messages see "Defining Logical Messages Used by a Service" on page 33.
- `portType` — A collection of `operation` elements describing the logical interface of a service. For information about defining port types see "Defining Your Logical Interfaces" on page 37.
- `operation` — The description of an action performed by a service. Operations are defined by the messages passed between two endpoints when the operation is invoked. For information on defining operations see "Operations" on page 38.
- `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.

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 2. Defining Logical Data Units

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

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

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

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

This chapter discusses the first step. "Defining Logical Messages Used by a Service" on page 33 discusses the second step.

Mapping data into logical data units

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

Available type systems for defining service data units

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

XML Schema as a type system

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

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

Considerations for creating your data units

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

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

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

Adding data units to a contract

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

Procedure

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

1. Determine all the data units used in the interface described by the contract.
2. Create a `types` element in your contract.
3. Create a `schema` element, shown in Example 2.1 on page 17, 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 2.1. Schema entry for a WSDL contract

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

4. For each complex type that is a collection of elements, define the data type using a `complexType` element. See "Defining data structures" on page 22.
5. For each array, define the data type using a `complexType` element. See "Defining arrays" on page 26.
6. For each complex type that is derived from a simple type, define the data type using a `simpleType` element. See "Defining types by restriction" on page 29.
7. For each enumerated type, define the data type using a `simpleType` element. See "Defining enumerated types" on page 31.
8. For each element, define it using an `element` element. See "Defining elements" on page 32.

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 2.2 on page 18.

Example 2.2. Defining an element with a simple type

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

Supported XSD simple types

Fuse Services Framework supports the following XML Schema simple types:

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

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

- xsd:gYearMonth
- xsd:gMonthDay

Defining complex data types

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

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

Defining data structures

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

1. The name of the defined type is specified in the `name` attribute of the `complexType` element.
2. The first child element of the `complexType` describes the behavior of the structure's fields when it is put on the wire. See "Complex type varieties" on page 22.
3. Each of the fields of the defined structure are defined in `element` elements that are grandchildren of the `complexType` element. See "Defining the parts of a structure" on page 23.

For example, the structure shown in Example 2.3 on page 22 is be defined in XML Schema as a complex type with two elements.

Example 2.3. Simple Structure

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

Example 2.4 on page 22 shows one possible XML Schema mapping for the structure shown in Example 2.3 on page 22.

Example 2.4. 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 2.1 on page 23 shows the elements used to define complex type behavior.

Table 2.1. Complex type descriptor elements

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

If a `sequence` element, an `all` element, or a `choice` is not specified, then a `sequence` is assumed. For example, the structure defined in Example 2.4 on page 22 generates a message containing two elements: `name` and `age`.

If the structure is defined using a `choice` element, as shown in Example 2.5 on page 23, it generates a message with either a `name` element or an `age` element.

Example 2.5. 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 2.6 on page 24.

Example 2.6. 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 2.7 on page 24. In this case `age` can be omitted and the data will still be valid.

Example 2.7. 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 2.8 on page 24 shows a complex type with an attribute.

Example 2.8. Complex type with an attribute

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

The `attribute` element has three attributes:

- `name` — A required attribute that specifies the string identifying the attribute.

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

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

Defining arrays

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

Complex type arrays

Complex type arrays are a special case of a sequence complex type. You simply define a complex type with a single element and specify a value for the `maxOccurs` attribute. For example, to define an array of twenty floating point numbers you use a complex type similar to the one shown in Example 2.9 on page 26.

Example 2.9. 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 2.10 on page 26.

Example 2.10. 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 2.11 on page 27, 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 2.11. 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 2.12 on page 27.

Example 2.12. Syntax for a SOAP array derived using an element

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

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

Defining types by extension

Like most major coding languages, XML Schema allows you to create data types that inherit some of their elements from other data types. This is called defining a type by extension. For example, you could create a new type called `alienInfo`, that extends the `personalInfo` structure defined in Example 2.4 on page 22 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 2.13 on page 28.

Example 2.13. Type defined by extension

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

Defining types by restriction

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

The definition of a type by restriction requires three things:

1. The name of the new type is specified by the `name` attribute of the `simpleType` element.
2. The simple type from which the new type is derived, called the *base type*, is specified in the `restriction` element. See "Specifying the base type" on page 29.
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the `restriction` element. See "Defining the restrictions" on page 29.

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 2.14 on page 29.

Example 2.14. 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 2.15 on page 30 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 2.15. SSN simple type description

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

Defining enumerated types

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

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in Example 2.16 on page 31.

Example 2.16. 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 2.17 on page 31, would be valid if it contained <widgetSize>big</widgetSize>, but it would not be valid if it contained <widgetSize>big,mungo</widgetSize>.

Example 2.17. widgetSize enumeration

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

Defining elements

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

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

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

Chapter 3. Defining Logical Messages Used by a Service

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

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

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

Tip

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

Messages and parameter lists

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

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

Message design for integrating with legacy systems

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

One approach to defining your messages is RPC style. When using RPC style, you define the messages using one part for each parameter in the method's parameter list. Each message part is based on a type defined in

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

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

Message design for SOAP services

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

- It is a complex type containing a sequence of elements. For more information see "Defining complex data types" on page 21.
- 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 3.1 on page 35.

Table 3.1. Part data type attributes

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

Messages are allowed to reuse part names. For instance, if a method has a parameter, *`foo`*, that is passed by reference or is an in/out, it can be a part in both the request message and the response message, as shown in Example 3.1 on page 35.

Example 3.1. 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 3.2 on page 35.

Example 3.2. personalInfo lookup method

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

This method signature can be mapped to the RPC style WSDL fragment shown in Example 3.3 on page 36.

Example 3.3. 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 3.4 on page 36.

Example 3.4. 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 4. Defining Your Logical Interfaces

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

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

Process

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

1. Create a `portType` element to contain the interface definition and give it a unique name. See "Port types" on page 37.
2. Create an `operation` element for each operation defined in the interface. See "Operations" on page 38.
3. For each operation, specify the messages used to represent the operation's parameter list, return type, and exceptions. See "Operation messages" on page 38.

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 4.1 on page 38.

Table 4.1. 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 4.2 on page 39.

Table 4.2. 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 4.1 on page 39.

Example 4.1. personalInfo lookup interface

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

This interface can be mapped to the port type in Example 4.2 on page 39.

Example 4.2. 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>
```

Index