Artix ESB

Writing Artix ESB Contracts

Version 5.1
December 2007

Making Software Work Together™

Writing Artix ESB Contracts

IONA Technologies

Version 5.1

Published 28 Mar 2008

Trademark and Disclaimer Notice

Copyright Notice

Table of Contents

List of Tables

List of Examples

Preface

Table of Contents

What is Covered in This Book

This book describes how to write an abstract service definition using Web Service Description Language (WSDL). An abstract service definition describes the operations exposed by a service in terms of the messages exchanged during the execution of each operation. These messages are described as XML documents that are implementation neutral. The abstract service definition does not describe how the messages are mapped to data that is transmitted over a network or what communication protocols an implementation of the defined service will use.

Who Should Read This Book

This book is intended for users of Artix ESB who are not familiar with WSDL.

How to Use This Book

This book is organized as follows:

- *Introducing WSDL Contracts* provides a brief overview of the concepts needed to understand a WSDL contract. It also provides an overview of the structure of a WSDL contract.
- *Defining Logical Data Units* describes how to define data types using XML Schema.
- *Defining Logical Messages Used by a Service* describes how data types are built up into the messages that are used in the definition of a WSDL interface.
- *Defining Your Logical Interfaces* describes how to define a service interface in WSDL. Since interface definitions are built up from the elements discussed, you should be sure you understand the concepts in the previous chapters before reading this chapter.

For imformation on adding the physical details to a WSDL document see Bindings and Transports, C++ Runtime or Bindings and Transports, Java Runtime.

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see Using the Artix ESB Library [http://www.iona.com/support/docs/artix/5.1/library_intro/index.htm].

Introducing WSDL Contracts

Summary

WSDL contracts define services using Web Service Description Language and a number of possible extensions. The contracts 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 contract defines how an endpoint implementing a service will interact with the outside world.

Table of Contents

The recommended approach to design services is to define your services in WSDL and XML Schema before writing any code. The GUI tools provided with Artix ESB provide wizards that automate most of the tasks involved in creating a well-formed and valid WSDL document. When hand-editing WSDL contracts you will need to ensure that the contract is valid, as well as correct. To do that you must have some familiarity with WSDL. You can find the standard on the W3C web site, www.w3.org [http://www.w3.org/TR/wsdl].

WSDL Elements

A WSDL document is made up of the following elements:

- `definitions`—the root element of a WSDL contract. The attributes of this element specify the name of the WSDL contract, the contract's target namespace, and the shorthand definitions for the namespaces referenced by the WSDL.
- `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*.
- `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*.
- `portType`—a collection of `operation` elements describing the logical interface of a service. For information about defining port types see *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 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 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 define the physical endpoint on which a service is exposed.

Structure of a WSDL Contract

A WSDL contract 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.

When looked at closely, a WSDL contract has two distinct parts:

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

The logical part

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

Designing a contract

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

1. Define the data types used by your services.
2. Define the messages used in 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.

Defining Logical Data Units

Summary

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

Table of Contents

When defining a service, the first thing you need to consider is how the data used as parameters for the exposed operations are 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* 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 need to 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 need to determine the building blocks from which your messages are built in such a way as 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 like 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 Artix ESB.

XML Schema as a type system

XML Schema is used to define how an XML document is structured. This 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 will hold 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 will make up the message parts.

Considerations for creating your data units

You may 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 that can be accessed at http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES. In addition, the W3C also provides guidelines on 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 Artix ESB 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 at least have some knowledge about what the resulting contract will look like.

Procedure

Defining the data used in an WSDL contract involves seven 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 1, "Schema Entry for a WSDL Contract", as a child of the `type` element.

   The `targetNamespace` attribute is where you specify the namespace under which your new data types are defined. The remaining entries should not be changed.

Example 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.
5. For each array, define the data type using a `complexType` element. See Defining Arrays.
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.
7. For each enumerated type, define the data type using a `simpleType` element. See Defining Enumerated Types.

8. For each element, define it using an `element` element. See Defining Elements.

XML Schema Simple Types

If a message part is going to be of a simple type you do not need 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, “Defining an Element with a Simple Type”.

Example 2. Defining an Element with a Simple Type

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

Supported XSD simple types

Artix ESB 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

Table of Contents

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 allowing you to build 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.
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.

For example the structure shown in Example 3, "Simple Structure" would be defined in XML Schema as a complex type with two elements.

Example 3. Simple Structure

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

Example 4, "A Complex Type" shows one possible XML Schema mapping for the structure shown in Example 3, "Simple Structure".

Example 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 when passed on the wire. The first child element of the `complexType` element determines

which variety of complex type is being used. Table 1, "Complex Type Descriptor Elements" shows the elements used to define complex type behavior.

Table 1. Complex Type Descriptor Elements

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

If neither a `sequence` element, an `all` element, nor a `choice` is specified, a `sequence` is assumed. For example, the structure defined in Example 4, "A Complex Type" would generate a message containing two elements: `name` and `age`.

If the structure was defined using a `choice` element, as shown in Example 5, "Simple Complex Choice Type", it would generate a message with either a `name` element or an `age` element.

Example 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. 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 could define a field, `previousJobs`, that must occur at least three times and no more than seven times as shown in Example 6, "Simple Complex Type with Occurrence Constraints".

Example 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 could also use the `minOccurs` to make the `age` field optional by setting the `minOccurs` to zero as shown in Example 7, "Simple Complex Type with `minOccurs` Set to Zero". In this case `age` can be omitted and the data will still be valid.

Example 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 8, "Complex Type with an Attribute" shows a complex type with an attribute.

Example 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` is 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

Artix ESB 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 transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are nothing more than 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 would use a complex type similar to the one shown in Example 9, “Complex Type Array”.

Example 9. Complex Type Array

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

You could 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 10, “Syntax for a SOAP Array derived using `wsdl:arrayType`”.

Example 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 you would use `[]`; to specify a two-dimensional array you would use either `[][]` or `[,]`.

For example, the SOAP Array, SOAPStrings, shown in Example 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, `[]` implying one dimension.

Example 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 12, "Syntax for a SOAP Array derived using an Element".

Example 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 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 as they would be for a regular complex type.

For example, `alienInfo` would be defined as shown in Example 13, "Type Defined by Extension".

Example 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 could 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.
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.

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 would use a definition like Example 14, “int as Base Type”.

Example 14. int as Base Type

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

Defining the restrictions

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

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

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

Example

Example 15, "SSN Simple Type Description" shows an example of a simple type, `SSN`, which represents a social security number. The resulting type will be 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 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 16, "Syntax for an Enumeration".

Example 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 17, "widgetSize Enumeration", would be valid if it contained <widgetSize>big</widgetSize>, but not if it contained <widgetSize>big,mungo</widgetSize>.

Example 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. At their most basic, an 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` is a required attribute that specifies the name of the element as it will appear 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 if an element can be left out of a document entirely. If `nillable` is set to `true`, the element can be omitted from any document generated using the schema.

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.

Defining Logical Messages Used by a Service

Summary

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 would be 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 only have 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 generally 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 need to 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 would contain one part for each input parameter in the method. Your output message would contain one part for each output parameter and a part to represent the return value if needed. If a parameter is both an input and an output parameter, it would be listed as a part of 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.
- If it is a wrapper for an input message:
  - It would have one element for each of the method's input parameters.
  - Its name would be the same as the name of the operation with which it is associated.
- If it is a wrapper for an output message:
  - It would have one element for each of the method's output parameters and one for each of the method's inout parameters.
  - Its first element would represent 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 also 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. They are 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 2, “Part Data Type Attributes”.

Table 2. 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 18, “Reused Part”.

Example 18. 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 an employee ID number. The method signature for looking up the data would look similar to Example 19, "personalInfo lookup Method".

Example 19. personalInfo lookup Method

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

This method signature could be mapped to the RPC style WSDL fragment shown in Example 20, "RPC WSDL Message Definitions".

Example 20. 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 could also be mapped to the wrapped document style WSDL fragment shown in Example 21, "Wrapped Document WSDL Message Definitions".

Example 21. 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/>
```

Defining Your Logical Interfaces

Summary

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

Defining a logical interface in a WSDL contract entails the following:

1. Creating a `portType` element to contain the interface definition and give it a unique name. See Port types.
2. Creating an `operation` element for each operation defined in the interface. See Operations.
3. For each operation, specifying the messages used to represent the operation's parameter list, return type, and exceptions. See 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 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, specified using the `name` attribute, and is made up of a collection of operations, 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 3, "Operation Message Elements".

Table 3. 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 have any number of `fault` elements.

The elements have the two attributes listed in Table 4, “Attributes of the Input and Output Elements”.

Table 4. 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 `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 22, “personalInfo lookup interface”.

Example 22. personalInfo lookup interface

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

This interface could be mapped to the port type in Example 23, “personalInfo lookup port type”.

Example 23. 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