



Artix 5.6.3

A decorative graphic consisting of several overlapping, wavy blue lines that curve and flow across the lower half of the page, creating a sense of motion and depth.

**Getting Started
with Artix: C++**

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

<http://www.microfocus.com>

Copyright © Micro Focus 2015. All rights reserved.

MICRO FOCUS, the Micro Focus logo and Micro Focus Licensing are trademarks or registered trademarks of Micro Focus IP Development Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

All other marks are the property of their respective owners.

2015-02-11

Contents

Preface	v
Contacting Micro Focus	v
About Artix ESB	1
What is Artix ESB?	1
C++ Runtime Features	3
Key Concepts in Depth.....	4
Artix ESB Runtime Components	4
Artix Bus	5
Artix Endpoints	6
Artix Contracts	6
Artix Services	7
Solving Problems with Artix ESB.....	9
Understanding WSDL	11
WSDL Basics	11
Abstract Data Type Definitions	12
Abstract Message Definitions	14
Abstract Interface Definitions.....	16
Mapping to the Concrete Details.....	18
Index	21

Preface

What is Covered in This Book

Getting Started with Artix introduces the Artix ESB technology and Web Services Description Language (WSDL).

Who Should Read This Book

Getting Started with Artix is for anyone who needs to understand the concepts and terms used in the Artix product.

Organization of This Book

This book contains conceptual information about Artix and WSDL:

- “[About Artix ESB](#)” introduces the Artix ESB product, discussing key concepts in depth and describing the types of problems it is designed to solve.
- “[Understanding WSDL](#)” explains the basics of WSDL.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and finding additional resources, see *Using the Artix Library*.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter <http://www.microfocus.com> in your browser to go to the Micro Focus home page.

Note:

Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, <http://www.microfocus.com>. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter <http://www.microfocus.com> in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- <http://www.microfocus.com/products/corba/artix.aspx> (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx_ (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

<http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp>

About Artix ESB

This chapter introduces the main features of Artix ESB.

What is Artix ESB?

Artix ESB is an extensible enterprise service bus. It provides the tools for rapid application integration that exploits the middleware technologies and products already present within your organization.

The approach taken by Artix ESB relies on existing Web service standards and extends these standards to provide rapid integration solutions that increase operational efficiencies, capitalize on existing infrastructure, and enable the adoption or extension of a service-oriented architecture (SOA).

Web services and SOAs

The information services community generally regards Web services as application-to-application interactions that use SOAP over HTTP.

Web services have the following advantages:

- The data encoding scheme and transport semantics are based on standardized specifications.
- The XML message content is human readable.
- The contract defining the service is XML-based and can be edited by any text editor.
- They promote loosely coupled architectures.

SOAs take the Web services concept and extend it to the entire enterprise. Using a SOA, your infrastructure becomes a collection of loosely coupled services. Each service becomes an endpoint defined by a contract written in Web Services Description Language (WSDL). Clients, or service consumers, can then access the services by reading a service's contract.

Artix and services

Artix extends the Web service standards to include more than just SOAP over HTTP. Thus, Artix allows organizations to define their existing applications as services without worrying about the underlying middleware. It also provides the ability to expose those applications across a number of middleware technologies without writing any new code.

Artix also provides developers with the tools to write new applications in C++ that can be exposed as middleware-neutral services. These tools aid in the definition of the new service in WSDL and in the generation of stub and skeleton code.

Just like the WSDL contracts used to define a service, the code that Artix generates adheres to industry standards.

Benefits of Artix

Artix ESB's extensible nature provides a number of benefits over other ESBs and older enterprise application integration (EAI) products. Chief among these is its speed and flexibility. In addition, Artix ESB provides enterprise levels of service such as session management, service discovery, security, and cross-middleware transaction propagation.

EAI products typically use a proprietary, canonical message format in a centralized EAI hub. When the hub receives a message, it transforms the message to this canonical format and then transforms the message to the format of the target application before sending it to its destination. Each application requires two adapters that are typically proprietary and that translate to and from the canonical format.

By contrast, Artix ESB does not require a hub architecture, nor does it use any intermediate message format. When a message is received by the bus, it is transformed directly into the target application's message format.

Artix ESB is highly configurable and easily extendable. You can configure it to load only the pieces you need for the functionality you require. If Artix ESB does not provide a transport or message format you need, you can easily develop your own service, extend the contract definitions, and configure Artix to load it.

Artix ESB features

Artix ESB includes the following features:

- Support for multiple transports and message data formats
- C++ development
- Message routing
- Cross-middleware transaction support
- Asynchronous Web services
- Deployment of services as plug-ins via a number of different containers
- Role-based security, single sign-on, and security integration
- Session management and stateful Web services
- Look-up services
- Load-balancing
- High-availability service clustering
- Integration with EJBs
- Easy-to-use development tools
- Integration with enterprise management tools such as IBM Tivoli and BMC Patrol
- Support for XSLT-based message transformation
- No need to hard-code WSDL references into applications

Runtimes and programming models

Artix ESB ships with a C++ Runtime that supports development using the Artix C++ API.

Using Artix ESB

There are two ways to use Artix ESB in your enterprise:

- You can use Artix ESB to develop new applications using one of the supported APIs. In this situation, developers generate stub and skeleton code from WSDL, and Artix becomes a part of your development environment.
- You can use the Artix bus to integrate two existing applications, built on different middleware technologies, into a single application. In this situation, developers simply create an Artix contract defining the integration of the systems. In most cases, no new code is needed.

Becoming proficient with Artix ESB

To become an effective Artix ESB developer you need an understanding of the following:

1. The C++ runtime and the programming model available in Artix ESB.
2. The syntax for WSDL and the Artix ESB extensions to the WSDL specification.
3. The configuration mechanisms available in the Artix runtime.
4. The Artix APIs that you can use in your application.

This book introduces these concepts. The other books in the Artix documentation library covers the same technologies in greater detail.

C++ Runtime Features

Artix ESB C++ Runtime provides developers with a C++ API to implement services. It is built on top of Progress Software's patented Adaptive Runtime Technology (ART).

Artix ESB C++ Runtime has a C++ core that provides a fast and stable platform for building applications.

Features supported

The following bindings, transports, and quality of service features are supported by the C++ runtime:

- Supported APIs
 - ◆ WSDL 1.1
- Bindings
 - ◆ SOAP (1.1 and 1.2)
 - ◆ MTOM/XOP
 - ◆ CORBA
 - ◆ Pure XML
 - ◆ Fixed length records
 - ◆ Tagged data
 - ◆ FML
- Transports
 - ◆ HTTP
 - ◆ JMS
 - ◆ FTP
 - ◆ WebSphere MQ
 - ◆ IIOP
 - ◆ Tuxedo
- Quality of Service
 - ◆ Message routing
 - ◆ Security
 - ◆ Reliable messaging
 - ◆ High availability
 - ◆ Load balancing
 - ◆ Location resolution
 - ◆ Statefulness

Key Concepts in Depth

This section discusses key Artix ESB concepts in depth.

Artix ESB Runtime Components

How it fits together

The Artix ESB runtime consists of the following components:

- [Artix Bus](#) is at the core of Artix, and provides the support for various transports and payload formats.
- [Artix Contracts](#) describe your applications in such a way that they become services that can be deployed as [Artix Endpoints](#).
- [Artix Services](#) include a number of advanced services, such as the locator and session manager. Each Artix service is defined with an Artix contract and can be deployed as an Artix endpoint.

Figure 1 illustrates how the Artix ESB elements fit together.

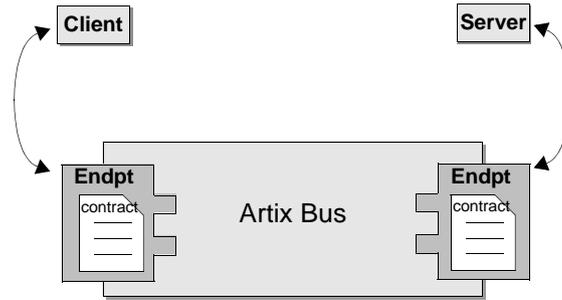


Figure 1: *Artix ESB Runtime Components*

Artix Bus

The Artix bus is at the heart of the Artix ESB architecture. It is the component that hosts the services that you create and connects your applications to those services. The bus is also responsible for translating data from one format into another.

In this way, Artix ESB enables all of the services in your company to communicate, without needing to communicate in the same way. It also means that clients can contact services without understanding the native language of the server handling requests.

Benefits

While other products provide some ability to expose applications as services, they frequently require a good deal of coding. The Artix bus eliminates the need to modify your applications or write code by directly translating the application's native communication protocol into any of the other supported protocols.

For example, by deploying an Artix instance with a SOAP-over-WebSphere MQ endpoint and a SOAP-over-HTTP endpoint, you can expose a WebSphere MQ application directly as a Web service. The WebSphere MQ application does not need to be altered or made aware that it is being exposed using SOAP over HTTP.

The Artix bus translation facility also makes it a powerful integration tool. Unlike traditional EAI products, Artix translates directly between different middlewares without first translating into a canonical format. This saves processing overhead and increases the speed at which messages are transmitted.

Artix Endpoints

An Artix endpoint is the connection point at which a service or a service consumer connects to the Artix bus. Endpoints are described by a contract describing the services offered and the physical representation of the data on the network.

Reconfigurable connection

An Artix endpoint provides an abstract connection point between applications, as shown in [Figure 1](#). The benefit of this abstract connection is that it allows you to change the underlying communication mechanism without recoding any of your applications. You only need to modify the contract describing the endpoint.

For example, if one of your back-end service providers is a Tuxedo application and you want to swap it for a CORBA implementation, you simply change the endpoint's contract to contain a CORBA connection to the Artix bus. The clients accessing the back-end service provider do not need to be aware of the change.

Artix Contracts

Artix contracts are written in WSDL. In this way, a standard language is used to describe the characteristics of services and their associated Artix endpoints. By defining characteristics such as service operations and messages in an abstract way— independent of the transport or protocol used to implement the endpoint—these characteristics can be bound to a variety of protocols and formats.

Artix ESB allows an abstract definition to be bound to multiple specific protocols and formats. This means that the same definitions can be reused in multiple implementations of a service. Artix contracts define the services exposed by a set of systems, the payload formats and transports available to each system, and the rules governing how the systems interact with each other. The simplest Artix contract defines a single pair of systems with a shared interface, payload format, and transport. Artix contracts can also define very complex integration scenarios.

WSDL elements

Understanding Artix contracts requires some familiarity with WSDL. The key WSDL elements are as follows:

WSDL types provide data type definitions used to describe messages.

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with a defined type.

A WSDL operation is an abstract definition of the capabilities supported by a service, and is defined in terms of input and output messages.

A WSDL portType is a set of abstract operation descriptions.

A WSDL binding associates a specific data format for operations defined in a `portType`.

A WSDL port specifies the transport details for a binding, and defines a single communication endpoint.

A WSDL service specifies a set of related ports.

The Artix Contract

An Artix contract is specified in WSDL and is conceptually divided into logical and physical components.

The logical contract

The logical contract specifies components that are independent of the underlying transport and wire format. It fully specifies the data structure and the possible operations or interactions with the interface. It enables Artix to generate skeletons and stubs without having to define the physical characteristics of the connection (transport and wire format).

The logical contract includes the `types`, `message`, `operation`, and `portType` elements of the WSDL file.

The physical contract

The physical component of an Artix contract defines the format and transport-specific details. For example:

- The wire format, middleware transport, and service groupings
- The connection between the `portType` operations and wire formats
- Buffer layout for fixed formats
- Artix extensions to WSDL

The physical contract includes the `binding`, `port`, and `service` elements of the WSDL file.

Artix Services

In addition to the core Artix components, Artix also provides the following services:

- [Container](#)
- [Locator](#)
- [Session manager](#)
- [Transformer](#)
- [Accessing contracts and references](#)

These services provide advanced functionality that Artix deployments can use to gain even more flexibility.

Container

The Artix container provides a consistent mechanism for deploying and managing Artix services. It allows you to write Web service implementations as Artix plug-ins and then deploy your services into the Artix Container.

Using the container eliminates the need to write your own C++ server mainline. Instead, you can deploy your service by simply passing the location of a generated deployment descriptor to the Artix container's administration client.

Locator

The Artix locator provides service look-up and load balancing functionality to an Artix deployment. It isolates service consumers from changes in a service's contact information.

The Artix WSDL contract defines how the client contacts the server, and contains the address of the Artix locator. The locator provides the client with a reference to the server.

Servers are automatically registered with the locator when they start, and service endpoints are automatically made available to clients without the need for additional coding.

Session manager

The Artix session manager is a group of plug-ins that work together to manage the number of concurrent clients that access a group of services. This allows you to control how long each client can use the services in the group before having to check back with the session manager.

In addition, the session manager has a pluggable policy callback mechanism that enables you to implement your own session management policies.

Transformer

The Artix transformer provides Artix ESB with a way to transform operation parameters on the wire using rules written in Extensible Style Sheet Transformation (XSLT) scripts. The transformer can be used to provide a simple means of transforming data. For example, it can be used to develop an application that accepts names as a single string and returns them as separate first and last name strings.

The transformer can also be placed between two applications where it can transform messages as they pass between the applications. This functionality allows you to connect applications that do not use exactly the same interfaces and still realize the benefits of not using a canonical format without rewriting the underlying applications.

Accessing contracts and references

Accessing contracts and references in Artix ESB refers to enabling client and server applications to find WSDL service contracts and references. Using the techniques and conventions of Artix avoids the need to hard code WSDL into your client and server applications.

For more information

For more information on Artix services, see *Configuring and Deploying Artix*.

Solving Problems with Artix ESB

Artix ESB allows you to solve problems arising from the integration of existing back-end systems using a service-oriented approach. It allows you to develop new services using C++ or Java, and to retain all of the enterprise levels of service that you require.

There are three phases to an Artix ESB project:

1. The design phase, where you define your services and define how they are integrated using Artix contracts.
2. The development phase, where you write the application code required to implement new services.
3. The deployment phase, where you configure and deploy your Artix solution.

Design phase

In the design phase, you define the logical layout of your system in an Artix contract. The logical or abstract definition of a system includes:

- the services that it contains
- the operations each service offers
- the data the services will use to exchange information

Once you have defined the logical aspects of your system, you then add the physical network details to the contracts.

The physical details of your system include the transports and payload formats used by your services, as well as any routing schemes needed to connect services that use different transports or payload formats.

The Artix command-line tools automate the mapping of your service descriptions into WSDL-based Artix contracts. These tools allow you to:

- Generate Artix contracts from:
 - ◆ CORBA IDL
 - ◆ A description of tagged data
 - ◆ A description of fixed record length data
 - ◆ A COBOL copybook

- Add the following bindings to an Artix contract:
 - ◆ CORBA
 - ◆ Fixed record length
 - ◆ SOAP
 - ◆ Tagged data
 - ◆ XML

Development phase

You must write Artix application code if your solution involves creating new applications or a custom router, or involves using the Artix session management feature. The first step in writing Artix code is to generate client stub code and server skeleton code from the Artix contracts that you created in the design phase. You can generate this code using the Artix command-line tools.

After you have generated the client stub code and server skeleton code, you can develop the code that implements the business logic you require. For most applications, Artix-generated code allows you to stick to using standard C++ ~~or Java~~ code for writing business logic.

Artix ESB also provides advanced APIs for directly manipulating messages, for writing message handlers, and for other advanced features your application might require. These can be plugged into the Artix runtime for customized processing of messages.

Deployment phase

In the deployment phase, you configure the Artix runtime to fine-tune the Artix bus for your new Artix system. This involves modifying the Artix configuration files and editing the Artix contracts that describe your solution to fit the exact circumstances of your deployment environment.

This phase also includes the managing of the deployed system. This might involve, for example, using an enterprise management tool along with the Artix command interface. These tools allow you to further fine-tune your system.

Understanding WSDL

Artix contracts use WSDL documents to describe services and the data they use.

WSDL Basics

Web Services Description Language (WSDL) is an XML document format used to describe services offered over the Web. WSDL is standardized by the World Wide Web Consortium (W3C). Artix currently supports revision 1.1. You can find the standard on the W3C website at <http://www.w3.org/TR/wsdl>.

Elements of a WSDL document

A WSDL document is made up of the following elements:

- `import` allows you to import another WSDL or XSD file.
- Logical contract elements:
 - ◆ `types`
 - ◆ `message`
 - ◆ `operation`
 - ◆ `portType`
- Physical contract elements:
 - ◆ `binding`
 - ◆ `port`
 - ◆ `service`

These elements are described in [“WSDL elements” on page 6](#).

Abstract operations

The abstract definition of *operations* and *messages* is separated from the concrete data formatting definitions and network protocol details. As a result, the abstract definitions can be reused and recombined to define several endpoints. For example, a service can expose identical operations with slightly different concrete data formats and two different network addresses. Alternatively, one WSDL document could be used to define several services that use the same abstract messages.

The portType

A *portType* is a collection of abstract operations that define the actions provided by an endpoint.

Concrete details

When a `portType` is mapped to a concrete data format, the result is a concrete representation of the abstract definition. A *port* is defined by associating a network address with a reusable *binding*, in the form of an endpoint. A collection of ports (or endpoints) define a service.

Because WSDL was intended to describe services offered over the Web, the concrete message format is typically SOAP and the network protocol is typically HTTP. However, WSDL documents can use any concrete message format and network protocol. In fact, Artix contracts bind operations to several data formats and describe the details for a number of network protocols.

Namespaces and imported descriptions

WSDL supports the use of XML namespaces defined in the `definition` element as a way of specifying predefined extensions and type systems in a WSDL document. WSDL also supports importing WSDL documents and fragments for building modular WSDL collections.

Abstract Data Type Definitions

Applications typically use data types that are more complex than the primitive types, like `int`, defined by most programming languages. WSDL documents represent these complex data types using a combination of schema types defined in referenced external XML schema documents and complex types described in `types` elements.

Complex type definitions

Complex data types are described in a `types` element. The W3C specification states that XSD is the preferred canonical type system for a WSDL document. Therefore, XSD is treated as the intrinsic type system. Because these data types are abstract descriptions of the data passed over the wire, and are not concrete descriptions, there are a few guidelines on using XSD schemas to represent them:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.
- Define arrays using the SOAP 1.1 array encoding format.

WSDL does allow for the specification and use of alternative type systems within a document.

Example

The structure, `personalInfo`, defined in [Example 1](#), contains a string, an int, and an enum. The string and the int both have equivalent XSD types and do not require special type mapping. The enumerated type `hairColorType`, however, does need to be described in XSD.

Example 1: *personalInfo structure*

```
enum hairColorType {red, brunette, blonde};

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

[Example 2](#) shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in [Example 1](#). `hairColorType` is described using a named `simpleType` because it does not have any child elements. `personalInfo` is defined as an `element` so that it can be used in messages later in the contract.

Example 2: *XSD type definition for personalInfo*

```
<types>
  <xsd:schema
    targetNamespace="http://microfocus.com/personal/schema"
    xmlns:xsd="http://microfocus.com/personal/schema"
    xmlns="http://www.w3.org/2000/10/XMLSchema"/>
    <simpleType name="hairColorType">
      <restriction base="xsd:string">
        <enumeration value="red"/>
        <enumeration value="brunette"/>
        <enumeration value="blonde"/>
      </restriction>
    </simpleType>
    <element name="personalInfo">
      <complexType>
        <sequence>
          <element name="name" type="xsd:string"/>
          <element name="age" type="xsd:int"/>
          <element name="hairColor"
            type="xsd:hairColorType"/>
        </sequence>
      </complexType>
    </element>
  </types>
```

Another way to map `personalInfo` is to describe `hairColorType` in-line as shown in [Example 3](#). With this mapping, however, you cannot reuse the description of `hairColorType`.

Example 3: *Alternate XSD Mapping for `personalInfo`*

```
<types>
  <xsd:schema
    targetNamespace="http://microfocus.com/personal/schema"
    xmlns:xsd1="http://microfocus.com/personal/schema"
    xmlns="http://www.w3.org/2000/10/XMLSchema"/>
    <element name="personalInfo">
      <complexType>
        <sequence>
          <element name="name" type="xsd:string"/>
          <element name="age" type="xsd:int"/>
          <element name="hairColor">
            <simpleType>
              <restriction base="xsd:string">
                <enumeration value="red"/>
                <enumeration value="brunette"/>
                <enumeration value="blonde"/>
              </restriction>
            </simpleType>
          </element>
        </sequence>
      </complexType>
    </element>
  </types>
```

Abstract Message Definitions

WSDL is designed to describe how data is passed over a network. It describes data that is exchanged between two endpoints in terms of abstract messages described in `message` elements.

Each abstract message consists of one or more parts, defined in `part` elements.

These abstract messages represent the parameters passed by the operations defined by the WSDL document and are mapped to concrete data formats in the WSDL document's `binding` elements.

Messages and parameter lists

For simplicity in describing the data consumed and provided by an endpoint, WSDL documents allow abstract operations to have only one input message, the representation of the operation's incoming parameter list, and only one output message, the representation of the data returned by the operation.

In the abstract message definition, you cannot directly describe a message that represents an operation's return value. Therefore, any return value must be included in the output message.

Messages allow for concrete methods defined in programming languages like C++ to be mapped to abstract WSDL operations. Each message contains a number of `part` elements that represent one element in a parameter list.

Therefore, all of the input parameters for a method call are defined in one message and all of the output parameters, including the operation's return value, are mapped to another message.

Example

For example, imagine a server that stores personal information as defined in [Example 1 on page 13](#) and provides a method that returns an employee's data based on an employee ID number. The method signature for looking up the data would look similar to [Example 4](#).

Example 4: *Method for Returning an Employee's Data*

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

This method signature could be mapped to the WSDL fragment shown in [Example 5](#).

Example 5: *WSDL Message Definitions*

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

Message naming

Each message in a WSDL document must have a unique name within its namespace. Choose message names that show whether they are input messages (requests) or output messages (responses).

Message parts

Message parts are the formal data elements of the abstract message. Each part is identified by a `name` attribute and by either a `type` or an `element` attribute that specifies its data type. The data type attributes are listed in [Table 1](#).

Table 1: *Part Data Type Attributes*

Attribute	Description
<code>type="type_name"</code>	The data type of the part is defined by a <code>simpleType</code> or <code>complexType</code> called <code>type_name</code> .
<code>element="elem_name"</code>	The data type of the part is defined by an element called <code>elem_name</code> .

Messages are allowed to reuse part names. For instance, if a method has a parameter, `foo`, which is passed by reference or is an in/out, it can be a part in both the request message and the response message. An example of parameter reuse is shown in [Example 6](#).

Example 6: *Reused Part*

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

Abstract Interface Definitions

WSDL `portType` elements define, in an abstract way, the operations offered by a service. The operations defined in a `portType` list the input, output, and any fault messages used by the service to complete the transaction the operation describes.

PortTypes

A `portType` can be thought of as an interface description. In many Web service implementations there is a direct mapping between `portTypes` and implementation objects. `PortTypes` are the abstract unit of a WSDL document that is mapped into a concrete binding to form the complete description of what is offered over a port.

`PortTypes` are described using the `portType` element in a WSDL document. Each `portType` 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 `portTypes`.

Operations

Operations, described in `operation` elements in a WSDL document, are an abstract description of an 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 within a `portType` must have a unique name, specified using the required `name` attribute.

Elements of an operation

Each operation is made up of a set of elements. The elements represent the messages communicated between the endpoints to execute the operation.

The elements that can describe an operation are listed in [Table 2](#).

Table 2: *Operation Message Elements*

Element	Description
input	Specifies a message that is received from another endpoint. This element can occur at most once for each operation.
output	Specifies a message that is sent to another endpoint. This element can occur at most once for each operation.
fault	Specifies a message used to communicate an error condition between the endpoints. This element is not required and can occur an unlimited number of times.

An operation is required to have at least one `input` or `output` element. The elements are defined by two attributes listed in [Table 3](#).

Table 3: *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 <code>message</code> attribute must correspond to the <code>name</code> 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 `portType` 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 is mapped into the output message as the last part of that message. The concrete details of how the message parts are mapped into a physical representation are described in ["Bindings" on page 19](#).

Example

For example, in implementing a server that stores personal information in the structure defined in [Example 1 on page 13](#), you might use an interface similar to the one shown in [Example 7](#).

Example 7: *personalInfo Lookup Interface*

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

This interface could be mapped to the portType in [Example 8](#).

Example 8: *personalInfo Lookup Port Type*

```
<types>
...
<element name="idNotFound" type="idNotFoundType">
<complexType name="idNotFoundType">
<sequence>
    <element name="ErrorMsg" type="xsd:string"/>
    <element name="ErrorID" type="xsd:int"/>
</sequence>
</complexType>
</types>
<message name="personalLookupRequest">
<part name="empId" type="xsd:int" />
</message>
<message name="personalLookupResponse">
    <part name="return" element="xsd1:personalInfo" />
</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>
```

Mapping to the Concrete Details

The abstract definitions in a WSDL document are intended to be used in defining the interaction of real applications that have specific network addresses, use specific network protocols, and expect data in a particular format. To fully define these real applications, the abstract definitions discussed in the previous section must be mapped to concrete representations of the data passed between applications. The details describing the network protocols in use must also be added.

This is accomplished in the WSDL `bindings` and `ports` elements. WSDL binding and port syntax is not tightly specified by the W3C. A specification is provided that defines the mechanism for defining these syntaxes. However, the syntaxes for bindings other than SOAP and for network transports other than HTTP are not defined in a W3C specification.

Bindings

Bindings describe the mapping between the abstract messages defined for each `portType` and the data format used on the wire. Bindings are described in `binding` elements in the WSDL file. A binding can map to only one `portType`, but a `portType` can be mapped to any number of bindings.

It is within the bindings that you specify details such as parameter order, concrete data types, and return values. For example, a binding can reorder the parts of a message to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

Services

To define an endpoint that corresponds to a running service, the `port` element in the WSDL file associates a binding with the concrete network information needed to connect to the remote service described in the file. Each port specifies the address and configuration information for connecting the application to a network.

Ports are grouped within `service` elements. A service can contain one or many ports. The convention is that the ports defined within a particular service are related in some way. For example, all of the ports might be bound to the same `portType`, but use different network protocols, like HTTP and WebSphere MQ.

Index

A

Artix
 bus 5
 contracts 6, 7
 locator 8
 session manager 8
 transformer 8

B

bindings 7, 19
bus 5

C

C++ Runtime 3
contracts 6, 7
CORBA IDL 9

D

deployment phase 10
design phase 9
development phase 10
documentation
 .pdf format vi
 updates on the web vi

E

EAI 2
enterprise application integration, see EAI
enterprise service bus, See ESB

I

IDL 9

L

locator 8

M

messages 6

O

operations 7, 16

P

ports 7
portTypes 7, 11, 16

R

runtimes
 C++ 3

S

service-oriented architecture, see SOA
services 7, 19
session manager 8
SOA 1
SOAP 1

T

transformer 8
types 6

W

W3C 11
Web Services Description Language, see
 WSDL
World Wide Web Consortium, see W3C
WSDL 6, 11–19
 defined 11

X

XSD 12

