Artix™

Getting Started with Artix

Version 2.0, March 2004

Making Software Work Together™

Updated: 22-Apr-2004

M 3 1 9 3

Contents

List of Figures

List of Tables

Preface

What is Covered in this Book

Getting Started with Artix provides an introduction to IONA's Artix technology. This book gives a brief overview of the architecture and functionality of Artix, and a brief introduction to Web Services Definition Language (WSDL). This book also points you to other documents in the Artix library for more detailed information and examples.

Who Should Read this Book

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

Organization of this Book

This book contains the following chapters:

- Chapter 1 introduces the Artix product, and the kind of problems that it is designed to solve.
- Chapter 2 explains the main concepts used in Artix.
- Chapter 3 explains the basics of Web Services Definition Language (WSDL).
- Chapter 4 gives an overview of the Artix Designer GUI tool.

Related Documentation

The documentation for Artix includes the following books:

- *Artix Tutorial*
- *Designing Artix Solutions with Artix Designer*
- *Designing Artix Solutions from the Command Line*
- *Deploying and Managing Artix Solutions*
- *Designing Artix Applications in C++*
- *Designing Artix Applications in Java*
- *Artix Security Guide*
- *Artix Thread Library Reference*
- *IONA Tivoli Integration Guide*
- *IONA BMC Patrol Integration Guide*

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs/artix/2.0/index.xml.

Online Help

The Artix Designer GUI includes a comprehensive online help system, which includes:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index and glossary.
- A full search feature.
- Context-sensitive help.

The **Help** menu in **Artix Designer** provides access to this online help. In addition, online help is provided for the Artix integration with BMC Enterprise Management Systems. See the BMC Patrol **Help** menu for details.

Suggested Path for Further Reading

If you are new to Artix, we suggest you read the documentation in the following order:

1. *Getting Started with Artix*

   This guide describes the Artix product and its main concepts.

2. *Artix Tutorial*

   This guide walks you through using the Artix tools to develop and deploy simple example applications.

3. *Deploying and Managing Artix Solutions*

   This guide describes deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.

4. *Designing Artix Solutions with Artix Designer*

   This guide shows how to use the Artix GUI to describe your services in an Artix contract.

5. *Designing Artix Solutions from the Command Line*

   This guide provides detailed information about the WSDL extensions used in Artix contracts and explains the mappings between data types and Artix bindings.

6. *Developing Artix Applications in C++/Java*

   These guides discuss the technical aspects of programming applications using the Artix API.

Additional Resources for Help

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about Artix and other products. You can access the knowledge base at the following location:

The IONA update center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with ***italic*** words or characters.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Introduction to Artix

This chapter introduces the main features of Artix, and describes where to look in the documentation for further information.

In this chapter

This chapter discusses the following topics:

What is Artix?

Overview

Artix is a Web services-based solution for enterprise application integration. It involves a new approach to application integration that exploits the middleware technologies and the products already present within an enterprise. Artix provides a rapid integration approach that increases operational efficiencies, and enables an enterprise to adopt or extend a Service-Oriented Architecture (SOA).

Artix technology

Artix allows organizations to expose existing applications as Web services, without changing underlying middleware. Artix enables developers to write new Web service applications in C++ and Java. In addition, Artix provides enterprise levels of service such as session management, service look-up, security, and transaction propagation.

Artix uses IONA's proven *Adaptive Runtime Technology* (ART) to provide a high-speed, robust, and highly scalable backbone for Web service deployments. In addition, Artix extends ART and the Web service metaphor by using the *Artix Bus*, IONA's transport and message format switching technology. The Artix Bus enables you to create Web services that communicate using protocols other than SOAP over HTTP. For example, you can develop and deploy Web services using proven enterprise quality communication mechanisms such as TIBCO Rendevous™, CORBA, and IBM WebSphere MQ.

Benefits of Artix

Artix differs from the typical approach used by Enterprise Application Integration (EAI) products. The EAI approach typically uses a canonical format in a centralized EAI hub. All messages are transformed from a source application's native format to this canonical format, and then transformed again to the format of the target application. Each application requires two adapters that translate to and from the canonical format.

Requiring two translations for every message incurs high overheads. Many enterprises would prefer a high-performance solution that directly transforms a small set of message types instead of a more general solution with lower performance.

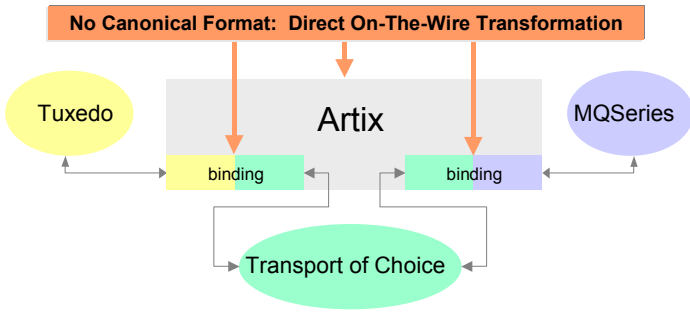

Figure 1: *Artix High-Performance Architecture*

The Artix model connects applications at the middleware transport level and translates messages only once. It hides the details of the connection and provides very high performance. Figure 1 shows an example Artix integration between BEA Tuxedo and IBM MQSeries.

Artix also enables you to obtain maximum value from your IT assets through the reuse of your existing systems. You can lower operating costs by consolidating diverse systems and existing infrastructure.

Finally, Artix provides a range of easy-to-use tools that enable you to create, manage, and deploy your integration solutions. These include GUI tools, command-line tools, and APIs. For example, Figure 2 shows the main window of the Artix Designer GUI tool. This tool automates and simplifies the creation of Web service integration applications.

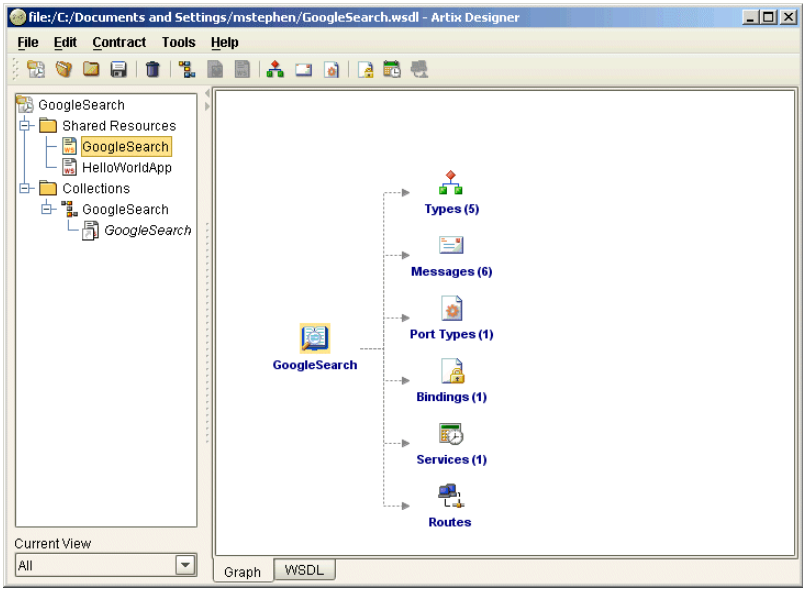

Figure 2: *Artix Designer GUI Tool*

Artix features

Artix includes for the following unique features:

- Support for multiple transports and message data formats
- C++ and Java Web service development
- Message routing
- Transaction support for Web services
- Asynchronous Web services
- Role-based security, single sign on, and security integration.
- Session management and stateful Web services
- Look-up services and load-balancing

- Support for EJBs and JMS
- Easy-to-use WSDL tools
- Support for .NET
- Integration with enterprise management tools such as IBM Tivoli and BMC Patrol
- Internationalization

Note: Single sign-on, locator look-up services, and session management are not available in all editions of Artix. Please check the conditions of your Artix license to see if your installation supports these features.

Supported transports and protocols

A *transport* is an on-the-wire format for messages; while a *protocol* is a transport that is defined by an open specification. For example, MQ and Tuxedo are transports, while HTTP and IIOP are protocols.

In Artix, both protocols and transports are referred to as transports. Artix supports the following message transports:

- HTTP
- BEA Tuxedo
- IBM WebSphere MQ (formerly MQSeries)
- TIBCO Rendezvous™
- IIOP
- IIOP Tunnel
- Java Messaging Service

Supported payload formats

A *payload format* controls the layout of a message delivered over a transport. Artix can automatically transform between the following payload formats:

- G2++
- FML (a Tuxedo format)
- GIOP (a CORBA format)
- FRL (fixed record length)
- Tagged (variable length)

- SOAP
- TibrvMsg (a TIBCO Rendezvous format)
- Pure XML

The mapping of logical data items between payload formats is supported by the Artix tools.

Further information

For more details information about supported transports and payload formats, see *Designing Artix Solutions on the Command Line*.

For information about Artix mainframe support, see the Artix Mainframe documentation.

Solving Problems with Artix

Overview

Artix enables you to easily solve problems of how to integrate existing back-end systems using Web services. It also enables you to develop new Web services using C++ or Java, and retain all of the enterprise levels of service that you require.

This section describes the three main phases in an Artix solution: design, development, and deployment. Artix integration solutions that use a standalone Artix service do not always require the development of any code, and may involve design and deployment phases only.

Design phase

In the design phase, you map out the architecture of the systems that you wish to integrate or develop. You decide what services you wish to build, what operations each service will need, and the data that the services will need to exchange.

After making these decisions, you map the information into Artix contracts that describe the services, operations, and data types. As part of this step, you also map out the transports used by each service and any routing rules that will be used.

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

- Import existing WSDL documents (for example, those generated by third-party tools).
- Generate a WSDL contract from scratch.
- Generate a WSDL contract from an external metadata source (for example, CORBA IDL).

Development phase

If your solution involves creating new applications, a custom router, or locator or session management features, you will need to develop some Artix application code (C++ or Java). This involves generating 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 Designer GUI and command-line tools.

When you have generated the client stub code and server skeleton code, you can then develop the code that implements the business logic you require. Artix takes care of generating the starting point code, and you can then use your favorite development environment to develop and debug the application code.

If necessary, Artix also provides advanced APIs for directly manipulating SOAP messages, and for writing SOAP message handlers. These can be plugged into the Artix runtime for custom-built processing of SOAP messages.

Deployment phase

In the deployment phase, you take the Artix contracts from the design phase, and any applications created in the development phase, and deploy your integrated system. You may need to modify the generated Artix configuration files, or edit the Artix contracts describing your solution to fit the exact circumstances of your deployment environment.

Applications that use Artix can be deployed in two different ways:

Embedded mode

In embedded mode, applications are modified to invoke Artix functions directly and locally, instead of invoking a standalone Artix service. This approach is the most invasive to the application, but also provides the highest performance. Embedded mode requires linking the application with the Artix-generated stub and skeleton code to connect the client and server to the Artix Bus.

Standalone mode

In standalone mode, Artix runs as a separate process that is invoked as a service. Standalone mode provides a zero-touch integration solution that does not involve any coding. However, standalone mode is less efficient than embedded mode.

When designing a system, you simply generate and deploy the Artix contracts that specify each endpoint of the Artix Bus. Because a standalone switch is not linked directly with the applications that use it (in embedded mode), a contract for standalone mode deployment must specify routing information.

Using the Artix Documentation

Overview

The Artix documentation library consists of a number of guides to help you understand and use Artix. The guides are broken down into groups reflecting the three phases of Artix problem solving. In addition Artix provides a Tutorial that provides a number of guided exercises to build your skill using Artix.

If you are new to Artix

If you are approaching Artix for the first time, it is suggested that you work through the library in the following order:

1. *Getting Started with Artix*
2. *Artix Tutorial*
3. *Deploying and Managing Artix Solutions*
4. *Designing Artix Solutions with Artix Designer*
5. *Designing Artix Solutions from the Command Line*
6. *Developing Artix Applications (C++/Java)*

Design guides

Designing Artix Solutions with Artix Designer explains how to create and manage Artix contracts using the Artix Designer GUI. It also explains how to generate stub and skeleton code for development, and configuration files for deployment.

Designing Artix Solutions from the Command Line explains how to create and manage Artix contracts using the Artix command line tools. It also explains how to generate stubs, skeletons, and configuration files. This book contains detailed descriptions of the Artix WSDL extensions used to define routes, payload formats, and transports. It also provides an overview of WSDL and how it maps to certain programming concepts.

Development guides

Artix includes two main development guides:

- *Developing Artix Applications in C++*
- *Developing Artix Applications in Java*

Both guides describe how to develop clients and servers using the Artix APIs. They provide examples of advanced usages of Artix such as transactions, using locator services, session management, and dynamic configuration.

In addition, Artix provides the *Artix Security Guide* for security programmers, and *Artix Thread Library Reference*, which explains the thread control library used in the Artix API.

Deployment guides

Deploying and Managing Artix Solutions explains how to configure and deploy all aspects of an Artix solution. It describes the Artix configuration file, where to locate the contracts that control your Artix services, and how to run Artix applications. It also explains how to configure and deploy the Artix locator and session manager.

In addition, Artix provides guides for integration with third party Enterprise Management Systems (for example, IBM Tivoli and BMC Patrol). The *IONA Tivoli Integration Guide* explains Artix integration with the IBM Tivoli suite of management tools. The *IONA BMC Patrol Integration Guide* explains Artix integration with BMC Patrol tools.

Latest updates

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs/artix/2.0/index.xml.

CHAPTER 2

Artix Concepts

This chapter introduces the key concepts used in the Artix product.

In this chapter

This chapter discusses the following topics:

The Elements of Artix

Overview

This section gives a high-level overview of the main components in the Artix product:

- The Artix Bus
- Artix Service Access Points
- Artix Contracts

Artix components

Artix's unique features are implemented by a number of plug-ins to IONA's ART platform. These plug-ins form the core of Artix, the *Artix Bus*. Applications that make use of Artix connect to the Bus using Artix *Service Access Points* (SAPs). Service Access Points are described by *Artix Contracts*.

Figure 3 shows an example of how all of these Artix elements fit together.

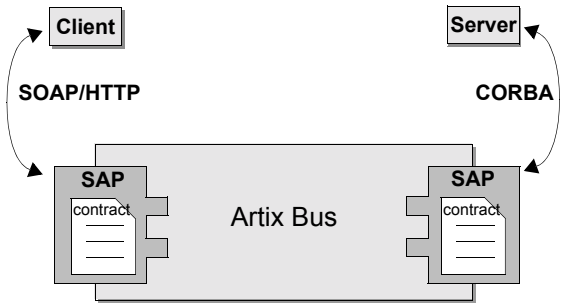

Figure 3: *The Artix Bus*

The rest of this chapter describes each of these components in more detail.

The Artix Bus

Overview

The Artix Bus is a set of plug-in that work in much the same way as simultaneous translators at the United Nations. The plug-ins read data that can be in a number of disparate formats, the Artix Bus directly translates the data into another format, and the plug-ins write the data back out to the wire in the new format.

In this way, Artix enables all of the applications in your company to communicate over the Web, without needing to understand SOAP or HTTP. It also means that clients can contact Web services without understanding the native language of the server handling requests.

Benefits

While other Web service product offerings provide some ability to expose enterprise applications as Web 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 between the enterprise application's native communication protocol and SOAP over HTTP, which is the prevalent protocol used for Web services.

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

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

Artix Service Access Points

Overview

An Artix Service Access Point (SAP) is where a service provider or service consumer connects to the Artix Bus. SAPs are described by a contract describing the services offered and the physical representation of the data on the network.

Reconfigurable connection

An Artix SAP provides an abstract connection point between applications, shown in Figure 3 on page 12. The benefit of using this abstract connection is that it allows you to change the underlying communication mechanisms without recoding any of your applications. You simply need to modify the contract describing the SAP.

For example, if one of your back-end service providers is a Tuxedo application and you want to swap a Tuxedo for a CORBA implementation, you would simply change the SAP's contract to contain a CORBA connection to the Bus. The clients accessing the back-end service provider never need to be aware that the application has changed.

Artix Contracts

Overview

Web Services Definition Language (WSDL) is used to describe the characteristics of the Service Access Points (SAPs) of an Artix connection. By defining characteristics like service operations and messages in an abstract way—independent of the transport or protocol used to implement the SAP—these characteristics can be bound to a variety of a specific protocols and formats. Artix 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 most simple Artix contract defines a set of systems with a shared interface, payload format, and transport. Artix contracts, however, can define very complex integration scenarios.

WSDL basics

Understanding Artix contracts requires some familiarity with WSDL. The key WSDL terms can be defined as follows:

WSDL types provide data type definitions used to describe messages.

A WSDL message is an abstract definition of the data being communicated and each part of a message is associated with defined types.

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 protocol and data format for operations defined in a portType.

A WSDL Port specifies a network address 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 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 (wire format and transport).

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.

Table 1: *Artix WSDL Contract Elements*

Logical contract:	
`<schema>`	
`<types>`	(analogous to typedefs)
`<message>`	(analogous to a parameter)
`<portType>`	(analogous to a class or CORBA interface definition)
`<operation>`	(analogous to a method)
Physical contract:	
`<binding>`	(payload format)
`<service>`	(groups of ports)
`<port>`	(transport addressing information)
`<route>`	(rules governing system interaction)

Payload formats

A payload format controls the layout of a message delivered over a transport. For example, SOAP payload controls the layout of messages delivered over HTTP. For more details, see "Supported transports and protocols" and "Supported payload formats" on page 5.

The WSDL definition of a port and its binding together associate a payload format with a transport. A binding can be specified in the logical portion of an Artix contract (`portType`). This allows a logical contract to have multiple bindings and thus enable multiple on-the-wire formats to use the same contract.

Further information

For more a more detailed introduction to the WSDL concepts used in Artix contracts, see Chapter 3.

CHAPTER 3

WSDL Concepts

Artix contracts are WSDL documents that describe logical abstract services and the data they use. This chapter provides a more detailed introduction to WSDL concepts.

In this chapter

This chapter discusses the following topics:

Web Services Description Language Basics

Overview

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) and is currently at revision 1.1. You can find the standard on the W3C website, www.w3.org.

The Artix Designer tool simplifies and automates the creation and management of Artix contracts. You do not need to be a WSDL expert to use Artix. However, a basic understand of WSDL concepts is recommended.

Web service endpoints and service access points

WSDL documents describe a service as a collection of *endpoints*. Each endpoint is defined by binding an abstract operation description to a concrete data format, and specifying a network protocol and address for the resulting binding.

Artix *Service Access Points* extend the concept of endpoint to include services that are available over any computer network, not just over the Web. A Service Access Point (SAP) can be bound to payload formats other than SOAP, and can use transports other than HTTP.

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 can be used to define several services that use the same abstract messages.

Port types

A *portType* is a collection of abstract operations that defines the actions provided by an endpoint. When a port type is mapped to a concrete data format, the result is a concrete representation of the abstract definition, in the form of an endpoint or service access point.

Concrete details

The mapping of a particular port type to a concrete data format results in a reusable *binding*. A *port* is defined by associating a network address with a reusable binding, and a collection of ports 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.

Elements of a WSDL document

A WSDL document is made up of the following elements:

`<types>`	The definition of complex data types based on in-line type descriptions and/or external definitions such as those in an XML Schema (XSD).
`<message>`	The abstract definition of the data being communicated.
`<operation>`	The abstract description of an action.
`<portType>`	The set of operations representing an abstract endpoint.
`<binding>`	The concrete or physical data format specification for a port type.
`<port>`	The endpoint defined by a binding and a physical address.
`<service>`	A set of ports.

Example

Example 1 shows a simple WSDL document. It defines a SOAP over HTTP Service Access Point that returns the date.

Example 1: *Simple WSDL*

```
<?xml version="1.0"?>
<definitions name="DateService"
   targetNamespace="urn:dateservice"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="urn:dateservice"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://iona.com/dates/schemas">
  <types>
    <schema targetNamespace="http://iona.com/dates/schemas"
   xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="dateType">
         <complexType>>
          <all>
            <element name="day" type="xsd:int"/>
            <element name="month" type="xsd:int"/>
            <element name="year" type="xsd:int" />
           </all>
         </complexType>
      <element>
    </schema>
  </types>
  <message name="DateResponse">
    <part name="date" element="xsd1:dateType" />
  </message>
  <portType name="DatePortType">
    <operation name="sendDate">
      <output message="tns:DateResponse" name="sendDate" />
    </operation>
  </portType>
```

Example 1: *Simple WSDL*

```
  <binding name="DatePortBinding" type="tns:DatePortType">
    <soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="sendDate">
      <soap:operation soapAction="" style="rpc" />
      <output name="sendDate">
        <soap:body
   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   namespace="urn:dateservice" use="encoded" />
      </output>
    </operation>
  </binding>
  <service name="DateService">
    <port binding="tns:DatePortBinding" name="DatePort">
      <soap:address location="http://www.iona.com/DatePort/" />
    </port>
  </service>
</definitions>
```

Data Type Definitions

Overview

Applications typically use data types that are more complex than the primitive types, such as `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 XML Schema Names Definition language (XSD) is the preferred canonical type system for a WSDL document. Therefore, XSD schemas are treated as the intrinsic type system. Because these data types are abstract descriptions of the data passed over the wire and 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 2, 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 2: *personalInfo*

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

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

Example 3 shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in Example 2. `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 3: *XSD type definition for personalInfo*

```
<types>
  <xsd:schema targetNamespace="http:\\iona.com\personal\schema"
   xmlns:xsd1="http:\\iona.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>
        <element name="name" type="xsd:string" />
        <element name="age" type="xsd:int" />
        <element name="hairColor" type="xsd1:hairColorType" />
      </ complexType>
    </ element>
  </ schema>
</ types>
```

Another way to map personalInfo is to describe `hairColorType` in-line as shown in Example 4. WIth this mapping, however, you cannot reuse the description of `hairColorType`.

Example 4: *Alternate XSD mapping for personalInfo*

```
<types>
  <xsd:schema targetNamespace="http:\\iona.com\personal\schema"
   xmlns:xsd1="http:\\iona.com\personal\schema"
   xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="personalInfo">
      <complexType>
        <element name="name" type="xsd:string" />
        <element name="age" type="xsd:int" />
```

Example 4: *Alternate XSD mapping for personalInfo*

```
        <element name="hairColor">
          <simpleType>
            <restriction base="xsd:string">
              <enumeration value="red" />
              <enumeration value="brunette" />
              <enumeration value="blonde" />
            </ restriction>
          </ simpleType>
        </ element>
      </ complexType>
    </ element>
  </ schema>
</ types>
```

Message Definitions

Overview

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 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, would be mapped to another message.

Example

For example, imagine a server that stored personal information as defined in Example 2 on page 24 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 5.

Example 5: *personalInfo lookup method*

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

This method signature could be mapped to the WSDL fragment shown in Example 6.

Example 6: *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. It is also recommended that messages are named in a way that represents 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 and an attribute specifying its data type. The data type attributes are listed in Table 2

Table 2: *Part Data Type Attributes*

Attribute	Description
`type="`*`type_name`*`"`	The data type of the part is defined by a `simpleType` or `complexType` called *`type_name`*
`element="`*`elem_name`*`"`	The data type of the part is defined by an `element` called *`elem_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 7.

Example 7: *Reused part*

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

Interface Definitions

Overview

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

Port types

A portType can be thought of as an interface description and in many Web service implementations there is a direct mapping between port types and implementation objects. Port types 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.

Port types are described using the <portType> element in a WSDL document. Each port type 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

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 ten widgets can both be defined as operations.

Each operation within a port type must have a unique name, specified using the name attribute. The name attribute is required to define an operation.

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 3.

Table 3: *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.

Table 3: *Operation Message Elements*

Element	Description
<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 inTable 4.

Table 4: *Attributes of 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 port type is an abstract definition of the data passed during in 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 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 the <binding> section.

Example

For example, in implementing a server that stored personal information in the structure defined in Example 2 on page 24, you might use an interface similar to the one shown in Example 8.

Example 8: *personalInfo lookup interface*

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

This interface could be mapped to the port type in Example 9.

Example 9: *personalInfo lookup port type*

```
<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>
```

Physical Definitions

Overview

The abstract definitions in a WSDL document (types, messages, and port types) are intended to be used to define the interaction of real applications. These applications have specific network addresses, use specific network protocols, and expect data in a particular format.

To fully define real applications, abstract definitions must be mapped to concrete representations of the data passed between the applications, and the details of the network protocols need to be added. This is done using WSDL bindings and ports.

Binding syntax

WSDL binding and port syntax is not tightly specified by W3C. While there is a specification defining the mechanism for defining the syntax, the syntax for bindings other than SOAP, and network transports other than HTTP, are not bound to a W3C specification.

Bindings

To define an endpoint that corresponds to a running service, port types are mapped to bindings which describe how the abstract messages defined for the port type map to the data format used on the wire. The bindings are described in `<binding>` elements. A binding can map to only one port type, but a port type can be mapped to any number of bindings.

It is within the bindings that details such as parameter order, concrete data types, and return values are specified. For example, the parts of a message can be reordered in a binding 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.

Ports and services

The final piece of information needed to describe how to connect a remote service is the network information needed to locate it. This information is defined inside a <port> element. 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 port type, but use different network protocols, like HTTP and WebSphere MQ.

CHAPTER 4

The Artix Designer

The Artix Designer GUI tool simplifies the creation and management of Artix contracts. It also enables you to generate source code, and runtime configuration files for your Artix integration solution. This chapter provides a quick overview of the main features in Artix Designer.

In this chapter

This chapter discusses the following topics:

The Artix Designer Environment

Overview

Artix Designer is a GUI tool for creating, managing, and deploying Artix contracts. It provides editing tools for creating contracts from standard WSDL and CORBA IDL. Artix Designer also provides wizards to walk you through each step of key tasks—for example, defining new data types, logical interfaces, payload bindings, and transports.

Artix Designer generates all the components that you need to complete your Artix solution, for example:

- Contracts describing your services.
- Stub or skeleton code for writing application code.
- Configuration files to deploy applications.

Artix Designer can also generate CORBA IDL from any contracts that have a CORBA binding.

Artix Welcome dialog

When you start Artix Designer, the **Artix Welcome** dialog is displayed, as shown in Figure 4. This enables you specify whether to create a new or open an existing workspace, start Artix Designer, or view a demo.

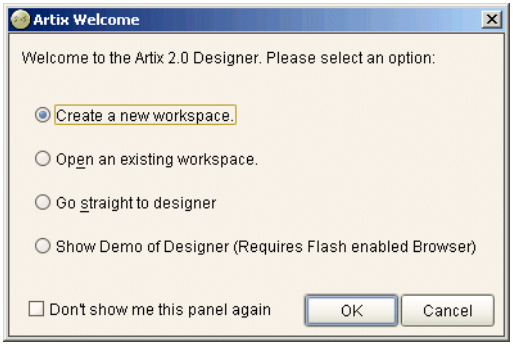

Figure 4: *Welcome Dialog*

New Workspace wizard

An Artix workspace defines the structure of your Artix solution, and includes all your WSDL contracts. Artix Designer provides a range of wizard templates to help you get started.

For example, you can create a new workspace, a C++ Web Services client, or a C++ Web services client and server. Figure 5 shows the new workspace wizard selected in the **Select WS Type** dialog.

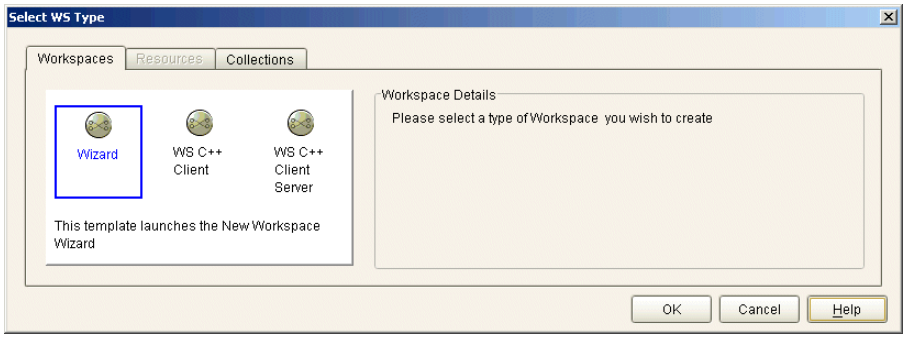

Figure 5: *Select WS Type Dialog*

Designer Tree

The Designer Tree is a navigation tree displayed on the left side of Artix Designer. The Designer Tree displays the following information:

Workspace

An Artix Workspace includes all the WSDL contracts in your Artix solution.

The Designer Tree displays the workspace name (for example, `HelloWorld`), and contains its collections and shared resources.

Shared Resources

These are all the WSDL contracts that you want to work with.

Shared resources are also displayed within collections, by italicized text and a dimmed icon.

If you click on a shared resource, the pane on the right of the screen displays the WSDL view of that resource.

Collections

These are groups of WSDL contracts that are organized into logical collections for deployment purposes. A collection maps to an executable or process that implements the WSDL defined in it.

You can drag and drop resources between collections and also from the shared resource folder to a collection.

If you click on a collection, the pane on the right of the screen displays the details of that collection.

The **Current View** drop-down list at the bottom of the tree filters the amount of detail shown in the tree. The default is to show all information in the workspace. You can select to view only collections or shared resources. Figure 6 shows a simple example of a workspace named `GoogleSearch` displayed in the Designer Tree.

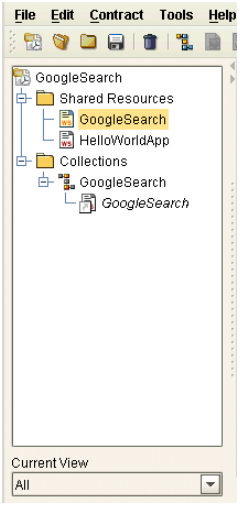

Figure 6: *Designer Tree*

Artix Designer main window

Figure 7 displays the entire Artix Designer main window. The right-hand pane displays summary information for items displayed in the Designer Tree. For example, clicking on your workspace folder in the tree displays the **Workspace Details**, shown in Figure 7.

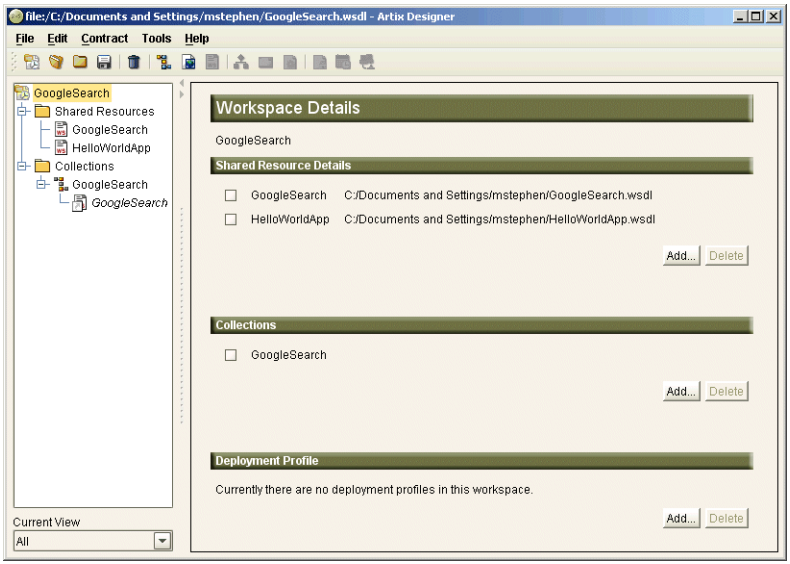

Figure 7: *Artix Designer Main Window*

Similarly, clicking on the **Shared Resources** or **Collections** folder displays summary information for these items in right pane.

The Contract Editor

Overview

The Contract Editor is the engine room of the Artix Designer GUI. It has two main purposes—firstly, it provides a way for you to navigate around the various components of your WSDL contract. Secondly, it provides you with editing tools to add or update components in your Artix contract.

Contract Editor

Clicking on a WSDL contract in the Designer Tree displays a graphical view of the contract in the right-hand pane. This is the Contract Editor **Graph** view, shown in Figure 8.

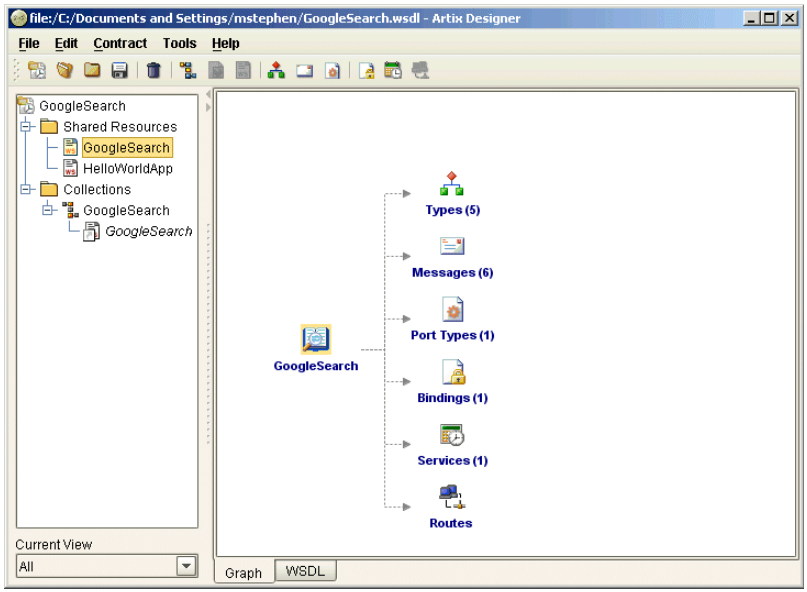

Figure 8: *Contract Editor—Graph View*

The icons representing the contract elements (types, messages, services, and so on) in the graphical view may have a small plus sign attached. This indicates that the element has children.

You can view children (types, messages, and so on) by double clicking on the element icon. You can then view or edit the individual items directly from the Contract Editor. Each child component has an associated dialog to enable viewing or editing. For example, double clicking on a type launches the **Edit Type Attributes** dialog, shown in Figure 9.

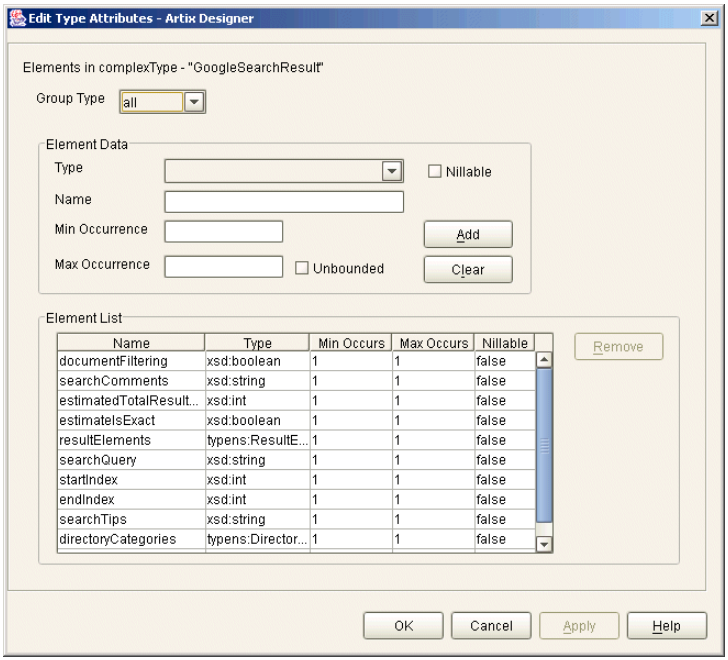

Figure 9: *Edit Type Attributes Dialog*

Working with WSDL

The Contract Editor also enables you to view and edit the contract WSDL directly instead of working through the graphical representation as previously described.

To access the **WSDL** view of the contract, shown in Figure 10, click on the **WSDL** tab at the bottom of the Contract Editor pane.

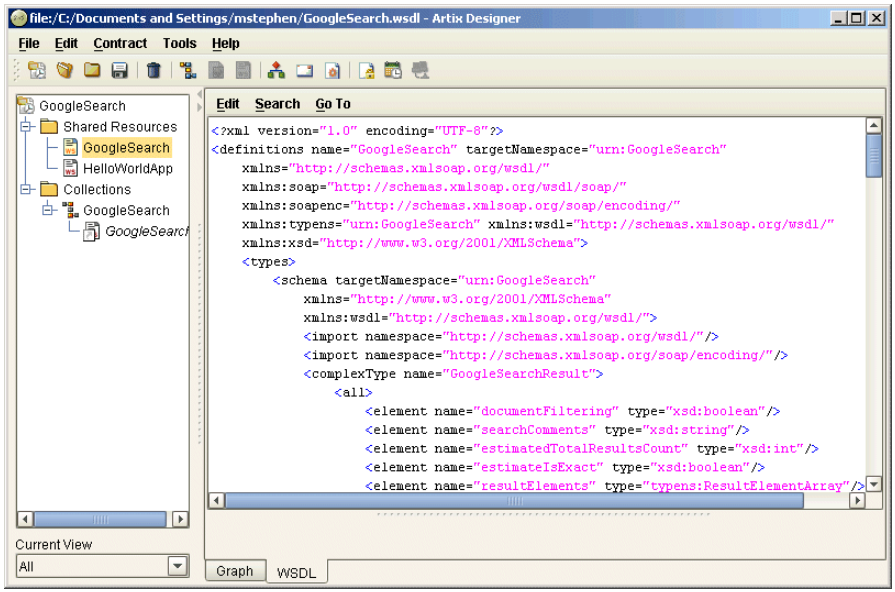

Figure 10: *Contract Editor—WSDL View*

Working in the WSDL view of the contract requires a sound knowledge of WSDL. Be aware that if you make changes to the WSDL, it could easily invalidate your contract.

If you do make a change to the WSDL that causes a problem, errors are identified in a separate **ERRORS** panel directly under the WSDL. This enables you to easily identify the exact position of the problem within the WSDL file.

Artix Deployment Tools

Overview

Artix Designer provides deployment tools that enable you to generate Artix stub and skeleton code and Artix configuration scripts for deployment on different platforms.

Deployment Profiles

The **Deployment Profile** wizard enables you to generate a reusable deployment profile for your WSDL collection. A deployment profile defines machine-specific details, for example:

- operating system
- location of the Artix installation
- programming language

A deployment profile can be reused and is not specific to any particular collection. Typically, you would have one deployment profile for every deployment machine.

Figure 11 shows the first screen of the **Deployment Profile** wizard.

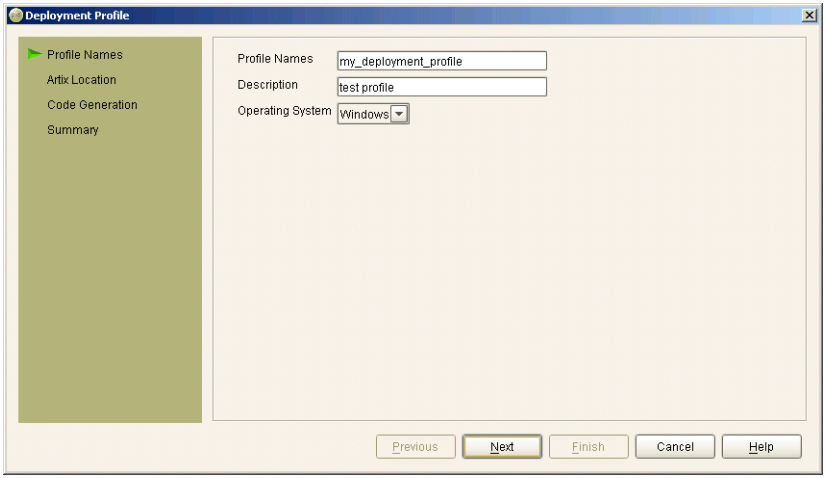

Figure 11: *Deployment Profile Wizard*

Deployment Bundles

The **Deployment Bundle** wizard enables you to create a reusable deployment bundle for your WSDL collection. A deployment bundle defines a collection's deployment-specific details, for example:

- deployment type (client, server, or middleware switch)
- code generation options (C++, Java, or IDL)
- configuration options (security, management, locator, session management).

Note: You can create multiple deployment bundles for a collection, but you must create at least one deployment profile before creating a bundle.

Figure 12 shows the first screen of the **Deployment Bundle** wizard.

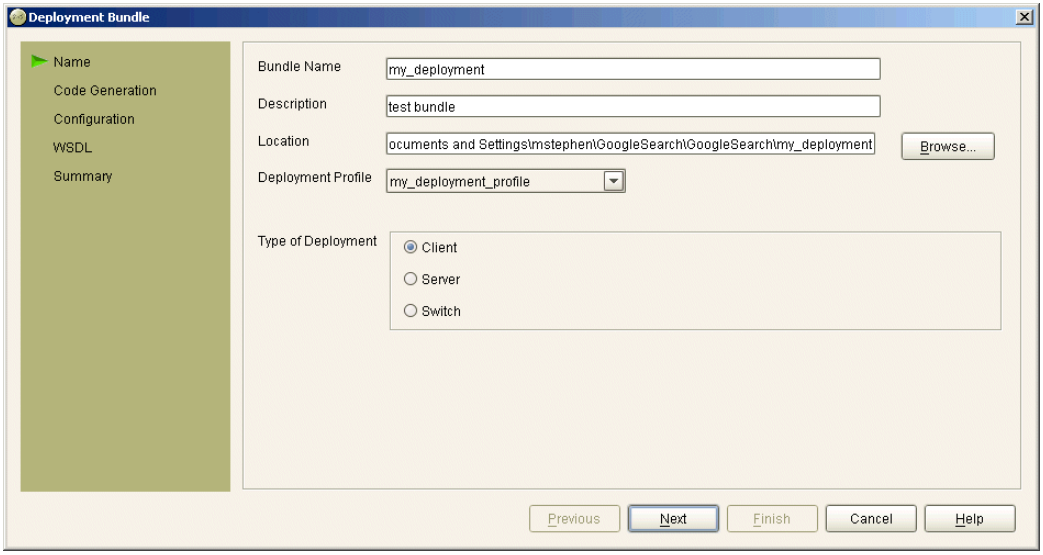

Figure 12: *Deployment Bundle Wizard*

Artix Deployer

When you have created your deployment profile and bundle, you can then use the **Run Deployer** dialog to deploy your WSDL collection. This deploys your solution based on the information that you provided in the deployment bundle. It generates the code, environment scripts, and configuration files in the locations that you specified.

When your collection has successfully deployed, a green box is displayed around the collection in the Designer Tree. If you make any change to the collection after it has been deployed, you must deploy it again. This state is indicated by a broken-lined red box around the collection icon to remind you to redeploy the collection.

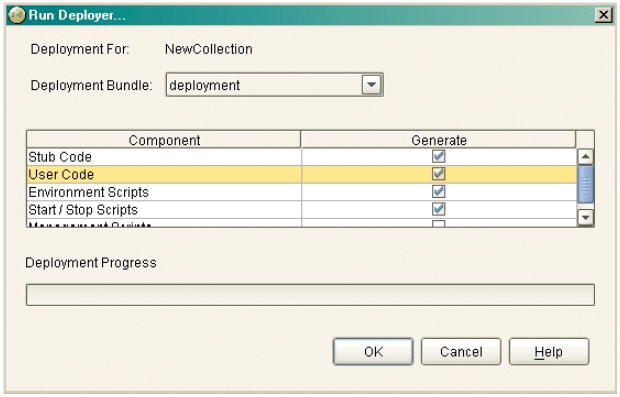

Figure 13: *Run Deployer Dialog*

Further information

For more detailed information, and step-by-step examples of how to use Artix Designer, see *Designing Artix Solutions with Artix Designer*.

Artix Designer also provides online help, available from the Artix Designer **Help** menu. You can access context-sensitive help by selecting the **Help** buttons available on Artix Designer dialog boxes.

Glossary

A

Artix Designer

A suite of GUI tools for creating, managing, and deploying Artix integration solutions.

B

Binding

A binding associates a specific transport/protocol and data format with the operations defined in a <portType>.

Bus

See Service Bus

Bridge

A usage mode in which Artix is used to integrate applications using different payload formats.

C

Collection

A group of related WSDL contracts that can be deployed as one or more physical entities such as Java, C++, or CORBA based applications. It can also be deployed as a switch process.

Connection

An established communication link between any two Artix endpoints.

Contract

An Artix contract is a WSDL file that defines the interface and all connection-related information for that interface. A contract contains two components: logical and physical. The logical contract defines things that are independent of the underlying transport and wire format, and is specified in the <portType>, <operation>, <message>, <type>, and <schema> WSDL tags.

The physical contract defines the payload format, middleware transport, and service groupings, and the mappings between these things and portType 'operations.' The physical contract is specified in the <port>, <binding> and <service> WSDL tags.

Contract Editor

A GUI tool used for editing Artix contracts. It provides several wizards for adding services, transports, and bindings to an Artix contract.

D

Deployment Mode

One of two ways in which an Artix application can be deployed: Embedded and Standalone. An embedded-mode Artix application is linked with Artix-generated stubs and skeletons to connect client and server to the service bus. A standalone application runs as a separate process in the form of a daemon.

E

Embedded Mode

Operational mode in which an application creates a Service Access Point, either by invoking Artix APIs directly, or by compiling and linking Artix-generated stubs and skeletons to connect client and server to the service bus.

Endpoint

The runtime deployment of one or more contracts, where one or more transports and its marshalling is defined, and at least one contract results in a generated stub or skeleton (thus an endpoint can be compiled into an application). Contrast with Service.

H

Host

The network node on which a particular service resides.

M

Marshalling Format

A marshalling format controls the layout of a message to be delivered over a transport. A marshalling format is bound to a transport in the WSDL definition of a Port and its binding. A binding can also be specified in a logical contract portType, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

P

Payload Format

The on-the-wire structure of a message over a given transport. A payload format is associated with a port (transport) in the WSDL using the binding definition.

Protocol

A protocol is a transport whose format is defined by an open standard.

R

Routing

The redirection of a message from one WSDL binding to another. Routing rules are specified in a contract and apply to both endpoints and standalone services. Artix supports port-based routing and operation-based routing defined in WSDL contracts. Content-based routing is supported at the application level.

Router

A usage mode in which Artix redirects messages based on rules defined in an Artix contract.

S

Service

An Artix service is an instance of an Artix runtime deployed with one or more contracts, but with no generated language bindings. The service has no compile-time dependencies. A service is dynamically configured by deploying one or more contracts on it.

Service Access Point

The mechanism, and the points at which individual service providers and consumers connect to the service bus.

Service Bus

The set of service providers and consumers that communicate via Artix. Also known as an Enterprise Service Bus.

Standalone Mode

An Artix instance running independently of either of the applications it is integrating. This provides a minimally invasive integration solution and is fully described by an Artix contract.

Switch

A usage mode in which Artix connects applications using two different transport mechanisms.

System

A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-in

A plug-in module that provides wire-level interoperation with a specific type of middleware. When configured with a given transport plug-in, Artix will interoperate with the specified middleware at a remote location or in another process. The transport is specified in the `<port>` element of a contract.

W

Workspace

The Artix Workspace defines the structure of your Artix solution. It is the first thing you need to create when using the Designer, and all of the solution's components are included within it.

A workspace will typically have one or more collections, which in turn contain resources that define your solution's interface. A workspace also contains shared resources that are common across one or more collections.

Index